

Rapport de projet d'Architecture des Ordinateurs

LABBE Emeric, PINERO Alexandre

4 mai 2015

Table des matières

1	Exercice 1 : De la place dans les opcodes	2
1.1	Factorisation de iaddl etc. avec addl etc.	2
1.2	Factorisation de irmovl avec rrmovl	2
2	Exercice 2 : Ajout du support d'unstruction sur plusieurs cycles	3
2.1	Version séquentielle	3
2.2	version Pipe-linée	3
3	Exercice 3 : Ajout d'instructions	4
3.1	L'instruction "enter"	4
3.1.1	Fetch Stage	4
3.1.2	Decode Stage	4
3.1.3	Execute Stage	5
3.1.4	Memory Stage	5
3.2	L'instruction "mul"	5
3.3	L'instruction "repstos"	5

1 Exercice 1 : De la place dans les opcodes

1.1 Factorisation de *iaddl* etc. avec *addl* etc.

Dans cette partie, pour l'implémentation séquentielle il y avait un piège, c'était de changer la taille des instructions contenues dans OPL et RRMovL. Donc dans le fichier "misc/isa.c", nous avons passé la taille des instructions *addl*, *subl*, *andl*, *xorl*, *sall* et *sarl* à 6 au lieu de 2 ainsi elles ont désormais la même taille que les instructions *iaddl*, *isubl*, *iandl*, *ixorl*, *isall* et *isarl*.

À l'étage Execute, les IOPL utilisent valC et les OPL valA. Pour les différencier, on regarde si ra est égal à 8 (ou RNONE) si c'est le cas on passe valC à l'ALUA sinon valA.

1.2 Factorisation de *irmovl* avec *rrmovl*

La factorisation des deux commande est la même chose que pour les OPL et les IOPL, il faut passé la taille de RRMovL à 6 et utiliser valC dans l'ALUA si ra est égal à RNONE si ce n'est pas le cas utiliser valA.

2 Exercice 2 : Ajout du support d'instruction sur plusieurs cycles

2.1 Version séquentielle

Ce n'était pas la chose la plus compliquée à faire, mais il fallait la faire correctement. Dans le fichier "seq/std-seq.hcl" nous avons ajouté l'instructions :

```
int instr_next_ifun = [  
1 : -1;  
];
```

Ce bloc indique au processeur quelle valeur de ifun utiliser après celle que l'on vient de traiter. Par convention, la valeur -1 indique que l'instruction est finie, et qu'il faut passer à la suivante. Le reste du code se trouve dans le fichier "seq/ssim.c"

2.2 version Pipe-linée

3 Exercice 3 : Ajout d'instructions

3.1 L'instruction "enter"

Pour ajouter l'instruction `enter` aux outils Y86 il a fallu modifier 5 de nos fichiers. Tout d'abord dans `isa.h` nous avons ajouté l'opcode `I_ENTER` dans le `itype_t`.

Dans `yas-grammar.lex` nous avons ajouté `enter` à la liste des instructions.

Dans `isa.c` nous avons ajouté `enter` et `enter1` au tableau `instruction_set` avec l'instruction suivante :

```
{"enter",    HPACK(I_ENTER, 0), 1, NO_ARG, 0, 0, NO_ARG, 0, 0 },
```

On lui donne ici la nouvelle instruction `enter`, qui est la première instruction à utiliser `I_ENTER`. Le paramètre 1 correspond à la taille de l'instruction.

Enfin nous avons ajouté `intsig ENTER` dans les déclarations des fichiers `seq-std.hcl` et `pipe-std.hcl`.

Ensuite nous avons implémenté cette instruction dans ces deux fichiers avec 2 comportements différents (`ifun = 0` et `ifun = 1`).

3.1.1 Fetch Stage

Nous avons ajouté `ENTER` à la liste `instr_valid`, puis nous avons mis la condition :

```
icode == ENTER && ifun == 0 : 1;
```

Ainsi après le premier passage de `ENTER`, le second prendra le même `icode` et le `ifun` suivant.

3.1.2 Decode Stage

Quand on a `icode == ENTER` : `srcA = REBP` et `srcB = RESP`.

Quand on a `icode == ENTER` et `ifun == 1` : `dstE = REBP` (instruction `rrmovl, %esp, %ebp`). Sinon `dstE = RESP` (instruction `pushl %ebp`).

3.1.3 Execute Stage

Dans le cas où `ifun == 1` : on soustrait 4 à `valB` :

```
aluA : icode == ENTER && ifun == 0 : -4;
aluB : icode == ENTER : valB;
```

Dans le cas où `ifun == 0` : `valB` ne change pas :

```
aluA : icode == ENTER && ifun == 1 : 0;
aluB : icode == ENTER : valB;
```

3.1.4 Memory Stage

On ajoute à `mem_write` l'instruction `ENTER` avec :

```
icode == ENTER && ifun == 0
```

On écrit ensuite le contenu de `valA` à l'adresse `valE` :

```
mem_addr : icode == ENTER && ifun == 0 : valE;
mem_data : icode == ENTER && ifun == 0 : valA;
```

3.2 L'instruction "mul"

Tout comme l'instruction "enter", nous avons ajouté l'opcode `I_MUL` dans le fichier "mics/isa.h", nous l'avons ajouté à la liste des instructions dans `yas-grammar.lex` et ajouter trois entrées dans le tableau des instructions de "mics/isa.c". Nous avons choisi de l'implémenter de la façon suivante :

```
EAX = 0;
while(ECX != 0){
    ECX--;
    EAX += EBX;
}
```

Les trois entrées dans le tableau servent alors aux trois étapes de l'algorithme :

- 0 - on initialise `EAX`
- 1 - on décrémente `ECX`
- 2 - on ajoute `EBX` à `EAX` si `ECX` supérieure à 0

– -1 - on sors de la boucle

Ainsi dans le code hcl de la version sequentielle, on commence avec l'ifun à 0 pour initialiser %eax à 0 puis on change d'état entre l'ifun à 1 et à 2. On fait cette opération tant que la variable cc est différente de 2. Autrement l'ifun passe à -1 et on passe à l'instruction suivante.

3.3 L'instruction "repstos"

Pour cet instruction, nous avons choisi l'implémentation suivante :

```
while(ECX > 0){  
    stos();  
    ECX--;  
}
```

où la fonction stos fait :

```
push EAX dans EDI  
EDI += 4
```

Comme pour l'instruction MUL, pour savoir si ECX est strictement positif il faut vérifier si "cc" est différent de 2. Dans le cas contraire on décrémente ECX et on repasse l'ifun à 1 pour refaire la commande STOS.