INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet Systèmes Informatiques
Du compilateur vers le microprocesseur

Arrondelle Hugo – Auban Marc – Fragonas Fabrice 4IR C- 2020/2021

Table des matières

Introduction

Dans ce rapport, nous allons vous expliquer la conception que nous avons décidé de mettre en place pour mener à bien les deux grandes parties de ce projet. A savoir, le compilateur C dans un premier temps, puis le processeur dans un deuxième temps. Le rapport sera rédigé dans le même ordre de conception que nous avons dû réaliser en cours.

Tout au long du rapport, nous nous appuierons sur des morceaux de code afin d'être le plus explicite et clair possible dans nos explications.

Dans la première partie du compilateur C sera évoqué la conception de l'analyse syntaxique en LEX, l'analyse sémantique en YACC, et la génération de la partie assembleur comme sous partie. Enfin, le rapport se terminera par la partie Processeur avec la création des différents modules en VHDL qui le composeront.

Compilateur

I. Lex

Nous avons commencé le projet par l'implémentation du Lex de notre compilateur. Le Lex est un outil qui permet de générer un analyseur lexicographique à partir de de la description de la lexicographie du langage. Pour cela nous avons ajouté des token relatifs aux fonctionnalités du langage C. Les token nous permette de reconnaitre des termes du langages : Le token « T_equal » reconnaitra tous les « = » dans un code en C.

```
"="                                 { return T_equal; }
```

De plus nous avons ajouté plusieurs expressions régulières afin de reconnaitre différents **paramètres définis au préalable**. Ainsi lorsque le Lex reconnaitra un « espace », un « Retour a ligne » ils seront ignorés par le système.

```
SPACE                               [ \t]+
NEWLINE                             [\n]+
BLANKS                              (({SPACE})|({NEWLINE}))+
```

II. YACC

Dans un second temps de notre projet nous avons réalisé le Yacc de notre projet. Le Yacc est un outil qui génère l'analyseur syntaxique à partir de la description de la syntaxe. Tout d'abord nous avons décrit les principale règles permettant de reconnaitre la syntaxique des fonctionnalités du langage C. Nous avons commencé par reconnaitre les affectations, les opérations arithmétiques pour vérifier que les Token de notre Lex était bien reconnue.
Par la suite nous avons ajouté des fonctionnalités nécessaires au bon fonctionnement de notre compilateur. Tout d'abord nous avons géré la priorité sur les Token afin que toutes les opérations arithmétiques soit correcte (« multiplication est plus prioritaire que l'addition »).

```
%left T_equal
%left T_less_than T_less_than_or_equal_to T_greater_than T_greater_than_or_equal_to T_different T_equal_comparison
%left T_add T_sub
%left T_mul T_div T_mod
%left T_po T_pf
```

Dans un premier temps nous avons dû effectuer nos propres tests pour vérifier le bon fonctionnement de l'analyse syntaxique et sémantique par l'intermédiaire de « printf ». Cela nous a permis de voir rapidement si notre compilateur reconnaissait le code C que nous lui envoyons. Celui-ci devait nous retourner les bons Token.

Par la suite, dans la continuité du projet, nous avons dû remplacer les « printf » par des fonctions qui génère les instruction ASM du langage C. Les fonctions et la manière dont nous avons décidé d'implémenter la partie ASM sera expliquée en profondeur dans la suite de ce rapport.

III. ASM

Table de Symbole

Dans notre projet nous avons utilisé une table de symbole sous forme de tableau et une table de symbole temporaire également sous forme de tableau. La table de symbole nous permet d'enregistrer toutes les variables déjà utiliser en ASM. Lorsque qu'une nouvelle affectation de variable parvient nous vérifions que cette variable n'a pas déjà était affecter dans un registre. Si cela est le cas nous mettons à jour la table des symboles en affectant la valeur à l'adresse du registre de la variable dans la table des symboles. La table des symboles temporaire nous permet de mettre des données utiles dans un temps limité.

Gestion erreur si une variable non déclarée

Déclarations de variables

Une déclaration de variable peut se faire de plusieurs façons. Une déclaration de variable simple ou une déclaration de variable avec affectation de valeur. Nous sauvegardons la variable déclaré dans la table des symboles. Pour affecter (u = 9) nous affections 9 dans la table des symboles temporaire puis nous copions la valeur de l'adresse 100 (temporaire) dans l'adresse 4 de la table des symboles.

```
; declaration and initialization of variable 'u'
AFC 100 9
COP 4 100

AFFICHAGE SYMBOL TABLE

_____________________________________
| Index | Adress  | Depth | Identifier |
|   0   |   0     |   1   |     e      |
|   1   |   4     |   1   |     u      |
_____________________________________
```

Fonction Printf()

La fonction printf(Variable) es une fonction simplifié de notre langage. Afin de faire un printf(a) nous copions la valeur dans le dans la table des symboles temporaire puis nous appelons l'instruction ***PRI*** de l'adresse temporaire.

Expressions Arithmétique

Dans cette partie de projet nous avons pu constater que les expressions arithmétiques pouvaient être différente. Tout d'abord nous avons les expressions de type Expression opération Expression (1) de plus il existe les expressions avec une certaine priorité comme par exemple les parenthèses ou multiplication (2).

(1) Notre programme va donc charger les deux éléments au sommet de la pile, faire le calcul et stocker le résultat dans la première expression.
(2) Notre programme va donc gérer les priorités et se référer à l'expression (1) pour la suite.

```
; declaration and initialization of variable 'a'
AFC 100 2
AFC 104 1
ADD 100 100 104
AFC 104 3
MUL 100 100 104
COP 12 100
```

```
int a = (2 + 1) * 3;
```

Tout d'abord nous affectons la valeur 2 dans la table temporaire (@ 100) puis nous affectons 1 dans la table temporaire (@ 104) ensuite nous faisons l'addition des deux variables (@ 100). Pour finir nous affectons 3 dans une adresse temporaire (@ 104) et puis nous faisons la multiplication de l'adresse 100 et 104.

Condition (if-else)

Tout d'abord pour gérer les différents sauts conditionnels nous avons décidé de sauvegarder le numéro de ligne afin de pouvoir faire un saut conditionnel (JMF). Si la condition du **if** n'est pas valide alors il faut faire un saut conditionnel afin de passer au **else**. Si la condition du **if** est valide alors on réalise les instructions du if pour Jump le **else** (JMP).
Afin de gérer le saut conditionnel nous avons décidé de créer une fonction afin de savoir si la condition du **if** est valide :

- A - On regarde d'abord si la condition du if est true (1) ou false(0)
- B - Nous allons comparer ensuite avec 0 la valeur de (A) . True(1) or False(0)
- C - On affecte la valeur 1
- D - On soustrait C – B pour ré inverser le chiffre afin de savoir si la condition du if est True ou False

```c
void main() {

    int a = (2 + 1) * 3;
    int b;
    if (a == 3) {
        b = 9;
    }else{
        b = 8;
    }
    printf(b);

}
```

```
; declaration and initialization of variable 'a'
AFC 100 2
AFC 104 1
ADD 100 100 104
AFC 104 3
MUL 100 100 104
COP 0 100
; if statement avec else
COP 100 0
AFC 104 3
EQU 100 100 104
; Translate expression to condition
AFC 104 0
EQU 100 100 104
AFC 104 1
SOU 100 104 100
; if block {
JMF 100 17
; assignment of variable 'b'
AFC 100 9
COP 4 100
JMP 19
; else statement
; assignment of variable 'b'
AFC 100 8
COP 4 100
; } end of if-else statement
```

Exemple :
A = 9
@A = 0

COP 100 0 //On copie @A (0) dans @100
AFC 104 1 //On affecte 1 à @104
EQU 100 100 104 // Résulat false (0) // Value @100 ≠ @104

AFC 104 0 // On affecte 0 dans @104
EQU 100 100 104 // Résultat true (1)) // Value @100 = @104
AFC 104 1 // On affecte 1 dans @104
SOU 100 104 100 // 1-1 = 0 // On inverse le 1 pour JMF
JMF 100 17 // Valeur @100 = 0 // False donc saut conditionnel

La condition du if n'est pas respecté donc nous devons passer à la ligne 17 (JMF 100 17)

Condition (While)

Le principe du while reste dans le même principe que la **condition du if.** Tout d'abords nous vérifions la condition du while pour savoir si la condition est valide.

- Si la condition n'est pas valide nous « sautons la partie des instructions du while JMF ».
- Si la condition est valide nous réalisons les instructions du while puis nous allons jumps JMP au début du while pour vérifier si la condition est toujours valide pour faire en boucle les instructions.

II. Processeur

Dans cette dernière partie concernant la réalisation des différents composants pour le processeur, nous allons vous exposer les problèmes que nous avons pu rencontrer et comment nous avons réussi à en comprendre les subtilités afin de les corrigés.

Nous avons pu implémenter la totalité des composants demandés à savoir : ALU, Banc de registre avec Bypass, Banc de mémoire (Mémoire des données et Mémoires des instructions), Chemin des données (Pipeline), MUX (Multiplexeur) et LC (Logique Combinatoire).

Sur l'ALU, nous avons implémenter simplement que les opérations de base (Addition, Soustraction et Multiplication) et les flags de détection d'Overflow, de Carry, et Négative.

Sur le Registre, il est aussi demandé d'implémenter un ByPass pour éviter la lecture et l'écriture en même temps. Ceci nous a posé des problèmes dans le démarrage du projet. Nous avons finalement réussi à comprendre ce qu'il fallait réaliser et somme arriver au code suivant :

```
QA <= DATA when (W='1' and addrA = addrW) else
    reg(to_integer(unsigned(addrA)));
QB <= DATA when (W='1' and addrB = addrW) else
    reg(to_integer(unsigned(addrB)));
```

Concernant le module Instruction_Memory, nous avions en tête de réaliser la lecture d'un fichier généré en hexa par notre compilateur puis de venir le lire par le processeur. Malheureusement, nous avons dû nous replier sur la méthode la plus simple, faute de réussite, à savoir d'envoyer des instructions en dur dans le code.

Après la réalisation des nombreux tests, tous les sous modules ont réussi leur test. Le test du processeur est aussi fonctionnel mais il y a des problèmes d'initialisation dont nous n'avons pas réussi à en trouver l'origine :