Noémy Artigouha
Timothée Ficat

Projet Systèmes Informatiques
Du compilateur vers le microprocesseur

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Année 2016/2017

Sommaire

Introduction

Vous trouverez dans ce rapport, toutes les démarches et les étapes qui nous ont permis de développer un compilateur d'une version simplifiée du langage C avec des LEX et YACC. Mais également les phases de conception d'un Soc avec Microprocesseur de type RISC avec Pipeline qui va exécuter le code C (compilé par notre compilateur).
Nous présenterons donc ici les choix faits en termes d'opérations C, acceptés par le compilateur. Les choix d'opérateur assembleur générés par notre compilateur et exécutés par notre microprocesseur. Enfin les choix de conception de notre microprocesseur en VHDL.

I - Développement d'un compilateur avec LEX et YACC

1 - Compilateur manipulant des expressions arithmétiques de type C

Nous avons commencé par créer un analyseur lexical LEX qui permet de détecter les tokens du langage C. Nous avons ensuite réalisé l'analyseur syntaxique afin d'analyser du code C et produire un code assembleur correspondant à ce code C.

Pour vérifier que la syntaxe du code C que nous passions en entrée était bonne, nous avons mis à plusieurs endroit des affichages de l'état du compilateur. Par exemple pour le code suivant :

```
int main() {
        int a;
        int b = 2;
        if(b == 2) {
                a = 3;
        }
        return b;
}
```

Dans notre terminal nous affichions les messages suivants :

- Aucun paramètre fourni
- Déclaration d'une variable a
- Déclaration d'une variable b avec initialisation
- Entre dans un if
- Sort d'un if
- Fin de la fonction, retourne b

Afin de régler les nombreux conflits que nous avions à la compilation, nous avons déclaré à l'aide des %left et %right dans le Yacc des ordres de priorités des expressions. Ainsi les &, | sont prioritaires par rapport aux <, >, =<, => par exemple.

```
%right tEGAL
%left tAND tOR
%left tSUP tINF tSUPEG tINFEG tEGEG
...
```

1.1 - Création d'une table des symboles

Afin de garder en mémoire les informations tel que la déclaration de variables nous avons créé une table des symboles sous forme de tableaux.
Pour les variables, chaque case du tableau est une structure possédant :

Nom variable	Type	Initialisée	Profondeur dans le code
a	TYPE_INT	1	1
_	TYPE_INT	1	1

De plus, comme il est très courant d'avoir recours à des variables temporaires pour faire des calculs; celle-ci sont donc ajoutées temporairement à la table des symboles avec un "_", puis supprimées dès que possible.

1.2 - Gestion des if et while

Pour gérer les sauts conditionnels ou de boucle nous avons procédé de la manière suivante :
Si la condition d'un "if" est vrai alors on ne fait rien, sinon on saute tout le corps du "if" grâce à un saut conditionnel. Nous avons donc dû ajouter l'opérateur assembleur JMPC qui sautera à l'adresse spécifier seulement si le registre donné contient la valeur 1.
De même pour les boucles. La condition de la boucle est réévaluée à chaque itération et un saut conditionnel permettra d'exécuter le contenu de la boucle ou non. En revanche, à la fin du corps de la boucle un saut va renvoyer le pointeur d'instruction sur la condition de la boucle, permettant ainsi une nouvelle itération.

1.3 - Gestion des fonctions

Nous avons choisi de passer les paramètres des fonctions par la pile. De cette façon le nombre de paramètres n'est pas limité. En effet nous avions au préalable choisi de passer les arguments par registres, cependant pour ne pas encombrer trop de registres, nous avions choisi d'imposer un nombre maximum de paramètres.
Lors de l'appel de la fonction le registre de base de pile (BP) est augmenté pour empêcher la fonction appelée d'avoir accès au contexte de la fonction appelante. Ensuite on stocke l'adresse de retour dans le registre LR. L'opérateur JMP va ensuite sauter dans la fonction. A la fin de celle-ci un autre saut va revenir dans la fonction appelante grâce à LR puis nous dépilerons tout le contenu de la fonction appelée ainsi que ses paramètres. Nous avons également créé une table des fonctions similaires à la table des symboles pour garder en mémoire les données sur les fonctions. Chaque case du tableau est une structure possédant :

Nom de la fonction	Nombre de paramètres	Adresse
fun	2	1
main	0	2

L'adresse permet de faire des sauts dans cette fonction.
Le main est lui même une fonction mais il faut l'appeler au début du programme. Nous avons donc ajouté un saut automatique dans le main en toute première instruction assembleur.

1.4 - Gestion des erreurs

Une erreur se lève et affiche (*fprintf*) un message d'erreur quand :

- Une variable est utilisée mais non déclarée
- Une variable est utilisée mais non initialisée
- Une variable est déjà déclarée
- Une fonction est appelée mais non déclarée
- Une fonction est appelée avec le mauvais nombre d'arguments.

Une fois l'erreur levée, nous quittons l'exécution à l'aide de la fonction *exit*().

2 - Interpréteur du langage assembleur

Afin de tester et d'interpréter le fichier assembleur généré par notre compilateur, nous avons créé un interpréteur. Pour ce faire, nous avons créé un Lex et un Yacc capablent de lire du langage assembleur. Dans un premier temps, le fichier contenant le code assembleur est parsé à l'aide du code Lex et Yacc pour remplir une table contenant toutes les instructions assembleur du fichier. Cette table est sous la forme d'un tableau à double dimension :

Numéro de l'instruction		**Opérande**	**Int/Registre**	**Int/Registre**	**Int/Registre**
0	------>	AFC	2	1	0
1	------>	ADD	2	2	0

Une fois la table remplie, il ne reste plus qu'à interpréter ce tableau pour obtenir le résultat du programme C au préalablement écrit. Pour interpréter les expressions assembleur nous avons créé deux tableaux supplémentaires "mem" et "reg" pour simuler l'utilisation de la mémoire et des registres. En effet, si nous prenons par exemple l'opérande LOAD, elle permet de stocker dans un registre une valeur en mémoire.
Nous avons dû simuler un pointeur d'instruction (IP) permettant de connaître le numéro de l'instruction courante et ainsi pouvoir faire des sauts (JMP).
Nous avons aussi dû créer un pointeur de bas de pile (BP) permettant de faire des appels de fonctions en faisant monter ce pointeur au dessus du contexte de la fonction appelante.
Enfin, nous avons dû créer un registre de retour de fonction (LR) permettant de sauvegarder l'adresse de retour pour sauter dans la fonction appelante à la fin de la fonction appelée.

3 - Programme de test

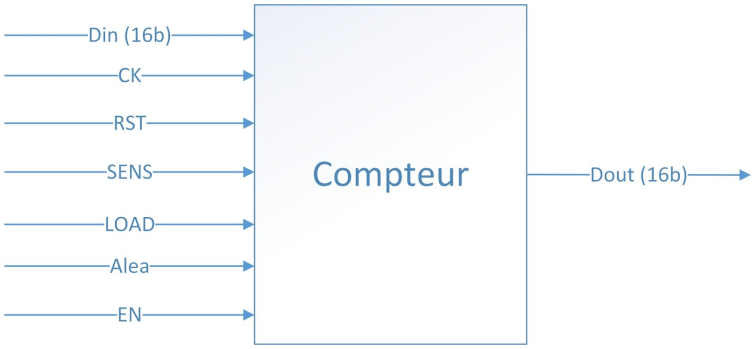
Pour tester l'ensemble de notre code, nous avons créé différents fichiers de tests tous nommés de la sorte : test_*numeroTest*.c. En modifiant le makefile nous pouvons choisir de tous les lancés les uns après les autres ou seulement ceux qui nous intéressent et ainsi optimiser la rapidité de nos tests.
Nous avons ainsi découvert un certain nombre de problèmes dans notre compilateur et avons pu les corriger avec succès.

4 - Améliorations possibles

Notre compilateur peut encore recevoir des améliorations mais nous avons choisit de passer à la phase de conception de notre microprocesseur par manque de temps.
Ainsi, nous n'avons pas pu compter le nombre de lignes du code pour retourner avec la levée d'une erreur, la ligne sur laquelle se trouve l'erreur.
Nous n'avons pas non plus gérer les pointeurs ou encore pris en compte la manipulation de nouveaux types comme String ou Char.

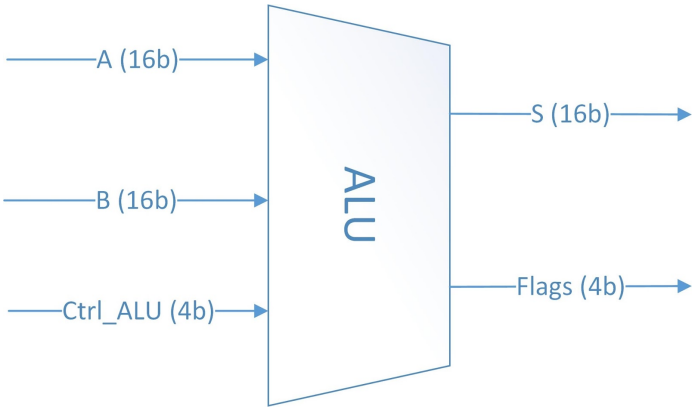
II - Conception d'un SoC avec microprocesseur de type RISC avec pipeline

1 - Compteur



Nous avions déjà réalisé un compteur 8 bits en TD. Nous l'avons donc adapté pour qu'il gère le 16 bits. Notre compteur va gérer notre pointeur d'instructions (IP). De cette façon le pointeur d'instruction va augmenter de 1 à chaque front montant de l'horloge.

2 - ALU

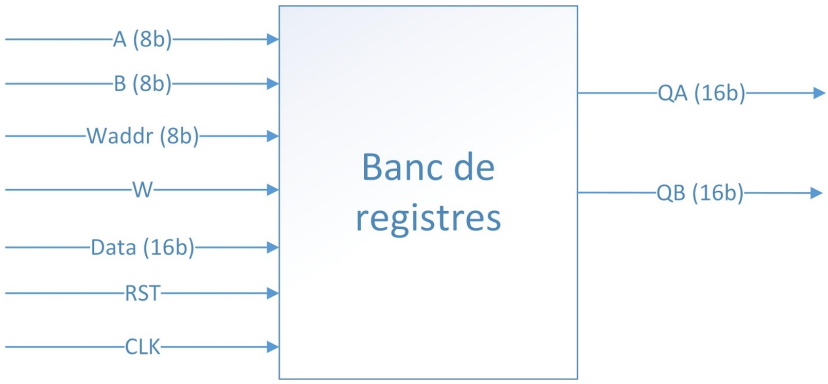


Notre unité arithmétique et logique va traiter tous les calculs demandés par l'instruction. Le bus Ctrl_ALU va définir quelle opération doit faire l'ALU selon l'opérateur de l'instruction. Ainsi, nous avons définit les valeurs de Ctrl_ALU comme suit:

0000 : S =A + B
0001 : S = A - B
0010 : S = -A
0011 : S = A < B
0100 : S =A <= B
0101 : S = A > B
0110 : S = A >= B
…..

Il ne gère pas les multiplications, ni les divisions. Celles-ci sont codées en assembleur dans le parser Yacc et fonctionnent très bien avec l'interpréteur.

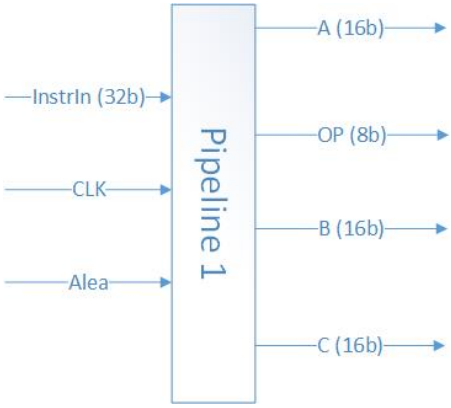
2 - Banc de registres



Le banc de registre gère l'accès aux valeurs stockées dans les registres ainsi que l'écriture de valeurs dans ces registres. Le bit W indique si on procède à une écriture ou non. Sinon dans tous les cas le banc de registre donne en sortie le contenu des registres A et B.

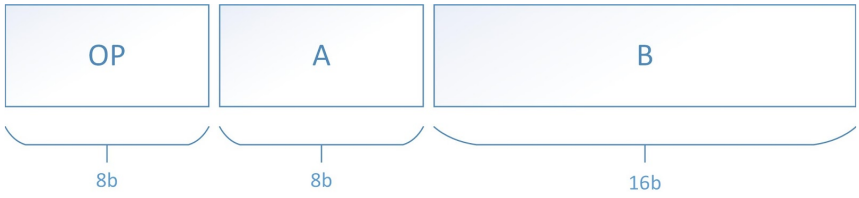
3 - Décodeur d'instructions

Nous avons décidé de décoder les instructions directement dans le premier pipeline afin de ne pas créer un nouveau module vhdl.

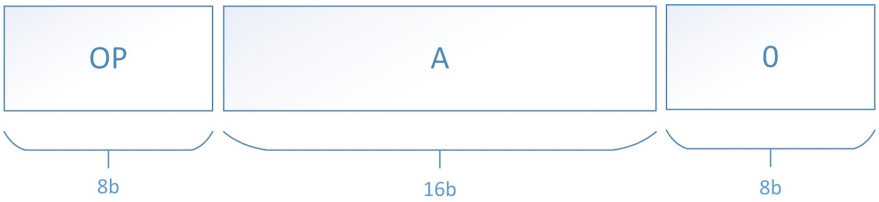


Ce premier pipeline va donc aiguiller les différentes parties de l'instruction entrante (InstrIn) vers les sorties A, B, C et OP. Pour cela il va analyser l'opérateur de l'instruction et la découper selon le résultat attendu. Nous avons opté pour quatre façons différentes de découper une instruction :

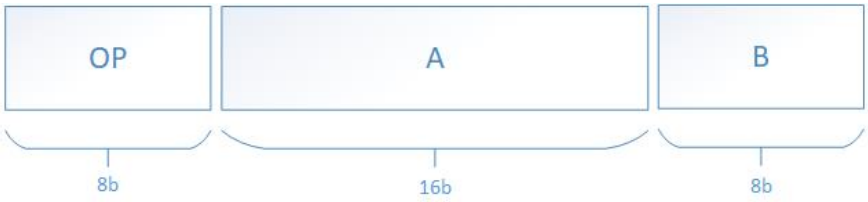
Si l'opérateur est une affectation (AFC) ou une lecture de la mémoire (LOAD) :



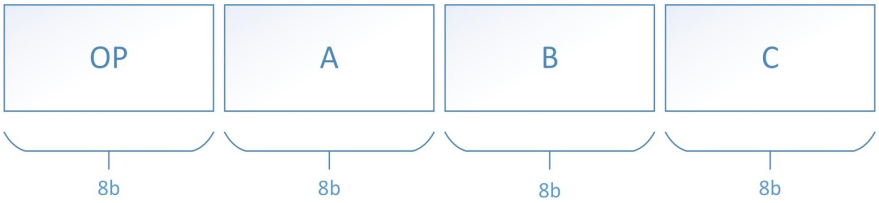
Si l'opérateur est un saut d'instruction (JMP) :



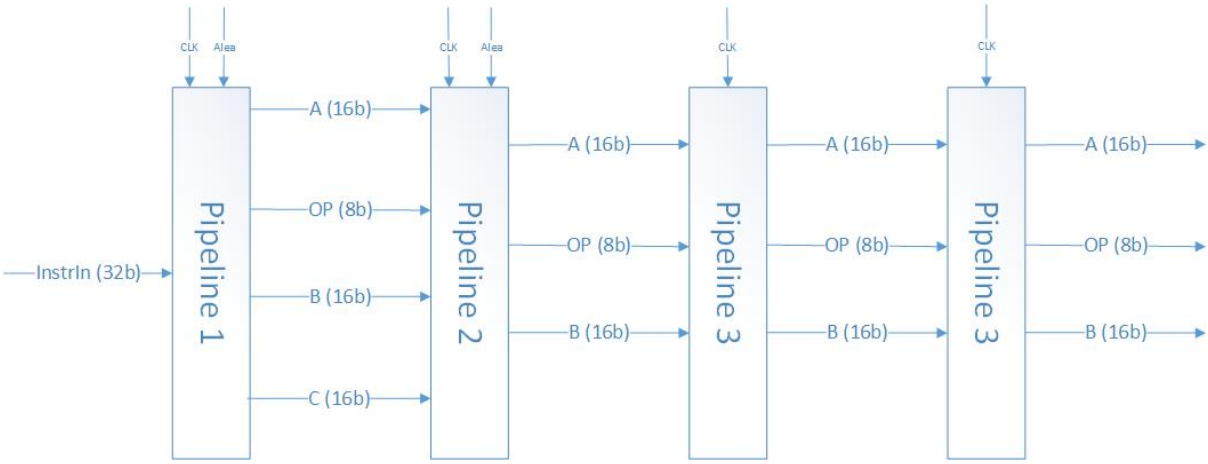
Si l'opérateur est une écriture dans la mémoire (STORE) ou un saut conditionnel (JMPC) :



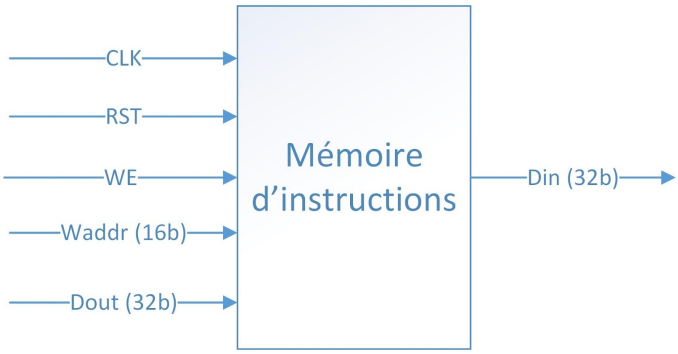
Et enfin, pour tous les autres opérateurs :



4 - Pipelines

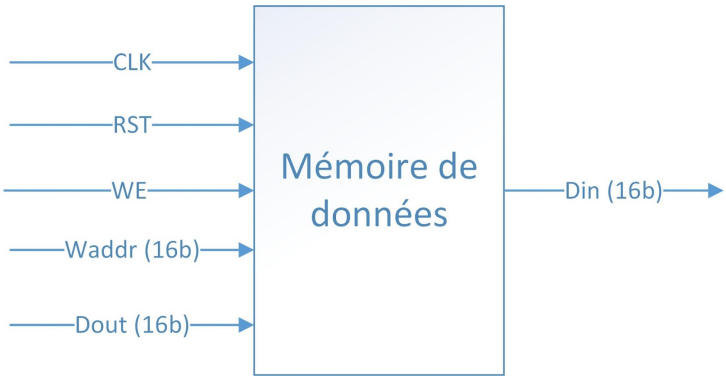


5 - Mémoire d'instruction



Comme les instructions sont sur 32 bits nous avons utilisé le module bram32 fourni pour créer notre mémoire d'instruction. Il nous a ensuite simplement suffit de l'initialiser avec un fichier remplit d'instruction en hexadécimal pour remplir la mémoire. Une fois bien câblé ce module marche et est simple à utiliser.

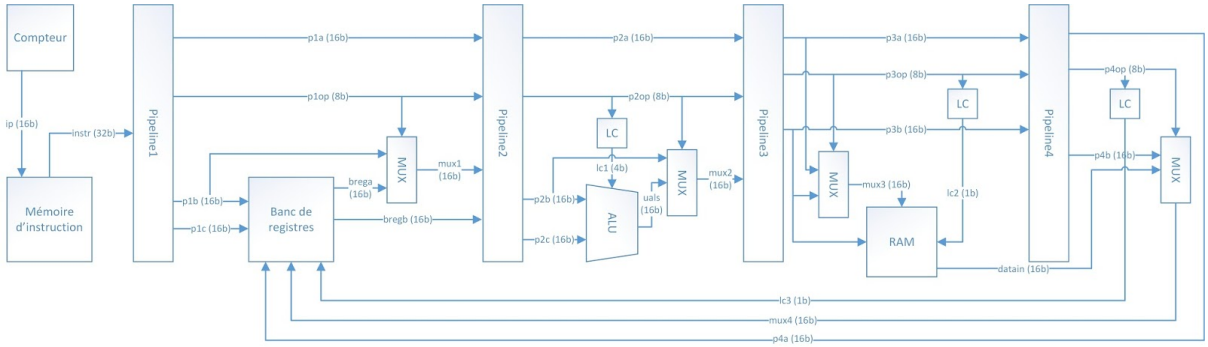
6 - Mémoire de données (RAM)



Notre ALU ne gérant que des données de taille 16 bits maximum nous avons choisis d'utiliser le module bram16 fournit pour créer notre mémoire de données. Une fois bien cablé nous avons pu utiliser ce module facilement.

7 - Chemin de données

Afin de simplifier la lecture de notre code nous avons nommé tous les fils du schéma suivant :



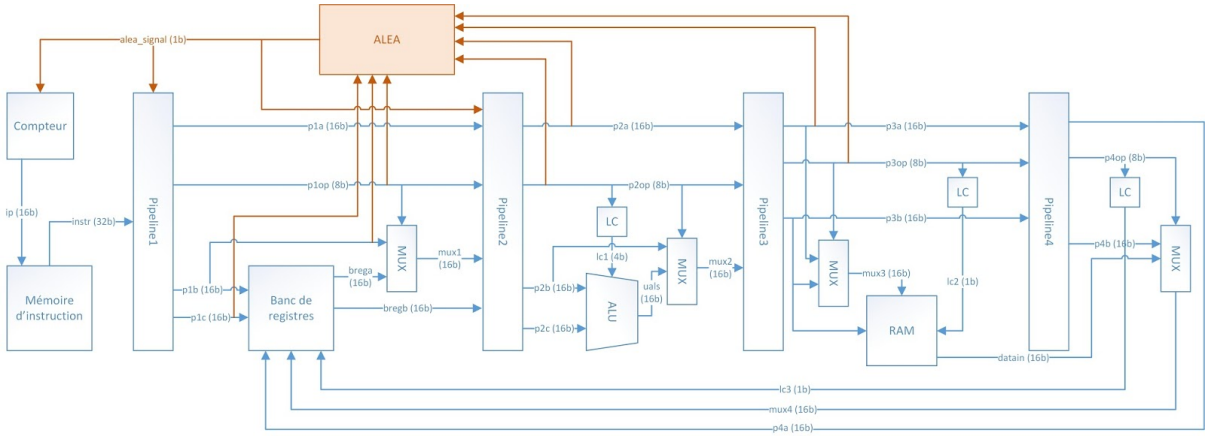
(Ce schéma est en annexe, en grand format)

Ceci nous a permi de nous y retrouver plus facilement dans la conception et la création de notre chemin de données. Une fois tout câblé, nous avons procédé à des tests et à du débogage afin de s'assurer que notre microprocesseur exécute bien les instructions demandées.

8 - Aléas

Il est courant d'avoir des aléas avec les registres. En effet, traverser les pipelines n'étant pas immédiat il arrive qu'une première instruction modifie un registre pendant que la suivante lise ce même registre. Le temps que l'instruction modifiant le registre arrive au quatrième pipeline, l'instruction lisant ce registre aura déjà lu sa donnée alors qu'elle n'aura pas encore été mise à jour. Dans ce cas il faut donc attendre que la première instruction se termine avant d'exécuter l'instruction suivante.
Nous avons donc détecté les aléas entre le pipeline 1 et le pipeline 2 mais aussi entre le pipeline 1 et le pipeline 3. Cette détection se fait grâce à l'analyse de l'opérateur qui indique si nous sommes en lecture ou en écriture mais également grâce à l'analyse des registres lus ou modifiés qui doivent être les mêmes.
Lorsqu'un aléa est déclenché plusieurs modules réagissent :
Le compteur s'arrête de compter et n'affiche plus sa valeur courante mais celle-ci moins 1.
Le pipeline 1 arrête de décomposer les instructions entrantes.
Le pipeline 2 injecte des NOP et ne prend plus en compte l'instruction du pipeline 1 afin de ne pas propager l'instruction causant l'aléa.
Le reste du microprocesseur continue à fonctionner afin de traiter l'instruction en cours et que celle-ci ne génère plus d'aléas avec l'instruction suivante.

Nous avons dû modifier notre chemin de données afin que le signal indiquant l'aléa puisse être propagé :

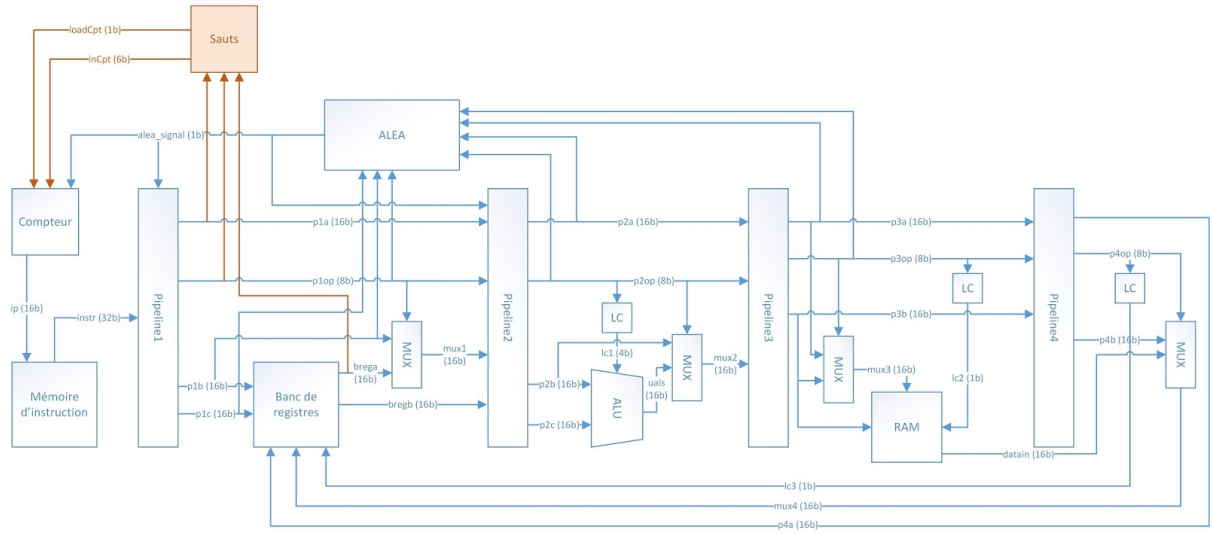


(Ce schéma est en annexe, en grand format)

9 - Sauts

Les sauts nous ont posé quelques problèmes. En effet, lorsqu'un saut entre dans le chemin de données, le temps qu'il affecte le compteur d'instruction, d'autres instructions avaient le temps de passer les premiers pipelines et ainsi de s'exécuter. A la manière d'un aléa, le pipeline 2 va bloquer les instructions qui viendront après le saut grâce à deux signaux permettant d'attendre deux front montant de l'horloge. En effet, un saut met exactement 2 clocks d'horloge avant d'être pris en compte. Une fois le saut exécuté tout revient à la normal et l'exécution continue.
Pour le saut conditionnel il faut s'assurer que la condition est vraie avant d'effectuer le saut. Si celle-ci est fausse le saut n'est pas détecté.

Nous avons dû modifier le chemin de données afin que les sauts puissent interagir avec le compteur d'instructions :



(Ce schéma est en annexe, en grand format)

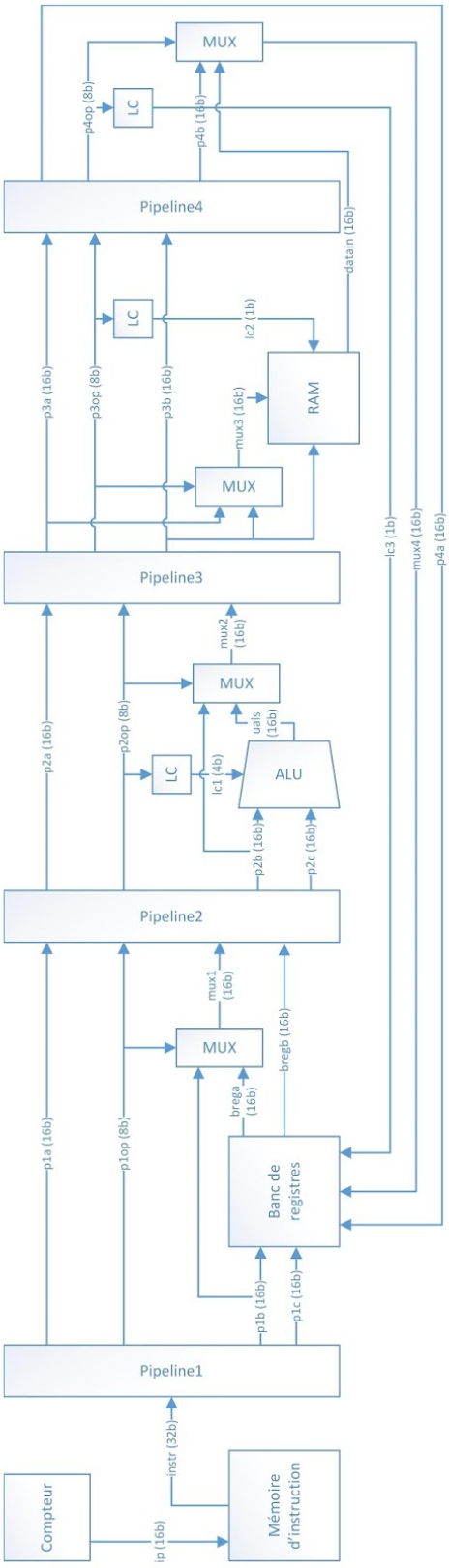
10 - Améliorations possibles

Certaines parties ne marchent pas, comme l'affichage, la multiplication et la division. Par manque de temps, nous n'avons pas eu le temps de les faire marcher. La multiplication et la division fonctionnent très bien avec notre interpréteur mais pas avec le microprocesseur. Pour le print, nous n'avons pas eu le temps de nous occuper de la partie VGA, même si cette partie nous semblait vraiment intéressante.

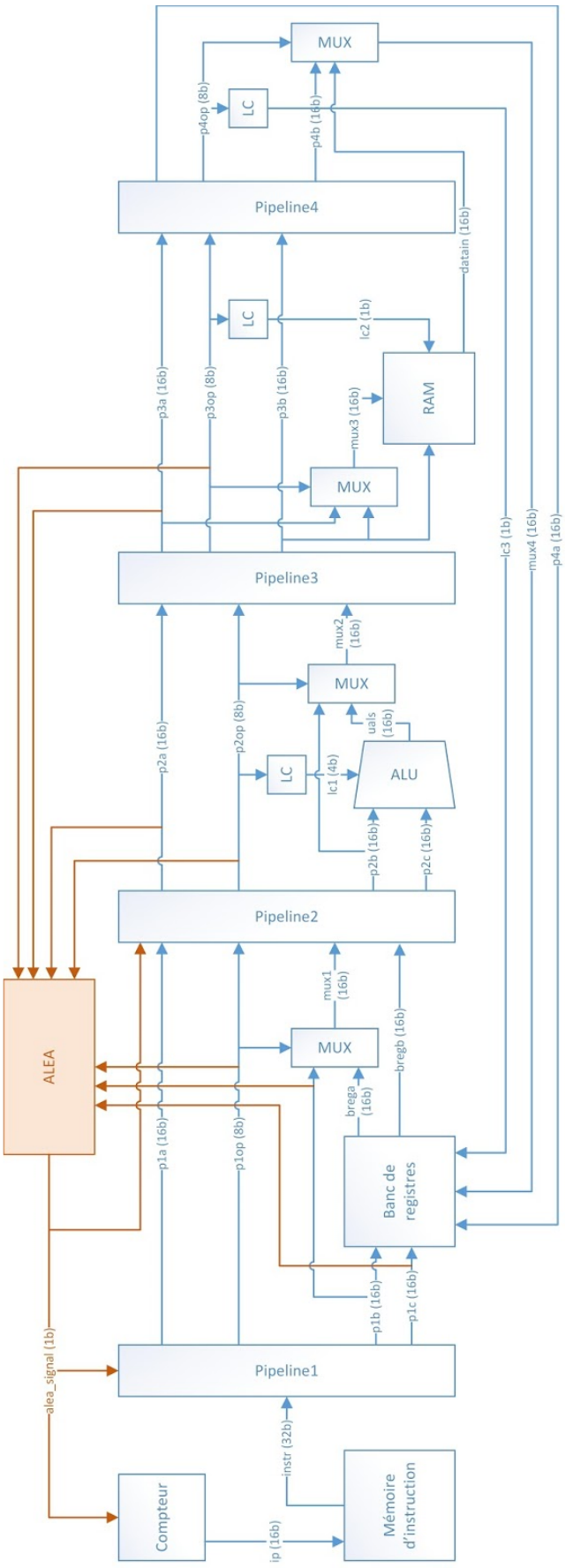
Conclusion

Ce projet fut réellement très intéressant de part sa longueur et sa richesse. Nous avons ainsi pu découvrir tous les aspects et les dessous de l'exécution d'un code C. De son écriture, en passant par sa compilation puis l'exécution du binaire généré par notre propre microprocesseur. Il nous est à présent possible de créer notre propre langage ainsi que notre propre machine pour exécuter ce langage. Nous regrettons néanmoins, de ne pas avoir réussi à tout finir car cela aurait pu nous en apprendre encore davantage.

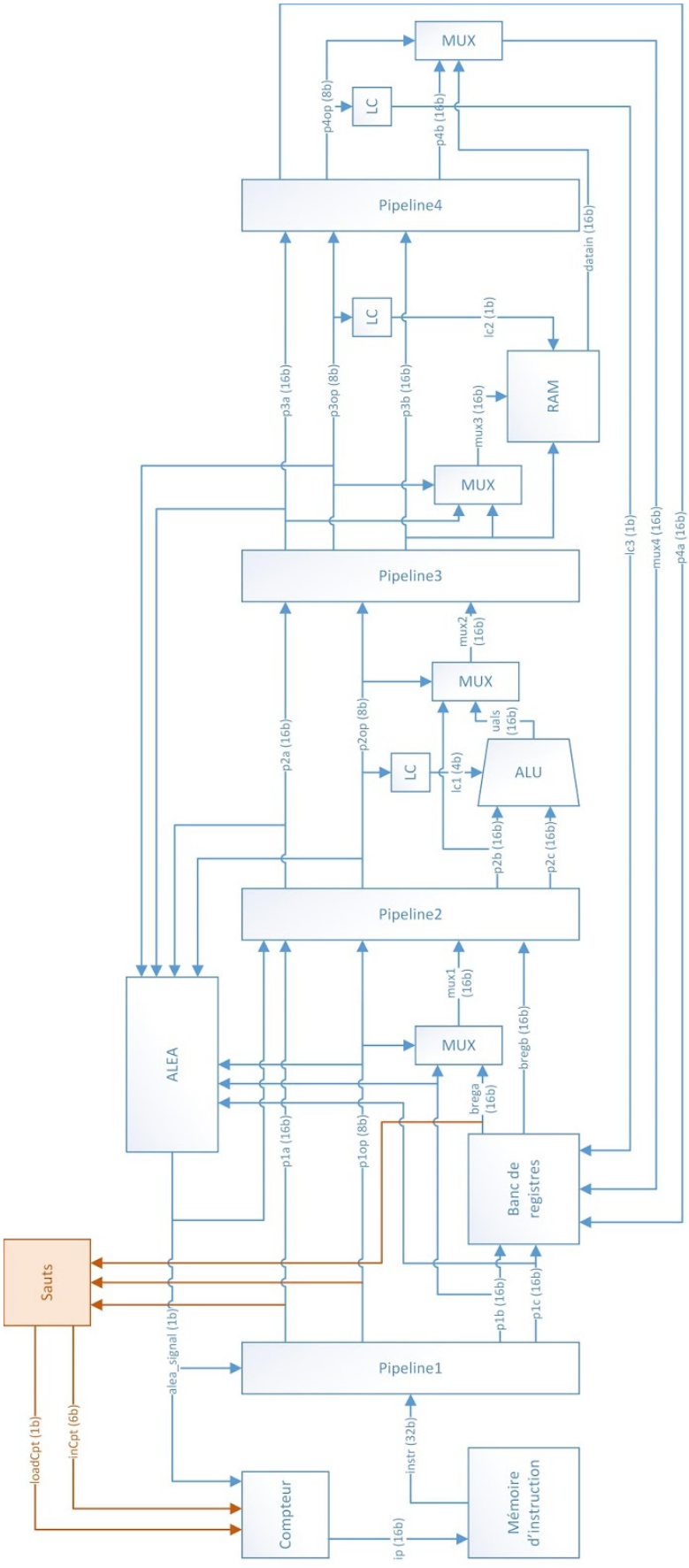
Annexes



Chemin de données


MUX
LC
p4op (8b)
p4b (16b)
Pipeline4
datain (16b)
LC
lc2 (1b)
RAM
p3a (16b)
p3op (8b)
p3b (16b)
mux3 (16b)
MUX
Pipeline3
lc3 (1b)
mux4 (16b)
p4a (16b)
mux2 (16b)
MUX
uals (16b)
p2op (8b)
p2a (16b)
LC
lc1 (4b)
ALU
p2b (16b)
p2c (16b)
Pipeline2
ALEA
mux1 (16b)
MUX
bregb (16b)
brega (16b)
p1a (16b)
p1op (8b)
Banc de registres
p1b (16b)
p1c (16b)
alea_signal (1b)
Pipeline1
instr (32b)
Compteur
ip (16b)
Mémoire d'instruction

Aléas


MUX
LC
p4op (8b)
p4b (16b)
Pipeline4
datain (16b)
LC
lc2 (1b)
RAM
p3a (16b)
p3op (8b)
p3b (16b)
mux3 (16b)
MUX
Pipeline3
lc3 (1b)
mux4 (16b)
p4a (16b)
mux2 (16b)
MUX
uals (16b)
p2a (16b)
p2op (8b)
LC
lc1 (4b)
ALU
p2b (16b)
p2c (16b)
Pipeline2
ALEA
mux1 (16b)
MUX
bregb (16b)
brega (16b)
p1a (16b)
p1op (8b)
Banc de registres
Sauts
p1b (16b)
p1c (16b)
alea_signal (1b)
Pipeline1
loadCpt (1b)
inCpt (6b)
instr (32b)
Compteur
ip (16b)
Mémoire d'instruction

Sauts