Alicia CALMET
Loïc ROBERT

Année 2020-2021
4 IR A2

COMPTE-RENDU
PROJET SYSTEME INFORMATIQUE

Lien github : https://github.com/RobertL31/C_compiler

Démarche de conception :

Tout au long de projet, nous avons suivi le schéma organisationnel proposé sur la page Moodle. De plus, afin de ne pas être en retard sur le planning des séances, nous avons travaillé en dehors des heures de cours. Ainsi, notre projet a abouti et nous avons pu implémenter de nouvelles fonctionnalités qui n'étaient pas attendu à notre compilateur et nous avons pu gérer les aléas dans la partie microprocesseur.

Partie Compilateur en utilisant LEX et YACC:

Analyse lexicale : l'analyseur lexical détecte les mots typés (tokens) spécifiques à notre langage. Ce fichier (compiler.l) est séparé en trois parties : une partie est consacrée aux définitions, une partie est consacrée aux règles et une dernière partie est dédiée au code. La partie LEX lit donc le fichier, reconnaît et décode les tokens définis et les renvoie à la partie YACC que nous allons aborder maintenant.

Analyse syntaxique : l'analyseur syntaxique analyse notre langage de type C et produit comme sortie le code assembleur correspondant. Le fichier (compiler.y) se structure de la même façon que la fichier compiler.l. Le seul changement est que dans YACC, la partie règles définit l'ensemble des règles qui définissent notre grammaire au lieu des expressions régulières dans LEX.
Pour gérer les variables, nous avons créé une table des symboles et des fonctions déclarer dans symbol.h qui permettent de les gérer. Un symbole a un nom, une profondeur (les variables globales sont à une profondeur de 0, celles déclarés dans la main à une profondeur 1, etc), et un type ().
Nous avons aussi implémenté une table des symboles temporaires qui fonctionne comme une pile pour les calculs.

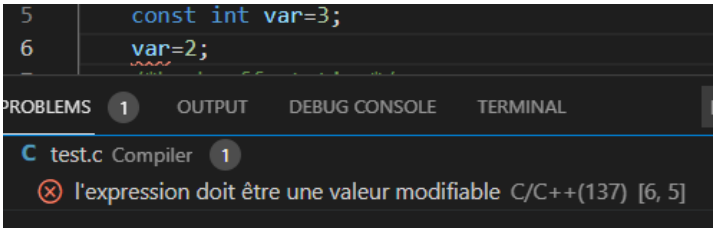
```
typedef struct s_symbol{
    char * name;
    int depth;
    TypeInfos typeInfos;
} Symbol;
```

Toutefois, au-delà de l'implémentation des expressions conditionnelles *if* et *while* demandée, nous avons implémenté de nouvelles fonctionnalités qui n'étaient pas attendu à notre compilateur.

En effet, nous avons implémenté :
- la possibilités d'écrire des commentaires (ils sont gérés de la même manière que les espaces)
- l'analyse d'une condition dans laquelle il y a une expression à droite et/ou à gauche

```c
/*boucle while*/
while( -(i + j) != (3*6) {
    /*code*/
}
```

- un message d'erreur lors de l'affectation d'une nouvelle valeur a une constante



- la possibilité de déclarer des variables sur une même ligne en les séparant par une virgule/ la possibilité d'affecter des valeurs à des variables sur une même ligne en les séparant par une virgule (des déclarations et affectations multiples ne marchent pas).

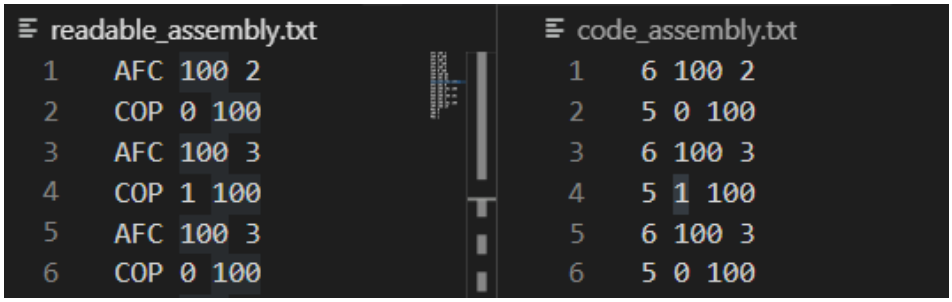
```c
/*déclaration des variables i et j*/
int i;
int j;

/*affectation des variables i et j*/
i = 2, j = 3, i = 3;
```

- le parsage du type d'une variable : nous avons réalisé du typage fort
- la gestion des opérateurs logiques ≥,<= et != en plus des opérateurs <,> et ==

```c
if (j<=4){
    /*code*/
}
```

- l'ébauches de pointeurs
-fprintf dans stderr pour warnings et erreurs
- la traduction avec instruction en code instruction



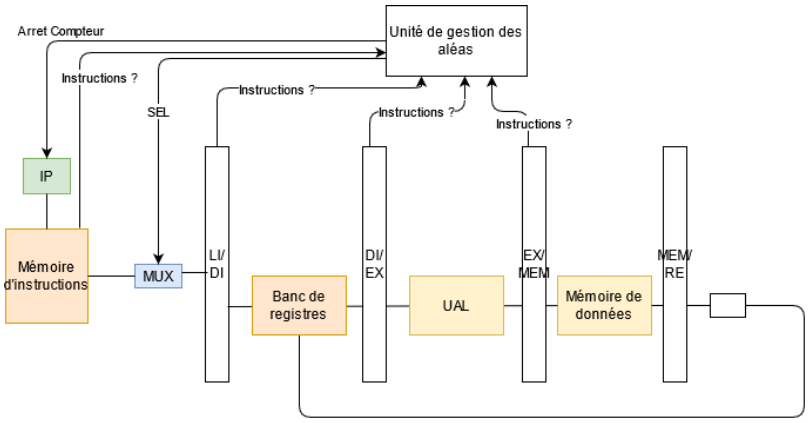
Il est à noter qu'il est facile de rajouter de nouvelles implémentations à notre code. Par exemple, nous pourrions rajouter le typage fort avec vérifications lors du parsage que tous les types sont compatibles entre eux.

Partie Microprocesseur de type RISC:

Dans cette partie, nous avons réalisé un microprocesseur de type RISC avec un pipe-line de 5 étages.

Chemin de données : Après avoir coder et tester l'ensemble des unités (unité arithmétique et logique, banc de registres à double port de lecture, mémoire d'instructions, mémoire de données), nous avons créé le chemin de données dans lequel tous les composants sont synchronisés sur front montant d'horloge excepté la mémoire de données qui fonctionne sur front descendant. Nous avons codé les instructions assembleur AFC, COP, ADD, MUL, SUB, DIV, LOAD et STORE.

Gestion des aléas : Ayant terminé plus tôt que prévu le chemin des données, nous avons pu gérer les aléas. Nous avons créé une unité de gestion des aléas qui prend en signaux d'input les instructions sortant de la mémoire d'instructions et des pipelines LI/DI, DI/EX et EX/MEM. S'il y a un aléa alors l'unité dit au compteur (IP) de s'arrêter et au multiplexeur de sortir l'instructions NOP. Cependant, notre unité de gestion des aléas fonctionne de manière asynchrone et ne devrait pas marcher sur un microprocesseur réel.



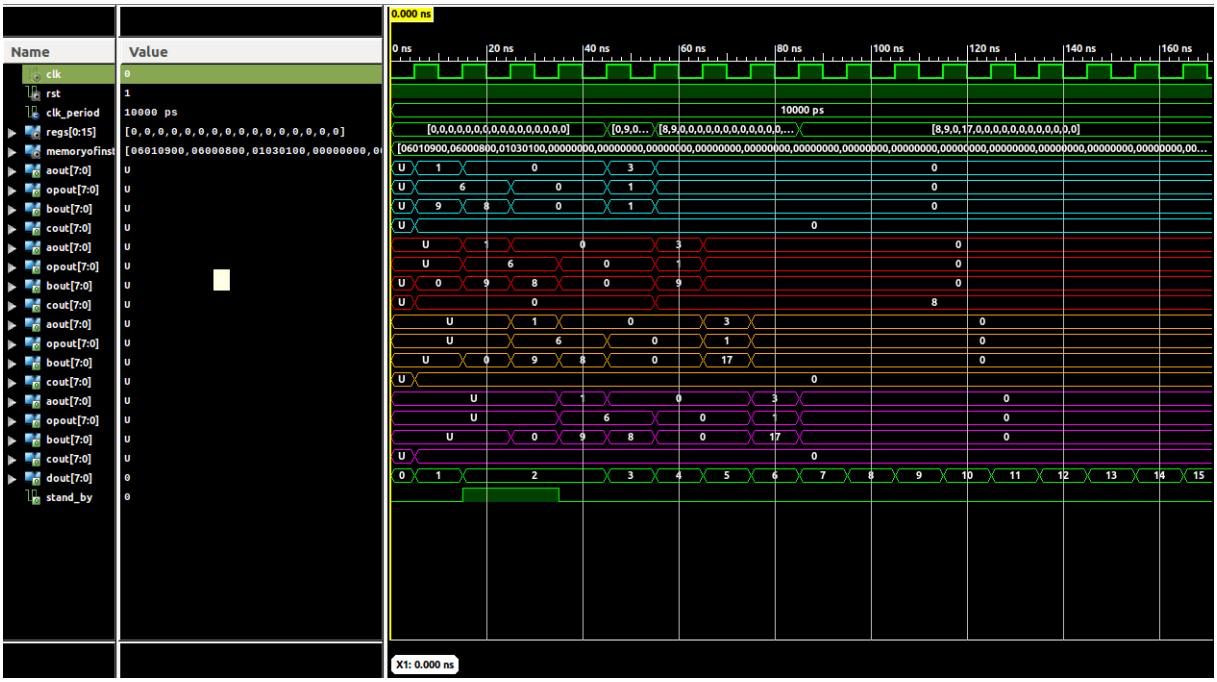


Diagramme obtenu avec gestion des aléas avec les instructions AFC R1 0x09 puis COP R3 R1