Rapport Projet
Compilateur & Processeur

UF : Architecture logicielle et matérielle des systèmes informatiques
Responsable d'uf : D.Dragomirescu

Wowk Timothée
4-IR-A2

Bravais Jérôme
06/06/2019

Partie compilateur & interpréteur

1 - Lex

Lex sert à l'analyse lexicale de notre programme. C'est ici que les différentes expressions régulières à reconnaître sont inscrites, ainsi que le token qui sera renvoyé pour notre analyse syntaxique. Nous avons rajouté une gestion des commentaires, qui ignore toutes les expressions régulières comprises entre */ et /* .

2 - Yacc

- Yacc permet, à partir des tokens que lui renvoie Lex, de vérifier la syntaxe de notre programme avec les règles que nous avons définies. Nous avons également pu définir des priorités entre les tokens, notamment au niveau des règles d'opérations mathématiques (i.e : multiplication prioritaire sur l'addition)

- L'ajout des règles error tPVR {yyerrok;} nous ont permis de continuer l'analyse du fichier lorsqu'une erreur est rencontrée. Cette règle notifie Yacc que l'erreur est acceptable et qu'il pourra reprendre son analyse après le prochain point virgule. Deux fonctions de gestion d'erreur ont également été implémentées, l'une pour compter les erreurs et afficher un message associé, et l'autre pour les warnings, si une erreur non fatale est détectée.

- Nous avons ajouté le fait de pouvoir parser notre programme depuis un fichier input.c. Si le fichier n'existe pas, un message en avertit l'utilisateur.

3 - Génération de l'assembleur

À la fin du parsing de notre fichier C, nous affichons dans le terminal le code assembleur sous forme de texte, mais nous générons également un fichier de sortie (sortie.banane) dans lequel nous écrivons toutes ces instructions, mais cette fois ci de manière binaire. Chaque ligne d'instruction est écrite sous forme de 4 octets: un pour l'opcode, puis trois pour les opérandes. Tout cela est bien évidemment fait uniquement si le fichier à été parsé sans erreur.

4 - Fonctionnalitées implémentées

Le langage supporté par notre compilateur permet de reconnaître et d'utiliser:

- les mandatory parts (expressions arithmétiques et conditionnelles, la déclaration d'entier, le « if-else », le « while », etc...)
- le else if
- les opérateurs && et ||
- les boucles « for »
- les commentaires (/* ... */)
- les expressions ternaires
- les pointeurs sans allocation de mémoire
- les fonctions : elles doivent être déclarées avant d'être appelées et obligatoirement au dessus du main, prennent 0 ou plusieurs entiers en paramètre, et doivent retourner un entier. Par manque de temps, nous n'avons pas pu implémenter les fonctions void, les pointeurs en argument, ainsi que le renvoi d'une erreur lorsque le programme ne contient pas d'instruction return dans les fonctions.

Les fonctions ont été implémentées de manière à ce que les variables des arguments soient au début des adresses mémoire qui lui sont allouées et sont suivies par l'adresse de retour. Lorsque l'on appelle une fonction, on sauvegarde la position du code assembleur dans le registre R15, puis cette valeur va être sauvegardée dans la variable _@RET_ (chaque fonction possède sa propre variable de retour. Attention, il n'est pas possible avec notre implémentation, qu'une fonction s'appelle elle même) au lancement de la fonction. Pour le « return », la valeur voulant être retournée est chargée dans le registre R14, puis de retour dans la fonction appelante, ce registre est sauvegardé dans une variable temporaire.

5 - L'interpréteur

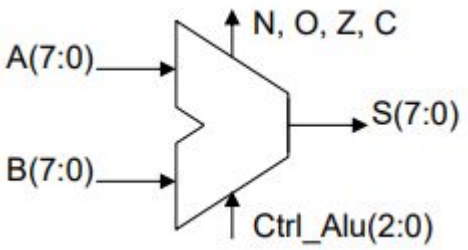
Il est utilisé au travers de la commande « python Interpret.py », il lit le fichier sortie.banane, puis régénère le tableau d'instructions assembleur. Les registres et la mémoire sont simulés au travers de deux tableaux. Ensuite, le programme parcours chacune des instructions, et les 'exécute en modifiant les différents registres, en chargeant ou sauvegardant des données dans la mémoire, ou bien en réalisant des JUMP dans le tableau d'instruction. Pendant cette exécution, il enregistre les différentes adresses mémoire qui ont été utilisées, puis les affiche à la fin. La même chose ayant été réalisé dans le parser, il suffit alors de comparer à l'adresse d'une variable, la valeur réelle et celle attendue, afin de vérifier le bon fonctionnement du programme.

Partie processeur

1. Les composants

a. L'unité arithmétique et logique (ALU)

L'unité arithmétique est le composant permettant de faire les opérations ADD, SOU, MUL, DIV.
Ce composant a **trois entrées** :

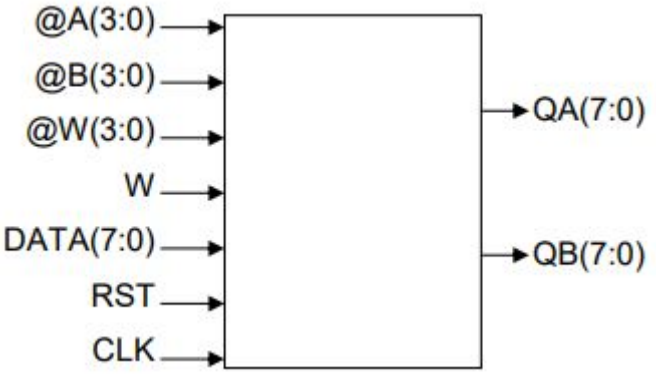


- A et B qui sont les éléments sur lesquels sera effectuée l'opération.
- Ctrl_Alu qui renseigne l'opération à réaliser.

Ce composant à **deux sorties** :

- S qui est le résultat de l'opération effectuée.
- Les flags C,N,Z,V qui fournissent des information sur l'opération.
    - C : Carry, renseigne sur la présence d'une retenue.
    - N : Négatif, indique que le résultat est inférieur à zéro.
    - Z : Zéro, indique que le résultat est nul.
    - V : Overflow, indique que l'opération a débordé et que le résultat ne peut pas être stocké dans les 8 bits de sortie.

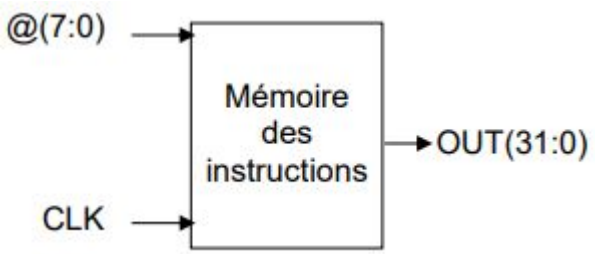
b. Le banc de registres



Les registres ont de nombreuses fonctionnalités dans un programme assembleur. Ils sont utilisés pour stocker des valeurs et des adresses mémoire.
Ci-contre un schéma du banc de registres résumant les différentes entrées et sorties du composant.
La lecture peut être asynchrone mais l'écriture est forcément synchrone pour éviter les conflits lecture/écriture. (Le Reset étant une écriture, il se fera de manière synchrone.) Notre banc de registres contient 16 registres de 8 bits chacuns.

c. La mémoire d'instruction

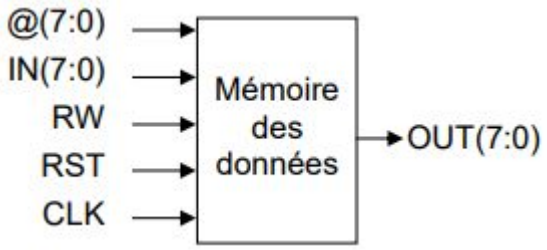
La mémoire d'instruction contient les instructions que le processeur exécutera. On peut les importer via un fichier ou les placer directement dans le code pour faciliter les tests. Les instructions sont stockées sur 4 octets dans le format suivant :
exemple : x"01030102"



01	03	01	02
ADD	R3	R1	R2

d. La mémoire de données

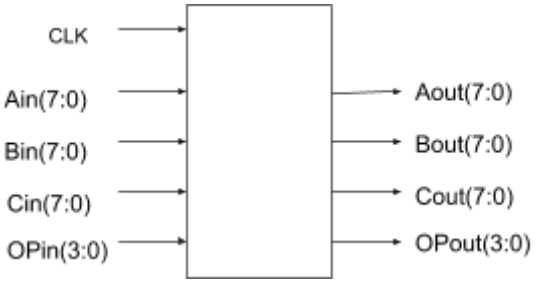
La mémoire de données permet de stocker des données en étant beaucoup moins limité en taille qu'avec les registres. Dans notre implémentation, nous utilisons un tableau de 256 cases de 8 bits pour représenter nos données. La lecture, l'écriture et le Reset se font de manière synchrone sur le front montant d'horloge.



2. Le CPU

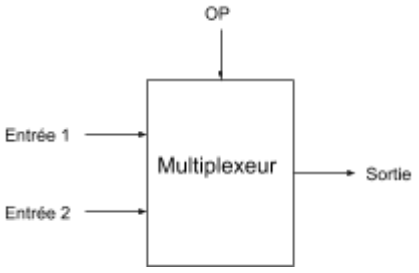
a. Les pipelines

Les pipelines permettent d'acheminer les données entre les différents étages du processeur. Lors du front montant de l'horloge, les données en entrée sont simplement transmises sur la sortie.



b. Le chemin de données

Le chemin de données est différent en fonction de l'instruction à effectuer, il faut donc traiter chaque cas et réaliser le chemin de données correspondant. Quatre multiplexeurs ont été placé sur le chemin de données, leur but est de décider quelle information transmettre en fonction de l'opération en cours.



Notre processeur prend en compte les chemins de données pour les instructions suivantes : AFC, COP, ADD, MUL, DIV, SOU, LOAD, STORE. Les aléas n'ont pas été implémentés par manque de temps.