Architecture des Ordinateurs : Rapport de Projet

Licence 2 d'Informatique
Architecture des Ordinateurs
Projet de Borde Antoine et Clement Nerestan

3 mai 2015

Table des matières

1 Introduction

Au travers de ce rapport nous allons vous présenter le projet de l'UE architecture des ordinateurs ainsi que le travail que nous avons effectué dessus.

Ce projet a pour but d'étendre notre compréhension et notre maîtrise du simulateur d'architecture y86 qui est une version simplifié du x86 normes des processeurs actuels. Pour cela nous allons chercher à étendre le jeu d'instruction disponible en y86.

Le projet se présente sous forme de trois exercices articulés autour des trois étapes nécessaire a l'ajout d'une nouvelle instruction à savoir la libération des icodes, rendre possible l'utilisation de plusieurs instructions en une et pour finir l'implémentation des instructions à rajouter.

En effet le code d'une instruction commence toujours par un icode qui n'est rien d'autre que le numéro de l'instruction et qui donc est parfaitement indispensable. Hors il se trouve que dans le simulateur y86 tout les icodes sont déjà utilisés par une instruction. Il va donc nous falloir factoriser des instructions en une afin de libérer des icodes.

Pour ce qui est de permettre a notre simulateur d'effectuer plusieurs instruction en une nous devront trouver une astuce nous permettant d'injecter plusieurs instruction les une a la suite des autre chacune ayant un comportement différent. Nous allons pour cela jouer avec l'ifun qui ce situe juste après l'icode et qui permet de différencier plusieurs instructions possédant le même icode.

Quand a l'implémentation a proprement parler de nos instructions il s'agira de jouer avec le simulateur afin de bien gérer le comportement de nos instructions.

Pour accomplir ce travail nous voulions au départ nous séparer et essayer de faire moitié moitié. Mais les exercice se suivent de manière logique tout se qu'on apprend à maîtriser dans une question est indispensable pour répondre à la suivante etc... nous avons donc décider de travailler en même temps et de chercher à tout faire à deux.

Nous n'avons pas non plus cherché à produire un code le plus optimiser possible étant donner que depuis le début de cette UE l'objectif semble plus être de tout bien comprendre que d'être parfaitement performant et optimiser. Notre objectif principale est donc de sortir de ce projet avec une très bonne compréhension du fonctionnement du processeur y86.

Pour finir nous ajouterons que par soucis de clarté nous avons décider que tout les ajouts de code dans le fichier hcl respecteront une norme de présentation :

```
1) Toutes conditions(les if) rajoutées dans une liste de
condition sera ajoutées au dessus des conditions implémentées
dans la version initial du programme et séparées d'un retour à
la ligne.

2) Tout les if ayant pour but d'implémenter des instructions
différentes seront séparées d'un retour à la ligne dans la
liste de conditions.

3) Tout les ajouts d'implémentations dans une liste de
conditions seront écrit de haut en bas dans l'ordre dicté
par le sujet (d'abord opl puis rrmovl etc....).
```

Nous avons également fait le choix d'utiliser un github pour améliorer la clarté de notre travail. Malheureusement nos commit n'étant pas bien organisés nous avons préféré créer un nouveau dépôt. Ce dépôt ne contient que deux commit le premier étant le dossier sim originel et le deuxième le dossier sim après toute nos modifications. Ainsi il est possible en utilisant un logiciel proposant une interface graphique a git de lire, de manière plus claire et plus rapide, l'intégralité des changement effectué dans le simulateur.

2 Exercice 1 : De la place dans les opcodes

2.1 Factorisation de ALU et ALUI

L'objectif de cette exercice est de libérer des opcodes pour cela nous allons factoriser certains opcodes principalement ceux de l'ALU puis irmovl et rrmovl.

Pour commencer nous avons suivis les instructions données dans le sujet afin de confondre les OPL avec les IOPL. Une fois celà fais nous avons fait un petit ménage dans le code hcl en supprimant toute les itération des IOPL

Ensuite il nous devions trouvé une solution pour différencier les deux instruction et adapter le comportement de ces dernière. Pour cela nous avons suivit le conseil donné dans la consigne de l'exercice qui explique que la seule vrais différence entre un iaddl et un addl et que le premier charge une constante et de faite passe sa variable rA a RNONE (pas de registre) du coup il ne restait plus qu'a rajouter quelque if supplémentaire dans le code hcl qui regarde si rA est a RNONE pour différencier les deux instructions et faire varier leur comportement déjà très similaire. Cependant après ces modification les IOPL ne fonctionnaient toujours pas.

Heureusement un collègue nous a invité à regarder de plus prés le fichier "isa.c", nous y avons découvert un tableau dans lequel été regroupés toute les instructions et toute les info concernant la génération de leur code d'instruction. Nous avons alors comprit qu'il allait nous falloir jouer avec les taille des instructions en effet dans ce cas la les IOPL charge une constante qui est de faite est codé sur 4 octet et non pas un demi octet comme c'est le cas pour les registres. Il fallait donc prévoir plus de mémoire pour les instruction OPL afin qu'elle puisse stocker des constante si besoin. Après cela tout c'est mis a fonctionner

2.2 Factorisation de irmovl et rrmovl

Afin de libérer un second opcode nous avons dut factorisé irmovl et rrmovl. Après une observation du fonctionnement de irmovl et rrmovl nous avons compris que la factorisation s'effectuerait comme celle de ALU et ALUI nous avons donc réutiliser la même technique pour différencier les deux instruction et tout a marché du premier coup.

Les instructions a factoriser étant relativement simple de fonctionnement et surtout déjà implémenter nous n'avons pas eu de réel soucis pour cette exercice et ceux pour la version séquentiel comme pour la pipeline. Mise a part de la taille des instructions à modifier qui est une problématique auquel nous n'avions pas penser.

2.3 Bonus : call/push/popl/ret

Pour cette question bonus la consigne nous demander de factoriser les instruction call, pushl, popl, et ret en laissant totalement le choix des couple a former. Après une rapide observation on remarque très vite que les couples d'instructions pushl/call et popl/ret ont une implémentation quasiment similaire a une ligne prêt. On a donc décidé de factoriser ce couple là, même si cela libère un icode de moins que de factoriser les quatre en un.

Nous avons également changer de technique pour différencier les différentes instructions et utiliser pour cela l'ifun. Du moment que l'on a comprit ou changer l'ifun (au même endroit que les taille des instruction) et comment s'en servir la suite na pas été difficile on a juste rajouter des "if" la ou c'était nécessaire en se servant de l'ifun pour différencier les instruction. On a juste un peu hésiter pour le ret qui utilise le système des bulle dans la version pipeliner qu'on a eu un peu de mal a comprendre mais après une ou deux tentative nos test ont montré que toutes les instructions marchaient correctement.

3 Exercice 2 : Ajout du support d'instruction sur plusieurs cycle

L'objectif de cet exercice est de pouvoir exécuter des instructions fonctionnant en plusieurs cycles. Ce genre d'instructions plus complexes n'est finalement rien d'autre qu'une suite d'instructions plus simples qui s'enchaînent dans le bonne ordre pour arriver a un résultat.

Dans cette exercice toute les implémentation nous sont donner. On ne peut donc pas vraiment se tromper. Le vrai travaille ce situe dans le fait de comprendre précisément ce que le sujet nous fait faire.

3.1 Version Sequentielle

Ici nous allons indiqué l'utilité de ce qui nous a été demandé d'ajouterou modifier dans l'exercice.

Tout d'abord nous avons ajouté la variable instr_next_ifun cette variable va accueillir la valeur du futur ifun qui par défaut vaudra -1 (valeur qui fera passer le programme a l'instruction suivante de manière normal). Ce sera à nous de modifier manuellement cette ifun en fonction des besoins de nos instructions.

Ensuite nous avons ajouter le prototype de la fonction gen_instr_next_ifun ainsi que

```
if(gen_instr_next_ifun () != -1)
ifun = gen_instr_next_ifun();
else
```

Ce bloc est placer juste au dessus de la partie du code du simulateur qui génère le code de la prochaine instruction. Donc lorsque la variable instr_next_ifun sera a -1 rien ne changera dans le fonctionnement du simulateur. Mais si l'ifun est à une autre valeur on conservera le même code d'instruction dont l'ifun sera modifier.

Enfin nous avons ajouter

```
if (gen_instr_next_ifun() == -1)
avant
pc_in = gen_new_pc();
```

qui permet de mettre à jour le PC seulement si la nouvelle instruction n'est pas ajouter par le processeur, et donc d'empêcher le simulateur de continuer a lire notre programme.

3.2 Version Pipeliné

Les ajout sont les même que sur séquentielle à deux trois détails prêt lié au petites différences d'implémentation mais on s'adapte vite. La seul étape qui nous a demandé de nous y reprendre à deux fois est celle qui empêche la génération du code de l'instruction suivante car il fallait bien englober toute les instructions lié a cette génération avec le else.

3.3 Test

Avant de passer à la suite nous avons décidé de tester si notre implémentation était bien correcte. Mieux valait ne pas attendre d'implémenter nos instructions à plusieurs cycles pour constater que nous avion commis une erreur a cette exercice. Nous avons donc fait en sorte que quand le processeur exécute une instruction type OPL le ifun se mette a zéro et non a -1 pour faire boucler l'instruction. Le test a réussi et nous somme passer a la suite.

4 Exercice 3 : Ajout d'instructions

4.1 Ajout de "Enter"

L'objectif de l'ajout de cette instruction est de permettre d'effectuer les instructions pushl %ebp puis rrmovl %esp,%ebp, soit mettre en place le cadre de pile, en une seul ligne de code.

Pour cela il faut implémenter une instruction que l'on peut ajouter grâce aux icodes libérés dans l'exercice 1 et déclarer 2 instructions dans le tableau des instructions dans "isa.c" La première fera le pushl %ebp, et aura un ifun à 0, la deuxième aura un ifun a 1 et fera le rrmovl %esp,%ebp. Le principe est simple, on se sert de ce que l'on a implémenter dans l'exercice 2 pour dire au processeur qui si il rencontre une instruction enter avec un ifun à 0 il garde la même instruction en faisant passer son ifun a 1. Après ce deuxième cycle la variable instr_next_ifun repassera a -1 et le programme reprendra son cour.

L'implémentation des deux comportement de enter n'a pas été très difficile dans la version séquentiel car on a déjà bien vu comment différencier deux instruction selon leur ifun dans exercice 1 et les implémentation de pushl et rrmovl sont la pour nous aider. La difficulté a été de faire en sorte que enter ne prenne pas de paramètre mais a ce moment la du projet on maîtrise suffisamment bien le code hcl du simulateur pour donner les registre a utiliser manuellement et pour réussir cette implémentation sans trop de difficultés.

Pour la version Pipeline le principe reste le même mais nous avons eu un problème de taille à certains étages l'ifun n'étais pas disponible. Après recherche nous avons appris que nous pouvions retrouver ces ifun en allant les chercher directement dans la structure de l'instruction. Nous avons donc dut implémenter à la main l'accès aux ifuns à tous les étages de l'architecture pipeliner. Après sa il nous a suffit de reproduire se que nous avion fait dans la version séquentiel.

4.2 Ajout de "Mul"

Cette instruction, plus difficile à implémenter que la précédente, doit multiplier les deux registre donner en paramètre(ebx et ecx) et mettre le résultat dans le registre %eax. Nous nous sommes tout de suite demandé combien d'instructions il allait nous falloir pour réussir cela. Nous avons décider de partir sur trois comportement différent.

La première des instructions se chargera d'initialiser la valeur du registre eax a 0 afin d'éviter tout problème(soit un irmovl de 1 vers eax). La deuxième décrémentera le registre passé en deuxième paramètre qui fera office de compteur(soit un addl de 1 vers le registre ecx). La dernière ajoutera la valeur du premier registre en paramètre ebx a eax (soit un addl de ebx vers eax).

Nous nous somme d'abord concentré sur la version séquentiel. Nous avons tenter d'implémenter les 3 comportement un par un en testant entre chaque étape si l'instruction fonctionnait bien. Une fois nos trois instruction bien implémentées il fallait créer une boucle qui fonctionne selon nos besoin. Pour cela nous avons suivit les conseille de la consigne qui suggère de se servir des flag de contrôle.

On a donc rajouter mul a la liste d'instructions qui mettent a jour les flags de contrôle mais malheureusement cela ne fonctionnait toujours pas. Aprés avoir tâtonner au hasard quelque temps nous avons essayé de mettre a jour les flags de contrôle uniquement lors de l'instruction mul1(la décrémentation du compteur) et là tout a fonctionné. Nous n'avons pas de certitude quand aux raisons de ce bug, nous avons pensé à un délais de mise a jour des flags mais cela est un peu étrange.

Pour la version pipeline nous avons suivit le même schéma et bien-sur cela n'a pas suffit l'instruction additionner la valeur une fois de trop. Pas de surprise la consigne parlait d'un détail dont il fallait faire attention. Il nous a tout de suite paru évident que cela venez des flags qui ne se mettaient pas à jour dans tout les étage en un cycle. Cela aurait pu nous poser problème mais nous avions fait la question bonus 1 qui nous a obliger a regarder comment

fonctionner l'ajout de bulles dans la version pipeline. Nous avons donc injecté une bulle à l'étage décode et une a l'étage exécute et tout a fonctionné.

5 Conclusion

Nous somme plutôt satisfait de notre travail pendant ce projet. Avant de commencer,le fonctionnement du processeur y86 nous paraissait pour le moins abstrait, surtout la version pipeline dont le premier contact avec cette dernière en tp fut très douloureux. Mais suite à ce projet tout nous parait beaucoup plus clair et finalement a l'instar de la logique combinatoire le codage des instructions du processeur n'est qu'un assemblage de chose simple qui forme une chose d'apparence compliquer mais qui au fond ne représente pas de difficulté insurmontable.

Bien sur nous n'avons pas regardé dans le détail chaque ligne de chaque fichier du simulateur et certaine parties nous échappe encore mais nous pensons avoir bien assimilé les détails technique que ce projet visait à nous faire comprendre. Cela devrait être une très bonne préparation a l'examen final de fin d'année.

Si ce projet était très bien construit dans son apprentissage progressif des détails du fonctionnement du simulateur au fur et a mesure qu'on avance dans les questions. On regrettera peut être la difficulté que nous avons rencontré à nous séparer pour diviser le travail en deux. Mais au moins nous avons tout les deux acquis les compétence que nous étions censé acquérir.

Il y a quand même des passages ou nous avons été content d'avoir communiqué avec d'autres binômes pour demander un coup de main (principalement le binôme Paul Beziau et Candice Bentéjac). Notamment pour le fonctionnement du tableau dans le fichier isa.c qui contiend toutes les informations des instructions. Mais le fait d'avoir réussie les dernieres etapes du projet (à savoir les implementations des instructions), sans demander conseil à qui que ce soit, est pour nous la preuve que les connaissances que nous devions aquérir sont bel et bien aquise.

Pour conclure nous dirons que une des parties qui nous a finalement poser le plus de problème a étais la rédaction du rapport qui demandais un exercice de

vulgarisation sur le travail que nous avons effectué dans le code du simulateur, ce qui n'a pas été évident car le travail demandé était très technique.