INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet Système Informatique

Rapport

https://github.com/thomasclgnt/PSI.git

Thomas CAYLA GINESTET
Marie MECALIFF

4 IR Groupe B2

Année 2022 - 2023

Table des matières

Introduction

L'objectif du Projet de Système Informatique, effectué dans le cadre de la 4ème année d'Informatique et Réseaux, est de réaliser un système informatique complet, composé d'un compilateur utilisant le générateur d'analyseur lexical LEX et le générateur d'analyseur syntaxique YACC, ainsi qu'un microprocesseur codé en VHDL.
Le compilateur traduit un langage source, qui sera une version simplifiée du langage C, en un langage cible qui constituera un jeu d'instructions assembleur orienté mémoire. Le microprocesseur quant à lui sera basé sur un jeu d'instructions orienté registre, afin de faciliter son implémentation. Un cross-compilateur est donc nécessaire afin d'effectuer la traduction du jeu d'instructions pour permettre sa transmission entre ces deux composants du système global.

Dans ce rapport, nous souhaitons présenter de manière rétrospective le processus de conception qui a guidé la réalisation des différents composants de notre système informatique. Nous détaillerons également notre processus de réflexion, en expliquant nos choix et en abordant les spécificités des langages de programmation que nous avons utilisés.

I Conception et implémentation du compilateur

Dans cette partie, nous aborderons la conception d'un compilateur pour une version simplifiée du langage C. Notre objectif est de développer un composant capable de traduire ces expressions en code assembleur correspondant. Ensuite, nous avons mis en place un interpréteur pour le langage assembleur, afin de pouvoir exécuter le code généré par le compilateur.

a) Analyse syntaxique - Lex

Afin de mettre en œuvre les règles lexicales et syntaxiques nécessaires à l'analyse d'un code source, nous avons commencé par décrire dans un fichier "lex.l" les définitions permettant d'identifier les différents éléments du langage. Par exemple les identificateurs valides (composés de lettres, chiffres et underscore), ou encore les nombres entiers, y compris les notations hexadécimales. Nous avons ensuite décrit les règles lexicales nécessaires pour analyser le code source d'entrée, et identifier les différents éléments du langage, tels que les mots-clés, les opérateurs et les délimiteurs. Ces règles lexicales sont définies à l'aide de motifs (ou patterns) et d'actions associées.

b) Parser et analyse grammaticale – Yacc

Le processus de conception du compilateur implique également la mise en œuvre de règles syntaxiques, pour analyser le code source et décrire la structure grammaticale du langage cible.

Le fichier YACC correspondant commence par la déclaration des symboles, tables et fichiers externes utilisés. Ensuite, la déclaration des unions, des types de tokens et des règles de priorité est spécifiée. Les règles de grammaire y sont définies à l'aide des non-terminaux et des terminaux du langage, chaque règle correspondant à une structure syntaxique spécifique, comme "Programme" ou "Function".

Des actions sont également associées à certaines règles, indiquées entre accolades {}. Ces actions sont exécutées lors de l'analyse syntaxique et nous ont permis de générer pas à pas le code assembleur correspondant, mais aussi de mettre à jour au fur et à mesure certaines informations telles que la profondeur globale du code à un endroit donné.

Ainsi, nous avons pris en compte dans la grammaire les éléments suivantes :

- Les **déclarations** et **initialisations de variables** : Ces dernières peuvent se faire ensemble ou séparément, notre compilateur permet également d'effectuer plusieurs exécutions à la suite (ex. : int a, b ;). Notre compilateur ne reconnaît que le type int mais accepte également les constantes.
- Les expressions arithmétiques et conditionnelles : Notre compilateur gère ainsi les quatres expressions arithmétiques principales. En établissant une hiérarchie des tokens utilisés dans les opérations arithmétiques, nous garantissons les priorités des opérations (multiplication et division d'abord, puis addition et soustraction). Cette approche vise à éviter certains conflits de décalage ou de réduction des règles définies dans le Yacc.
- Les ***printf*** de variables : La fonction printf a été gérée de manière à ce qu'elle n'accepte qu'un seul paramètre, correspondant à une variable dont la valeur doit être affichée.
- Les **conditions *if/else*** : Ces structures nécessitent de gérer les différents sauts liés à la condition évaluée. Nous avons ainsi créé des règles auxiliaires (index_jump, prof_p, …) que nous avons pour certaines associées à un type "<index>", afin de pouvoir y enregistrer des informations contextuelles telles que le numéro de ligne assembleur correspondant à l'instruction "jump", pour pouvoir plus tard corriger l'adresse de destination non définie quand on atteint fin du corps du "if", pour pouvoir y sauter directement en cas de condition non satisfaite. Dans le cas d'un "else", cela nous permet également, si la condition est vérifiée, de sauter après le corps de ce else quand nous atteignons la fin du corps principal du "if".
- Les **boucles *while*** : De la même manière que pour les conditions *if/else*, deux numéros d'instructions sont sauvegardés grâces aux règles "*index_jmp*" et "*index_jmf*". Ainsi, les deux instructions de sauts sont plus tard corrigées : la première quand la fin de la boucle est atteinte, pour y sauter si la condition n'est plus respectée, et le second à un saut vers l'instruction précédant la condition booléenne, produisant le phénomène de boucle, afin de réévaluer la condition à chaque itération.

Enfin, nous avons réalisé plusieurs programmes de test ciblés sur les différents éléments à traiter, pour évaluer les performances et la fiabilité du compilateur et de l'interpréteur que nous avons développés au fur et à mesure de leur implémentation.

c) Table des symboles

Le jeu d'instructions obtenu en sortie du compilateur est orienté mémoire. En effet, les données manipulées dans le code analysées sont rangées dans une mémoire de données appelées "table des symboles", avec une première zone réservée aux variables et constantes déclarées et une deuxième zone pour les résultats temporaires, dans laquelle il faut également gérer la libération de l'espace mémoire.

Nous avons fait le choix d'utiliser une pile pour conserver ces données. En effet, cette approche facilite la gestion de la mémoire en offrant un accès facile aux variables récentes, sans avoir à retenir des index afin d'accéder aux variables, et en permettant une libération efficace des variables temporaires avec la fonction pop(). Cette approche garantit une libération instantanée de la mémoire sans perturber l'ordre des variables restantes. De plus, la représentation par pile automatise la gestion des variables temporaires, optimisant ainsi l'utilisation de la mémoire : les variables temporaires utilisées pour les opérations en cours sont créées, utilisées et libérées au fur et à mesure de l'avancement du code, car elles se situent donc toujours sur le dessus de la pile. De fait, nous appliquons la même approche dans notre fichier YACC pour les variables déclarées (token tID) et les valeurs "brutes" (token tNB), en créant des variables temporaires correspondantes lors qu'elles sont rencontrées à droite d'une affectation. Cela permet de faciliter l'analyse et l'exécution du code en homogénéisant le traitement d'une expression du côté de la mémoire.

Enfin, nous avons implémenté la profondeur des variables. De fait, quand le parser rencontre des structures, la profondeur globale du code est incrémentée et elle sera appliquée et sauvegardée pour toute variable définie à l'intérieur de cette structure afin de conserver l'information sur sa portée et d'avoir une

meilleure visibilité de notre position. Cette gestion de la profondeur est cruciale tout au long de l'analyse du code car elle permet d'éviter d'alourdir la table des symboles, en supprimant alors les variables locales à une fonction ou une structure dès sa sortie. L'objectif de cette approche est de simuler au mieux le fonctionnement du processeur lorsqu'il exécute du code en langage C.

d) Table d'instructions

Le but final du compilateur est de générer les instructions assembleur correspondant au code analysé dans le langage cible. Au fil de l'analyse du code par le parser YACC, les instructions associées qui sont engendrées sont stockées dans un tableau de 4 colonnes dont l'index correspond au "numéro d'instruction" associé à chacune. Les instructions sont générées dans un format simplifié, où des opcodes sont utilisés pour identifier chaque opération prise en charge. Le tableau de nos correspondances est en annexe du rapport.

Ce compilateur manipulant un jeu d'instructions orienté mémoire et ne considérant pas les registres, LOAD et STORE ne sont utilisés que dans le contexte du processeur et du cross-assembleur. De même, les instructions RET, PUSH, POP et CALL sont reversées au traitement des fonctions, qui est abordé de manière approfondie dans une partie ultérieure du rapport. Enfin, bien que non présentes dans le sujet, nous avons décidé d'implémenter le traitement des opérations ">=", "<=" et "!=" dans notre compilateur.

Pour la gestion des structures de contrôle conditionnelles et des boucles, trois fonctions spécifiques ont été ajoutées au fichier de la table d'instruction : patch_jmf(), patch_jmp() et get_index(). En effet, au moment de la génération des instructions de sauts qu'impliquent ces structures, nous ne connaissons pas encore leur adresse de destination car celle-ci dépend de l'évaluation de la condition du "if" ou bien du while, et de la taille du corps de la structure à exécuter ou ou bien à sauter. Par conséquent, nous générons des instructions de sauts avec des adresses de destination non définies dont nous conservons l'index, comme expliqué précédemment. Une fois que l'ensemble du code concerné est analysé et donc que les adresses finales des différentes parties du bloc sont connues, une opération de "patch" est effectuée pour corriger les sauts à l'aide de leur numéro d'instruction sauvegardé, en remplaçant les adresses non définies par les adresses correctes. Cette étape d'ajustement permet de garantir que le flux d'exécution du programme se déroule de manière appropriée en fonction des conditions évaluées.

e) Gestion des fonctions

Nous avons décidé d'enrichir notre compilateur en ajoutant une nouvelle fonctionnalité : le traitement des appels de fonction. Les appels récursifs, les appels de fonctions de type int ou void, avec ou sans paramètres, sont également pris en compte dans notre implémentation, pour laquelle nous nous sommes appuyés sur le fonctionnement des processeurs assembleurs.

Tout d'abord, comme les fonctions étaient déclarées avant le main, nous avons initialisé notre tableau d'instruction avec un saut vers la première instruction du main, avec une adresse de destination initialement indéfinie que nous venons "patch" plus tard dans l'analyse du code quand nous rencontrons le main. En effet, cette inclusion est nécessaire car la fonction main doit être exécutée en premier, avant les fonctions qu'elle invoque.

Pour chaque fonction rencontrée lors de l'analyse, nous stockons dans un tableau dédié leur nom ou "ID" ainsi que leur première instruction assembleur, représentant le début de la fonction. Deux entrées sont ajoutées pour l'adresse et la valeur de retour dans la table des symboles, afin de leur réserver un espace dans la mémoire, et une instruction de retour est ajoutée en fin de fonction, moment auquel on supprime également les variables locales de la pile, grâce à la gestion de la portée des données.

Le reste de la compilation se déroule normalement, jusqu'à la rencontre d'un appel de fonction : afin de préparer le cadre pour la fonction appelée, nous sauvegardons la taille de la table des symboles, nous réservons de l'espace pour sa valeur et son adresse de retour, puis nous ajoutons dans la table d'instruction une opération correspondant à cet appel. Il faut ensuite libérer autant de variables temporaires dans la

table des symboles que de paramètres lui ont été passés en argument, avant d'à nouveau restaurer le cadre pour la fonction appelante.

f) Interpréteur

Dans un troisième temps, nous nous sommes concentrés sur le développement de l'interpréteur du langage assembleur précédemment généré. L'objectif de cet interpréteur est d'exécuter le code et donc chaque opération correspondant aux instructions données en entrant.

Ainsi, un fichier contenant toutes les instructions correspondant à la traduction du code initial dans le langage cible, est finalement généré par le compilateur à la fin de son analyse. Le fichier d'instructions est ensuite donné en entrée et lu par l'interpréteur, qui analyse chaque ligne et remplit le tableau représentant sa ROM (Read-Only Memory, contenant donc les instructions à lire) en conséquence. Un second tableau est défini par un tableau pour représenter la mémoire RAM (Random Access Memory). Ensuite, les instructions contenues dans le tableau ROM sont exécutées séquentiellement jusqu'à ce qu'un NOP (opération nulle) représentant la fin du code soit rencontré. Les différentes opérations arithmétiques et de contrôle sont donc effectuées une à une par l'interpréteur qui remplit son tableau RAM en conséquence, et affiche les variables à l'écran quand cela le lui est demandé. Cette phase marque la finalisation du processus de développement du compilateur.

L'interpréteur a été amélioré pour gérer les appels de fonctions, en ajoutant les instructions RET, CALL, PUSH et POP, spécifiques à cette fonctionnalité. Nous avons également introduit la variable globale "ebp" pour simuler un pointeur de base vers la mémoire RAM, bien que celle-ci soit en réalité représentée par un tableau. Cette variable est utilisée pour accéder aux variables locales et aux paramètres de fonction en déterminant leur emplacement relatif par rapport à la valeur actuelle de ebp. L'instruction PUSH met à jour ebp en ajoutant l'offset spécifié, tandis que l'instruction POP le soustrait. Lors d'un appel de fonction avec CALL, l'adresse de retour, qui correspond au numéro de l'instruction suivante, est stockée à l'adresse pointée par ebp avant de sauter à l'adresse de la fonction appelée indiquée en paramètre de l'instruction. Enfin, l'instruction RET lit l'adresse de retour à l'index ebp de la mémoire et permet ainsi de revenir à la fonction appelante. Une valeur de "-1" à cet emplacement indique la fin de l'exécution lorsque la fonction main se termine : la valeur l'adresse 0 du tableau mémoire, où ebp pointe au départ, est en effet initialisée "-1" en début d'exécution.

g) Problèmes identifiés, solutions adoptées et limites de l'implémentation

L'implémentation des appels de fonctions s'est avérée être un défi majeur, avec plusieurs difficultés rencontrées :

- Pour gérer le retour d'une fonction et la libération des variables temporaires, nous avons calculé la différence entre l'état actuel de la pile et son état au moment de l'appel, en soustrayant également les deux variables réservées à l'adresse et à la valeur de retour. Cela nous a permis de déterminer le nombre de paramètres passés en argument et donc de retirer le bon nombre de variables dans la table des symboles.
- Pour éviter les conflits créés par l'ajout d'actions en C entre l'analyse de deux tokens dans le Yacc, nous avons introduit des règles intermédiaires (par exemple "prof_p" et "index_jmf") permettant un déroulement fluide de l'interprétation.

Enfin, malgré l'inclusion de la définition des constantes dans le lex et le bison, nous n'avons pas pu implémenter la gestion complète de leur immutabilité dans notre compilateur par manque de temps. Cependant, nous avons prévu de stocker le type des variables dans la structure "stack" de notre table des symboles. En attribuant le type 2 aux constantes plutôt que le type 1 réservé aux entiers classiques, nous aurions pu les traiter différemment, par exemple en générant une erreur lors de toute tentative de modification d'une variable de type 2 au cours de l'analyse du code.

II Conception et implémentation du micro-processeur

Dans cette partie, nous aborderons la conception et l'implémentation d'un microprocesseur avec pipeline en VHDL. Notre objectif était de développer un composant correspondant à un jeu d'instructions orienté registre, capable d'exécuter différentes instructions assembleur, dont notamment l'addition, la soustraction, la multiplication, la copie et l'affectation.

a) ALU

L'ALU (Arithmetic Logic Unit) de notre processeur est responsable de l'exécution des opérations arithmétiques et logiques. L'ALU fournit les résultats attendus de ces opérations selon les instructions et les signaux de contrôle.

Dans cette partie, nous n'avons implémenté que trois opérations arithmétiques : l'addition, la multiplication et la soustraction associés, ainsi que des drapeaux déclarant une sortie négative ou nulle, une retenue sur l'addition ou un overflow.

Malgré nos essais, nous n'avons pas réussi à mettre en place la division, qui aurait pu être approximée par une division par des multiples de deux, par décalages successifs de bit vers la droite. Nous n'avons pas non plus défini les fonctions logiques qui devront être implémentées afin de complexifier encore notre processeur.

b) Banc de registre

Contrairement à un jeu d'instructions orienté mémoire comme celui généré par notre compilateur, un jeu d'instructions orienté registre privilégie l'utilisation intensive des registres du processeur. Ainsi, deux nouvelles instructions LOAD et STORE sont introduites, elles seules permettant un échange avec la mémoire de données.

Le banc de registre de notre processeur est un composant qui stocke et gère les données pendant l'exécution du programme. Il est composé de 16 registres, chacun capable de stocker une valeur de 8 bits. Il possède un double accès en lecture et un accès en écriture, avec la possibilité de lire deux valeurs différentes simultanément.

Notre banc de registre comporte une protection préventive en cas d'aléas de données par une fonctionnalité de bypass. Le bypass détecte si un accès en lecture et en écriture sont effectués simultanément sur le même registre, et dans le cas échéant gère cet accès simultané en propageant la nouvelle donnée d'écriture dans la sortie lecture. Cette protection nous permet ensuite de simplifier la gestion générale des aléas dans l'unité de contrôle.

c) Mémoires de données et d'instructions

Dans notre architecture, nous disposons de deux types de mémoires : une pour les données et une pour les instructions.

Notre implémentation de la mémoire de données permet à la fois de lire et d'écrire des données, de manière synchronisée avec l'horloge, en maintenant un état cohérant des données stockées dans la mémoire. Cette dernière est représentée par un tableau de 256 éléments, correspondant chacun à un octet : sa taille de 256 octets nous permet donc de stocker un volume significatif de données pour les besoins de notre système.

Dans notre implémentation de la mémoire d'instructions, nous avons utilisé une structure simplifiée qui représente une mémoire de type ROM (Read-Only Memory). Contrairement à la mémoire de données, nous supposons que le programme à exécuter par le microprocesseur est déjà stocké dans cette mémoire et qu'aucune modification du contenu n'est autorisée pendant l'exécution. Sans cross-assembleur, il nous a donc fallu initialement remplir cette mémoire manuellement pour les différents tests effectués sur notre

processeur. Tout comme pour la mémoire de données, la lecture des instructions se fait de manière synchrone avec l'horloge CLK, et elles se trouvent stockées dans un tableau de 256 éléments, représentant chacun une instruction sur 32 bits du programme, exécutée selon les besoins du microprocesseur.

d) Chemin de données et unité de contrôle

Pour la réalisation d'un microprocesseur avec pipeline, nous avons mis en place une architecture à cinq étages comprenant la lecture des instructions (LI), le décodage et l'interprétation (DI), l'exécution (EX), l'accès à la mémoire (MEM) et la gestion des résultats (RE).

L'implémentation d'un chemin de données et d'une unité de contrôle était nécessaire pour permettre la propagation des données à travers les différents étages du pipeline du processeur et assurer l'exécution correcte des instructions. Le chemin de données relie les composants et permet la transmission efficace des données, exploitant ainsi les ressources matérielles du processeur. Les cinq étages du pipeline permettent à chaque instruction de progresser d'un étage à l'autre à chaque cycle d'horloge, augmentant ainsi le débit d'instructions. Le chemin de données et l'unité de contrôle prennent en charge les opérations et les signaux de commande spécifiques à chaque étape du traitement des instructions, en fonction de l'instruction en cours d'exécution.

Ainsi, notre microprocesseur est capable de traiter les instructions suivantes : ADD, SUB, MUL, COPY, AFC, JMP et JMF. Bien que l'implémentation des instructions de saut JMP ne fût pas spécifiée dans les consignes, nous les avons intégrées afin d'enrichir cette partie du système.

Enfin, nous avons ajouté à cette étape de la conception le composant "Compteur_IP", pour gérer l'incrémentation de l'adresse de l'instruction à évaluer à chaque tic d'horloge.

e) Gestion des aléas

Suite à l'implémentation du pipeline, nous avons été confrontés à des situations d'aléas de données, se produisant lorsqu'une dépendance existe entre une instruction de lecture et une instruction d'écriture sur le même registre, nécessitant une temporisation pour éviter les conflits. L'instruction de lecture problématique se situe alors forcément à l'étage LiDi, tandis que l'aléa peut survenir si l'instruction d'écriture problématique se situe au même moment à l'étage DiEx ou bien ExMem.

Pour résoudre le problème, plutôt que d'implémenter une entité de gestion des aléas à part entière, nous avons préféré simplifier le code en supervisant ces aléas par le biais de signaux spécifiques. La solution consiste donc à introduire des "bulles" dans le pipeline lorsqu'un aléa est détecté, permettant de temporiser l'exécution des étages inférieurs tant qu'un conflit subsiste avec les étages supérieurs. Ces bulles sont représentées par l'injection d'une instruction NOP, représentant une absence d'opération. À la détection de l'aléa, nous inhibons également l'incrémentation dans le compteur IP ainsi que la sortie de la mémoire d'instruction, afin de n'en perdre aucune dans le processus.

Suite à l'implémentation des instructions de saut, nous avons dû prendre en compte la gestion des aléas de branchement associés. Ces aléas surviennent lorsque des instructions de saut conditionnel (JMF), par exemple, entrent dans le pipeline sans que nous sachions si le saut doit être réalisé ou non, car la condition sur laquelle il repose n'a pas encore été évaluée. Pour gérer ces situations, nous avons mis en place plusieurs mécanismes, à commencer par l'inhibition de l'incrémentation du compteur IP et de la sortie de la mémoire d'instruction lorsqu'une opération de saut entre dans l'étage LiDi. Cela permet de suspendre temporairement l'exécution des instructions suivantes. Une fois que l'instruction de saut a été propagée à l'étage DiEx, l'étage LiDi est bloqué et commence à injecter des instructions NOP pour occuper les cycles d'horloge jusqu'à ce que l'instruction de saut soit détectée à la sortie de l'étage MemRe. Si le saut peut être effectué, donc dans le cas du JMF, si la condition évaluée est fausse, le compteur d'instruction est mis à jour avec le numéro d'instruction vers lequel le saut doit être effectué et tous les composants précédemment inhibés sont

débloqués. La nouvelle instruction à exécuter écrase donc l'instruction de saut à l'entrée de l'étage LiDi. Si l'on ne saute finalement pas, l'instruction qui suit immédiatement l'instruction de saut dans la mémoire d'instructions écrase cette dernière à l'entrée de l'étage LiDi.

f) Problèmes identifiés, solutions adoptées et limites

Lors de l'implémentation de notre microprocesseur, nous avons a été confrontée à plusieurs problèmes et difficultés, pour lesquels nous avons finalement pu trouver des solutions appropriées :

- Les opérations logiques n'étant pas prises en charge dans notre implémentation du système, nous avons eu une approche simplifiée des tests pour l'instructions JMF en évaluant la valeur d'un registre donné, et en considérant que la condition était fausse s'il était égal à zéro.
- La gestion du banc de registres posait des difficultés pour mettre à jour les signaux QA et QB à chaque changement de valeur de leurs signaux d'entrée respectifs. En effet, nous nous sommes rendu compte tard qu'un retard indésirable avait lieu en sortie de ces signaux. Nous avons donc déplacé l'affectation de ces signaux à l'extérieur du process afin de régler ce problème.
- La mise à jour du compteur pour les sauts était complexe. Pour la gérer, nous avons ajouté une entrée IP_JUMP au compteur pour lui transmettre l'adresse à laquelle sauter, et nous avons adapté la sortie IP_OUT en fonction de la détection d'un saut à l'étage MemRe.
- L'ordre des arguments pour l'instruction JMF différait des autres opérations : l'adresse dans laquelle faire une lecture (pour l'évaluation de la condition) se trouvait en A et non en B comme à l'accoutumé. Nous avons pour régler ce problème rajouté deux multiplexeurs et ainsi circuit logique pour récupérer la valeur de condition ainsi que l'adresse de saut appropriées dans le cas d'un JMF.

III Cross-assembleur

Afin de réaliser la traduction du jeu d'instructions orienté mémoire généré par le compilateur, au jeu d'instructions orienté registre manipulé par le processeur, notre système complet nécessite l'implémentation d'une troisième entité principale : le cross-compilateur, ou cross-assembleur.

Pour mettre en place cette transposition, notre cross-assembleur récupère le fichier généré par le compilateur et obtient ainsi un tableau d'instructions, qu'il examine ligne par ligne, en adaptant les instructions gérées par le processeur :

- Pour l'AFC, la valeur indiquée est d'abord stockée dans un registre (AFC), que l'on store (STORE) ensuite à l'adresse mémoire spécifiée ;
- Pour le COPY, nous stockons le contenu de la deuxième adresse dans un registre grâce à un LOAD, avant de stocker le contenu du registre à la première adresse spécifiée dans l'instruction initiale grâce à un STORE ;
- Pour les instructions correspondant aux opérations arithmétiques réalisables par l'ALU, deux LOAD sont générés pour stocker les valeurs des deux adresses (B et C) indiquées dans des registres. L'opération est ensuite effectuée entre les valeurs de ces deux registres, et la valeur du registre dans lequel le résultat est inscrit est ensuite stockée à l'adresse résultat initialement spécifiée grâce à un STORE.

Le cross-assembleur ayant été implémenté avant l'ajout de fonctionnalités supplémentaires au processeur, nous n'avons malheureusement pas pu l'adapter pour traiter la traduction des nouvelles instructions JMP et JMF.

IV Résultats

La durée de traversée du chemin critique avant l'implémentation des sauts, qui peuvent ralentir davantage le parcours, était d'environ 6ns. La fréquence maximale d'utilisation de notre microprocesseur est ainsi d'environ 166MHz avant l'implémentation des instructions de saut et de la gestion des aléas de branchement.

Le processus critique de notre processeur, celui qui prend le plus de temps, est le chemin de retour entre le niveau DiEx et Lidi créé par l'implémentation des sauts. En fait, notre configuration peut créer une dépendance de données si une instruction dépend des résultats d'une instruction précédente pour pouvoir être exécutée correctement. Pour améliorer cette situation, nous pourrions envisager de mettre en place des techniques de cache et des techniques de prédiction afin de placer les données les plus susceptibles d'être accédées dans des niveaux de cache supérieurs, plus rapidement accessibles.

De plus, notre processeur n'a que cinq niveaux de pipeline. Nous pourrions donc augmenter sa fréquence maximale en divisant les tâches les plus lourdes en créant de nouveaux niveaux de pipeline, tout en limitant l'impact négatif des sauts engendrés par les nouveaux étages de pipeline. Également, avant l'implémentation des sauts, l'étage ralentissant le plus notre processeur était l'ALU. Pour améliorer le temps de propagation du niveau DiEx du pipeline correspondant à l'ALU, nous pourrions diviser les tâches de l'ALU en sous-tâches plus petites et plus rapides, en utilisant le pipelining de l'ALU et en utilisant des circuits logiques spécialisés pour des opérations courantes.

Finalement, afin d'optimiser notre implémentation des sauts conditionnels pour limiter son impact sur la fréquence maximale, nous pourrions évaluer directement évaluer leur condition en entrée de l'étage DiEx et donc en sortie du banc de registre, pour limiter le nombre de NOP propagés et débloquer les composants inhibés si le saut n'a pas à être effectués.

Conclusion

Ce projet nous a offert l'occasion d'approfondir notre compréhension du fonctionnement interne d'un processeur et d'un compilateur. À travers les différentes étapes de conception, de l'implémentation des instructions à la gestion des aléas, en passant par la création d'une table des symboles et la manipulation des variables temporaires, nous avons acquis une vision plus claire et approfondie de ces aspects essentiels de l'informatique. Ce projet nous a également permis d'explorer les défis pratiques liés à la mise en œuvre d'un compilateur et d'un processeur, tels que la résolution des conflits et des problèmes de gestion de la mémoire. Ainsi, cette expérience nous a dotés d'une solide base de connaissances et de compétences pratiques dans la conception et l'implémentation d'un système informatique.

Annexe

Instruction	Opcode	Utilisation
ADD	1	01 @résulat @opérande1 @opérande2 (ex. 01 00 01 02)
MUL	2	02 @résulat @opérande1 @opérande2
SOU	3	03 @résulat @opérande1 @opérande2
DIV	4	04 @résulat @opérande1 @opérande2
COP (ou COPY)	5	05 @résulat @opérande
AFC	6	06 @résulat valeur (en “brut”)
JMP (ou JUMP)	7	07 num (où num est le numéro d’instruction à laquelle sauter)
JMF	8	08 @cond num (@cond adresse de la condition à évaluer)
INF	9	09 @résulat @opérande1 @opérande2
SUP	10	0a @résulat @opérande1 @opérande2
EQ	11	0b @résulat @opérande1 @opérande2
PRI (ou print)	12	0c @var (@var adresse de la variable à afficher)
LOAD	13	0d @registre @memoire
STORE	14	0e @memoire @registre
GEQ (>=)	15	0f @résulat @opérande1 @opérande2
SEQ (<=)	16	10 @résulat @opérande1 @opérande2
RET	17	11
PUSH	18	12 nb (où nb correspond au décalage du pointeur de pile)
CALL	19	13 num (où num est le numéro d’instruction à laquelle sauter)
POP	20	14 nb (où nb correspond au décalage du pointeur de pile)
NEQ (!=)	21	15 @résulat @opérande1 @opérande2

Tableau des correspondances entre les instructions et les opcodes associés