INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet Système Informatique

Rapport

Etudiants :
Adama SALL
Ghizlane BADAOUI

Encadrants :
Eric Alata
Daniela Dragomirescu

4A Informatique et Réseaux

31 mai 2023

Table des matières

1 Introduction **2**

2 Partie compilateur **2**
2.1 Analyse lexicale 2
2.2 Analyse syntaxique 3
2.3 Table des symboles 3
2.4 Compilation et traduction vers le code assembleur 4
2.5 Interpréteur 5

3 Cross compilateur **5**

4 Partie processeur **7**
4.1 Pipeline et chemin de données 7
4.2 Gestion des aléas de données et de branchement 8
4.2.1 Aléas de données 8
4.2.2 Aléas de branchement 9

5 Conclusion **9**

1 Introduction

Le présent rapport présente les résultats et les principales étapes du projet de développement d'un compilateur et d'un microprocesseur de type RISC avec pipeline. Le projet s'est déroulé en deux grandes parties distinctes, à savoir la conception du compilateur et la conception du processeur. Il y'a aussi eu une partie intermédiaire entre les deux pour développer le cross compilateur. Chacune de ces parties a nécessité une organisation spécifique pour atteindre les objectifs fixés.

La première grande partie du projet était axée sur la conception du compilateur. Les différentes étapes de cette partie comprenaient l'analyse lexicale, l'analyse grammaticale, la création et la gestion de la table des symboles et la compilation proprement dite qui a été effectuée pour générer le code assembleur correspondant au programme source.
Cependant, ce dernier était seulement orienté mémoire. Voilà pourquoi il y'a eu une partie intermédiaire de developpement du cross compilateur qui a pour but de rendre ce code orienté mémoire en code orienté registre et mémoire.
La seconde grande partie du projet portait sur la conception d'un microprocesseur de type RISC avec pipeline. Son architecture a nécessité la mise en place de différents composants, notamment l'Unité Arithmétique et Logique (UAL), le banc de registres(BR), la mémoire d'instruction(MI) et la mémoire de données(MD). Pour se faire, des étages de pipeline ont été intégrés et des chemins de données ont été constituées pour chaque instruction. De plus, une gestion des aléas de données ainsi que des aléas de branchement a été mise en place pour gérer les dépendances entre ces instructions et éviter des incohérences lors de l'exécution.

Les prochaines sections de ce rapport détailleront chacune de ces parties, mettant en évidence les choix techniques, les défis rencontrés et les résultats obtenus.

2 Partie compilateur

2.1 Analyse lexicale

La première partie du projet concerne l'analyse lexicale du langage C, elle consiste à scanner le programme source afin d'identifier les éléments essentiels tels que les mots-clés, les identificateurs, les nombres et les opérateurs.

Dans notre projet, on a utilisé l'outil LEX pour mettre en place l'analyse lexicale. Et avec ceci, les mots-clés tels que "if", "else", "while", "print", "return", "int" et "void" sont détectés et renvoyés comme des tokens spécifiques. De même, les identificateurs, ainsi que les nombres entiers, sont reconnus et renvoyés avec leur type respectif. En effet, pour les identifiacteurs, on récupére leurs noms et pour les entiers leurs valeurs.

Concernant les commentaires, on a pris en compte les deux types : les commentaires de ligne commençant par "//" et les commentaires multilignes entourés de "/*" et "*/". Et on les a traité comme des éléments non significatifs lors de l'analyse lexicale. Ils ont donc été ignorés tout comme les espaces, les tabulations "\t" et les sauts de ligne "\r \n". Ainsi, ils n'auront aucun impact sur l'identification des tokens et des structures syntaxiques du

code.

2.2 Analyse syntaxique

La grammaire définie dans notre projet permet de gérer les déclarations des variables, les expressions arithmétiques, les instructions conditionnelles (if-else, while), les instructions de retour (return) et les instructions d'affichage (print), ainsi que **les définitions et les appels de fonctions**. Un aspect important de notre grammaire est sa capacité à **garder la priorité des opérateurs arithmétiques sans parenthèses**, assurant ainsi une évaluation correcte des expressions mathématiques.

2.3 Table des symboles

Pour implémenter une table de symboles, on a opté pour l'utilisation de structures. Chaque ligne de la table de symboles est représentée par une instance de la structure appelée ***ligne_tableau***. Cette structure est définie comme suit :

```
1  typedef struct {
2      char nom[8];
3      char init;
4      int profondeur;
5  } ligne_tableau;
```

- ***nom*** est un tableau de caractères qui représente le nom du symbole. Il est vide au cas d'une variables temporaire.
- ***init*** est un caractère qui indique si le symbole a été initialisé (***init=1***) ou non (***init=0***). Il est utilisé pour vérifier si une variable a été assignée avant son utilisation.
- ***profondeur*** est un entier qui représente le niveau de profondeur auquel le symbole est déclaré. Il est utile pour gérer la portée des variables dans le programme.

Cette approche, basée sur des structures, permet de simplifier la manipulation de la table des symboles et en même temps de minimiser la taille de celle-ci en stockant uniquement les informations nécessaires pour chaque symbole, sans gaspiller d'espace mémoire supplémentaire.

En ce qui concerne les principales fonctions utilisées pour gérer la table des symboles, il y'a :

- La fonction ***addVar*** qui ajoute une variable à la table des symboles si elle n'existe pas déjà.
- La fonction ***suppVarErrone*** qui supprime les variables locales erronées (qui n'ont plus de portée) de la table des symboles à chaque diminution de la profondeur. Ces derniéres sont décelées en comparant leur profondeur avec la profondeur à laquelle le programme se situe. Ainsi, si leur profondeur est supérieure à la profondeur du programme, elles sont supprimées.
- Les fonctions ***ts_new_tmp***, ***ts_free_last***, ***ts_get_last*** et ***ts_get_next_last*** qui sont utilisées pour gérer les variables temporaires dans la table des symboles.

- La fonction ***ts_ get_ addr*** qui retourne l'indice de la variable dont le nom est donné en paramétre dans la table des symboles.
- La fonction ***get_ assignedVar*** qui renvoie le nom de la dernière variable assignée.
- La fonction ***updateVar*** qui permet de mettre la valeur "init" d'un symbole à 1 pour signifier que celui-ci est initialisé.
- La fonction ***update_ assignedVar*** permet de sauvegarder le nom de la variable à laquelle on doit assigner le resultat d'un calcul. Par exemple, Si on a "int a=2+3", c'est la fonction *update_ assignedVar* qui nous permet de garder à l'idée que le résultat final de ce calcul doit être mis à l'adresse de *a* dans la table des symboles. ***get_ assignedVar*** renvoie le nom de la variable au moment d'assigner le résultat.
- La fonction ***update_ tailleActuelle*** met à jour la taille de la table des symboles.

2.4 Compilation et traduction vers le code assembleur

Les instructions assembleur générées comprennent :

— Les opérations de manipulation de variables (COP)

— L'affectation de valeurs constantes (AFC)

— Les sauts conditionnels (JMPF)

— Les sauts inconditionnels (JMP)

— Les opérateurs de comparaison (INF, SUP, EQU). Afin de différencier tous les cas de comparaisons, on a rajouté aussi les instructions INFE (pour <=), SUPE (>=) et NEQU (!=).

— Les instructions arithmétiques de base (ADD, SUB, MUL, DIV)

— Les instructions de gestion de fonctions telles que l'appel de fonction (CALL) et le retour de fonction (RET).

— Les instructions de lecture et d'écriture (LOAD et STR).

— L'instruction d'affichage (PRI).

Pour représenter chacune de ces instructions, on a opté pour l'utilisation de structures. Ainsi, chaque instruction de notre code assembleur est représentée sous forme d'une structure ***instruction*** qui est définie comme suit :

```
1  typedef struct {
2      char nom[8];
3      long op1;
4      long op2;
5      long op3;
6  } instruction;
```

- ***nom*** est un tableau de caractére qui représente le type de l'instruction. Cela peut être ADD, SUB, MUL, AFC, COP, etc.
- ***op1***, ***op2*** et ***op3*** représentent respectivement le premier, le deuxième et le troisième opcode de l'instruction. Et si une instruction n'a pas certains opcodes, on met les opcodes qu'elle n'a pas à -1.

Ces instructions sont ensuites stockées dans une liste de structure instruction qu'on a nommé "assembleur". Et les différentes fonctions permettant de gérer cette dernière sont :

- La fonction ***add_instruction*** qui ajoute une instruction à la table d'assembleur, en mettant les opcodes vides à -1.
- La fonction ***get_nbre_lignes_asm*** qui renvoie le nombre d'instructions stockées dans la table.
- La fonction ***display_asm*** qui affiche toutes les instructions stockées avec leurs opcodes s'ils existent.
- La fonction ***updateAsm*** qui modifie l'opérande 1 (en cas de JMP) ou 2 (en cas de JMPF) d'une instruction à l'indice spécifié en paramétre. Ceci permet de mettre à jour la ligne à laquelle on doit sauter en cas de JMP ou de JMF.

En générant les codes assembleurs correspondant à la définition et à l'appel de fonction, on s'est rendu compte qu'il nous fallait stocker pour chaque fonction définie son nom pour l'identifier, son nombre de paramétres ainsi que son adresse de retour. Ainsi, on a créé une structure ***détail_fonction*** qu'on a défini comme suit :

```
1    typedef struct {
2        char nom[8];
3        int addrFonction;
4        int nbre_params;
5    } detail_fonction;
```

Et pour gérer l'ensemble des fonctions rencontrées, les fonctions ***addDetailFonction***, ***getAddrFonction***, ***get_nbre_params*** et ***update_params_detailFonction*** sont utilisées pour gérer les détails des fonctions dans le code assembleur.

2.5 Interpréteur

Pour visualiser le contenu de la mémoire après avoir générer le code assembleur orienté mémoire, on a développé en Python un interpéteur nommé ***interpreteur.py***. Celui-ci parcourt les instructions une par une et les exécute en maintenant une mémoire pour stocker les valeurs des registres et des adresses. A la fin de l'exécution, on affiche à l'écran le contenu de la mémoire sous format d'un tableau, pour vérifier le comportement de notre programme.

3 Cross compilateur

Aprés avoir procédé à la compilation pour générer le code assembleur orienté mémoire correspondant au programme source, on a mis en place le cross compilateur pour transformer ce code orienté mémoire en code orienté registre et mémoire.

Cette étape intermédiaire a été effectuée en python. Notre code prend en entrée les instructions assembleur résultant de la première partie du projet(celles orientées mémoire) et l'adapte avant qu'il n'aille au processeur.

Le cross-compilateur fonctionne en traitant chaque ligne d'instruction individuellement et en générant le code hexadécimal correspondant. Et à travers une série de conditions, on identifie l'opération spécifiée dans chaque ligne générée par le code YACC. Ensuite, on

calcule les valeurs hexadécimales des paramètres et les utilise pour construire les instructions finales en code machine.

La tranformation du code assembleur a été fait ainsi :

— "ADD A B C" → "ADD RA RB RC". On a fait le choix de considérer directement que A B C sont des numéros de registres. Ainsi, de même, "MUL A B C" → "MUL RA RB RC". De même, "SUB A B C" → "SUB RA RB RC".

— Pour les AFC, puisqu'à chaque fois elles sont effectuées vers un emplacement temporaire et qu'un emplacement temporaire (dans le compilateur) correspond à un registre (dans le processeur), on a "AFC A B" → "AFC RA B".

— Pour "COP A B", il y a 2 transformations possibles. Car dans notre compilation, soit le COP s'effectue d'un emplacement temporaire vers un "vrai" emplacement de la mémoire (1er cas), soit dans le sens inverse c'est-à-dire d'un "vrai" emplacement à un emplacement temporaire (2ème cas) . Le 1er cas est détecté lorsque l'on a $B > A$ puisque l'adresse des emplacements temporaires à un instant donné est toujours supérieur à l'adresse d'un vrai emplacement. Et lorsqu'on est dans ce cas, "COP A B" → "STORE @A RB". Pour le 2éme cas, celui-ci est détecté lorque $A > B$ pour les mêmes raisons citées ci-dessus. Et lorsqu'on est dans ce cas, "COP A B" → "LOAD RA @B".

Pour le codage en hexadecimal de chaque opération, on s'est referé au tableau ci-dessous :

TABLE 1 – *Contenu du OP pour chaque instruction assembleur et les accès lecture/écriture dans le banc de registres*

opération	**OP**	**instruction asm**	**Read/Write**
pas d'opération	0x00	NOPE	-
addition	0x01	ADD	R/W
multiplication	0x02	MUL	R/W
soustraction	0x03	SUB	R/W
copie	0x05	COP	R/W
affectation	0x06	AFC	W
chargement	0x07	LOAD	W
sauvegarde	0x08	STORE	R
inférieur	0x09	INF	R/W
inférieur ou égal	0x19	INFE	R/W
supérieur	0x0A	SUP	R/W
supérieur ou égal	0x1A	SUPE	R/W
égal	0x0B	EQU	R/W
différent	0x2B	NEQU	R/W
affichage	0x0C	PRI	R
saut inconditionnel	0x0D	JMP	-
saut conditionnel	0x0E	JMPF	R

L'instruction NOPE signifie *pas d'opération* et est utilisée pour bloquer les étages du pipeline au cas de détéction des aléas de données et de branchement (cf. 4.2).

Une fois l'exécution du cross-compilateur terminée, nous obtenons une liste complète des instructions générées, prêtes à être exécutées sur la carte FPGA.

4 Partie processeur

4.1 Pipeline et chemin de données

Pour le pipeline, on a 4 étages qui sont : LI_DI, DI_EX, EX_MEM, MEM_RE. Pour représenter ces derniers, sachant que les 2 premiers étages (LI_DI et DI_EX) ont 4 paramètres chacun et différent seulement en fonction de ceux-ci, on a créé un composant nommé "LI_DI" pour les 2. Ce composant est synchrone et à chaque clock d'horloge, il sort ce qu'on lui a donné en entrée. Ceci, si le *enable* n'est pas activé (EN='0'). On verra pourquoi cela dans la partie de gestion des aléas. Pour les 2 derniers étages (EX_MEM et MEM_RE), on a défini un autre composant nommé "EX_MEM" pour présenter les deux autres étages restants du pipeline car ils ont tous les deux, 3 paramétres mais qui différent par contre. Celui-ci est aussi synchrone et à chaque clock d'horloge, sort ce qu'on lui a donné en entrée.

Concernant le chemin de données, il marche commme suit :
D'abord il y a un pointeur d'instructions (IP) qui permet de sortir de la mémoire d'instructions, instruction par instruction et qui est géré par un compteur 8 bit. En effet, IP est presque toujours égal au signal de sortie généré par ce compteur. Au début, supposons que IP (pointeur d'instruction) est égal à 0.

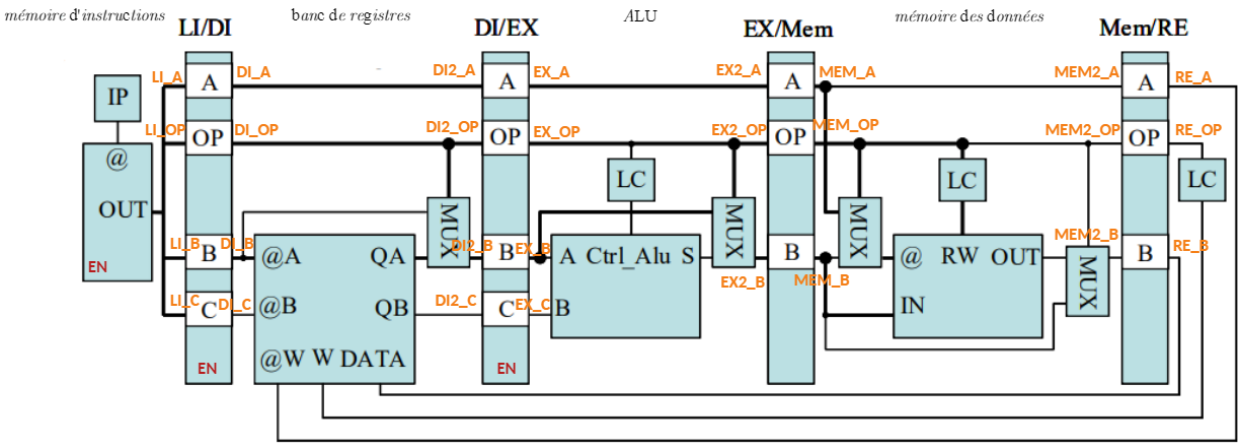

FIGURE 1 – *Chemin de données avec les entrées et sorties des étages du pipeline*

— Donc au **1er clock**, l'instruction N°0 sort de la mémoire d'instructions et se positionne en entrée du 1er étage du pipeline(LI_A, LI_OP, LI_B, LI_C).

— Au **2ème clock**, l'instruction N°0 sort du 1er étage (on a donc DI_A, , DI_OP, DI_B, DI_C) et se positionne en même temps à l'entrée du 2ème étage (on a donc DI2_A, DI2_OP, DI2_B, DI2_C). Et chacune de ces entrées du 2éme étage (les DI2) est soit directement égale à la sortie du 1er étage soit à la sortie du banc de registre après lecture. Ceci diffère en fonction de l'instruction (Voir code pour plus de détail).

— Au **3ème clock**, l'étage 2 sort les DI2 fournies en entrée. Ainsi, on a donc EX_A, EX_B, EX_C, EX_OP. En même temps, les entrées du 3ème étage (EX2_A, EX2_B, EX2_OP) sont instanciées. Ici aussi, pas tous les EX2 mais EX2_B est soit directement égale à la sortie du 2ème étage EX_B, soit à la sortie de l'ALU après calcul. On est dans ce dernier cas lorqu'on a soit des opérations arithmétiques à effectuer (ADD, MUL, SUB, DIV) soit des opérateurs de comparaison (EQU, NEQU, INF, INFE, SUP, SUPE).

— Au **4ème clock**, on a les sorties de l'étage "EX_MEM" (MEM_A, MEM_B,MEM_OP) et en même temps les entrées du dernier étage (MEM2_A, MEM2_B, MEM2_OP) qui sont instanciées. Ici aussi, chacune de ces entrées est soit directement égale à la sortie du 3ème étage, soit à la sortie de la mémoire de données après lecture lors d'un LOAD.

— Enfin, au **5ème clock**, le dernier étage fait sortir les instructions. RE_A, RE_B, RE_OP sont ainsi instanciés. Et en même temps à la descente de ce clock, on écrit, s'il y'a lieu, dans le banc de regitre.

Par contre, il y'a une gestion spéciale effectuée lorqu'on a un **JUMP**. En effet, à l'entrée du 1er étage, on vérifie toujours si LI_OP="0x0E"(JUMP). Si tel est le cas, on met le load du compteur à '1' tout en mettant le Din du compteur à LI_A. Ainsi, IP prend la valeur de LI_A et le saut sera donc effectué. Dans le cas d'un JUMPF, puisqu'on doit d'abord lire dans le banc de registre, on ne pourra donc détecter qu'il y'a un JUMP à faire qu'à la sortie du banc de registre où on lit la valeur de DI_A. Et si cette sortie équivaut à "0", cela veut dire que l'on doit sauter. Et dans ce cas, on modifie directement l'entrée du 1er étage (les LI) afin de mettre l'instruction "JUMP DI2_B" en entrée de celui-ci.

4.2 Gestion des aléas de données et de branchement

4.2.1 Aléas de données

Il arrivait qu'on lise une valeur d'un registre avant qu'une instruction qui devait modifier la valeur de ce registre ne puisse arriver au bout. C'est ce que l'on appelle un aléa de données. Pour résoudre celà, on a créé un composant asynchrone "aléa" qui prend en entrée l'opérateur en entrée des 4 étages, le paramètre A du 2ème, 3ème et 4ème étage et les 2 paramètres B et C du 1er étage, et renvoie en sortie OUT_alea activé à '0' si un aléa est détécté ou à '1' sinon.

Ce composant nous permet de détecter à chaque changement de valeur dans les entrées des étages s'il y a un aléa ou pas. Dans le cas d'un aléa, le signal "OUT_Alea='0'". Et on fait en sorte que le *enable* du compteur (EN) ainsi que le *enable* du 1er étage (EN_LI_DI) soit activé lorsque ce OUT_Alea='0'. Ainsi si l'enable du compteur vaut '1', le compteur se décrémente d'abord une fois pour ne pas prendre l'instruction qui suit et reste ensuite constant afin que les entrées du 1er étage restent figer tant qu'il y a un aléa. Pour le 1er étage, si son enable (EN_LI_DI) vaut '1' alors il ne prend pas en compte ce qu'on lui a donné en entrée et ne sort que des "0000 0000" qui équivaut à l'instruction NOPE. Ainsi, la gestion des aléas de données a été réalisée.

4.2.2 Aléas de branchement

Pour les aléas de branchement qui ont lieu lorque l'on doit se fier à la valeur contenue dans un registre pour sauter ou pas, et que en même temps il y'a une instruction en cours qui modifie la valeur du registre à lire.

Pour résoudre cet aléa, on a procédé presque de la même manière que les aléas de données en ajoutant JUMPF comme une opération de lecture. La différence est que lorsqu'on a un JUMPF qui suit une opération d'écriture dans le même registre, le registre de lecture du JMPF va être ici sur l'opérande A et non pas sur les opérandes B ou C comme les aléas de données. On a donc mis lorsqu'on a un JUMPF le paramétre B_LI_DI à LI_A.

5 Conclusion

En conclusion, ce projet de développement d'un compilateur a été une expérience enrichissante qui nous a permis d'explorer en profondeur les concepts fondamentaux de la compilation. À travers la conception de la grammaire, la génération des instructions assembleur et la conception d'un processeur pour executer ces dernières, nous avons acquis une compréhension approfondie du processus de transformation d'un langage de programmation en un code exécutable ainsi que son exécution par un processeur.