Compte rendu du projet
Architecture des ordinateurs

Gaëtan CHAMBRES Chrystelle PETUREAU

03/05/2015

Table des matières

Chapitre 1

De la place dans les opcodes

1.1 libérer I-ALU

Le but de cette question est de faire en sorte que toutes les opérations de type "iXX" soient confondues avec leur consoeurs de même type. Pour cela, nous avons renommé l'identifiant de l'opérateur I-ALUI en I-FREE1. On définit maintenant I-ALUI comme la même opération que I-ALU avec la ligne de commande suivante (fichier : isa.h, lignes 29 et 33) :

```
#define I_ALUI I_ALU
```

De ce fait dans le fichier isa.c, on commente la partie du case : (fichier : isa.c de la ligne 925 à la ligne 951)

```
case I_ALUI:
```

pour ne pas avoir d'erreur lors de la compilation, vu que les opérateurs I-ALU et I-ALUI sont, maintenant, confondu.

1.2 Distinguer IOPL et OPL

Dans cette question nous cherchons à corriger le fichier HCL car les opérateurs IOPL et OPL sont confondus.

1.2.1 Correction de la version séquentielle

Pour permettre l'utilisation distincte de I-OPL et OPL, nous avons effectué les modifications suivantes dans les fichiers HCL :
Dans le fichier seq-std.hcl, aux lignes 129 et 130, nous avons ajouté ces deux lignes :

```
icode in {OPL} && rA == RNONE : valC;
icode in {OPL } : valA;
```

Elles permettent de vérifier avant l'étage EXECUTE si le registre srcA de l'opération OPL vaut rA ou RNONE. Si le registre vaut RNONE, alors on lit la valeur de la constante valC, sinon on lit valA, le contenu de rA.

1.2.2 test de la version séquentielle

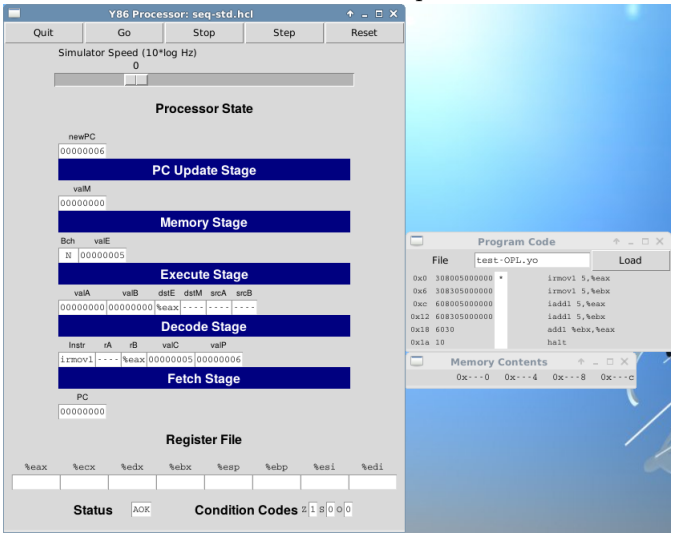
Nous avons testé cette correction de l'opérateur OPL ,après recompilation, avec le code suivant :

```
        .pos 0

        irmovl 5,%eax
        irmovl 5,%ebx
        iaddl 5,%eax
        iaddl 5,%ebx
        addl %ebx,%eax

        halt
```

Voici les scans écrans étapes après étapes.

L'étape 1 : l'initialisation

FIGURE 1.1 – Étape 1



Étape 2 et 3 :effectuer les opérations irmovl 5 dans le registre eax et irmovl 5 dans le registre ebx.

Étape 4 et 5 : ajouter la constante 5 aux registres eax et ebx.

Étage 6 : ajouter la valeur du registre ebx au registre eax.

FIGURE 1.2 – Étape 2 et 3

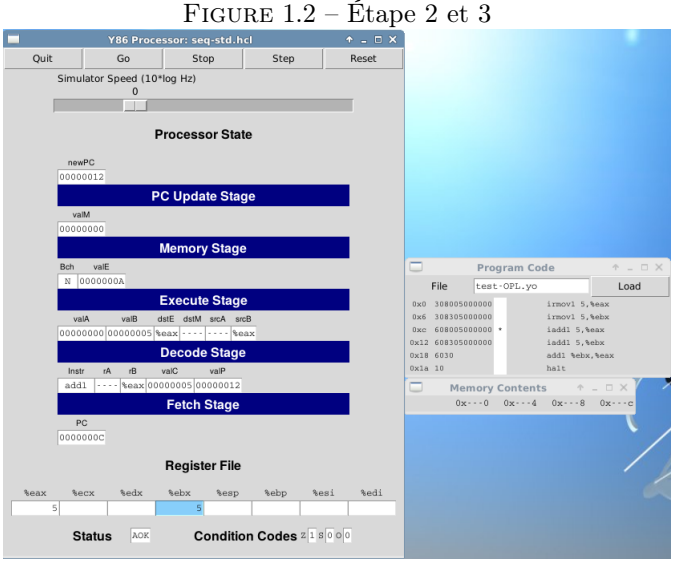
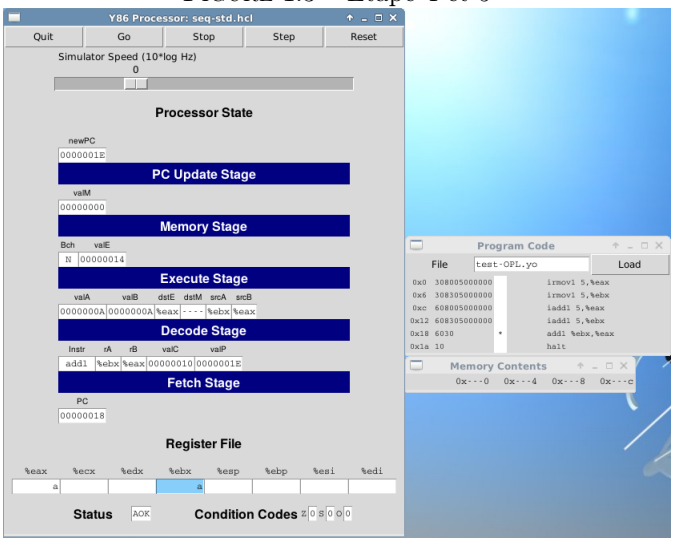


FIGURE 1.3 – Étape 4 et 5

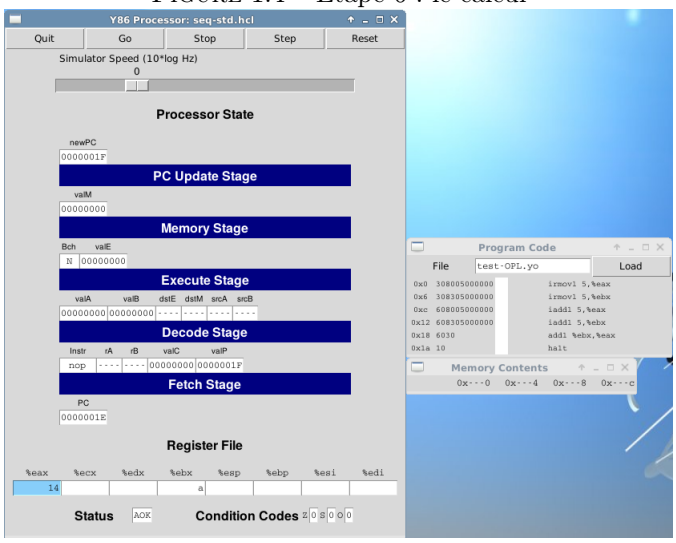


1.2.3 Correction de la version pipeline

Pour l'architecture pipeline, nous avons ajouté les lignes de code suivantes dans le fichier pipe-std.hcl à la ligne 204.

```
E_icode in {OPL} && D_rA == RNONE : E_valC;
E_icode in {OPL} : E_valA;
```

FIGURE 1.4 – Étape 6 : le calcul



Ce type de correction permet de faire la même vérification que pour la version séquentielle détaillée plus haut.

Cependant, lorsque nous avons testé le processeur avec le code suivant(version adaptée au pipeline)

```
.pos 0
irmovl 5,%eax
nop
irmovl 5,%ebx
nop
iaddl 5,%eax
nop
iaddl 5,%ebx
nop
nop
nop
nop
addl %ebx,%eax
halt
```

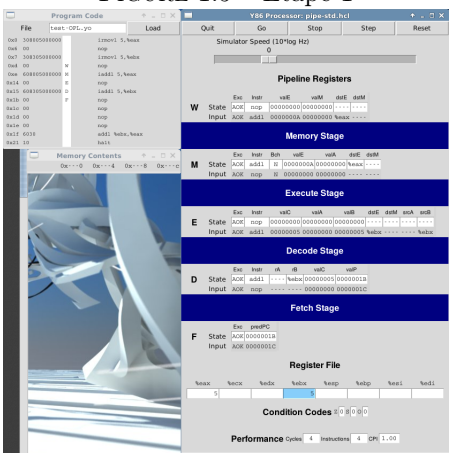
Les résultats obtenus n'ont pas été ceux escomptés. Le surnombre (volontaire) d'opérateurs NOPE nous permet d'être sûr que le problème ne vient pas de bulles manquantes.

L'erreur se présente au moment de l'opération "addl" pour laquelle le processeur lit le contenu des deux registres, comme voulu. Mais aussi une constante valC égale à 16 (en décimal) alors que le registre D-rA est différent de RNONE.

A l'exécution, le processeur ajoute au registre B le contenu de cette constante (valC) plutôt que celui du registre A.
Nous n'avons pas su résoudre ce problème. Voici les scans écran :

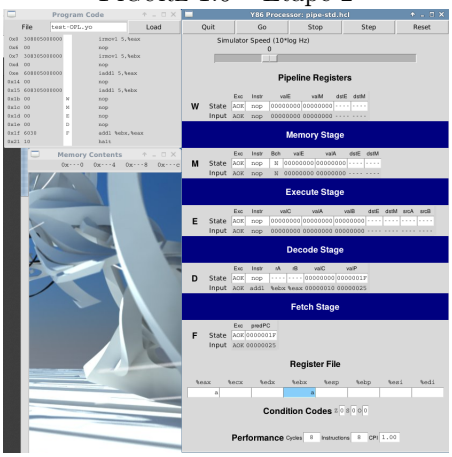
L'étape 1 : l'initialisation.

FIGURE 1.5 – Étape 1



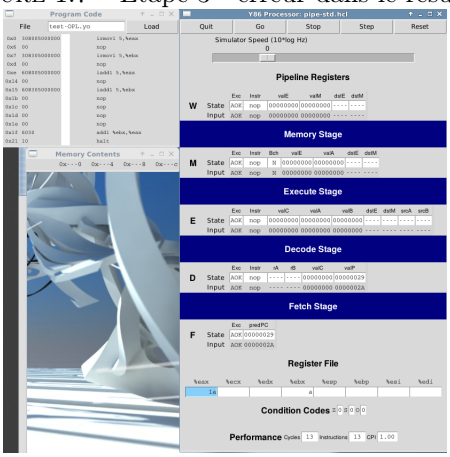
Étape 2 : effectuer l'opération irmovl 5 dans le registre eax.

FIGURE 1.6 – Étape 2



Étape 3 : effectuer l'opération irmovl 5 dans le registre ebx.

FIGURE 1.7 – Etape 3 - erreur dans le résultat



1.3 Factorisation de irmovl avec rrmovl

Le but de cette question est de faire le même type d'opérations pour libérer l'opérateur IRMOVL qui devient I-FREE2.

1.4 Factorisation des opérations push/pop/call/-ret

Le but est ici de faire le même genre d'opération pour libérer encore des opérateurs grâce à l'aide du champ ifun.
En regardant de près les similitudes dans le code HCL, on constate qu'il est beaucoup plus intéressant de factoriser par paires push/call et pop/ret plutot que par paire push/pop et call/ret ou bien encore le quadruplet entier push/-pop/call/ret.
En effet, les instructions push et call effectuent une écriture mémoire alors que les instructions pop et ret font, quant à elles, une lecture mémoire.
Un problème sur l'ENT lors de la semaine de vacances universitaires nous a empêché de effectuer ces modifications.

Chapitre 2

Ajout du support d'instruction sur plusieurs cycles

On cherche à créer un pas à pas dans le fonctionnement du processeur pour qu'une instruction se déroule sur plusieurs cycle. Le principe de base est que le processeur injecte lui-même les instructions suivant leurs valeurs de ifun.

2.1 Dans la version séquentielle

Dans le fichier seq-std.hcl à la ligne 92, nous avons ajouté une instruction qui précise pour quelles valeurs le processeur passe à l'instruction suivante.

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

Dans le fichier ssim.c à la ligne 375, nous avons ajouté le prototype de la fonction "gen-instr-next-ifun" qui "lit" le compteur d'instructions.
On ajoute cette fameuse fonction à la ligne 667.

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

Enfin à la ligne 772, toujours dans le fichier ssim.c, on ajoute l'instruction :

```
if (gen_instr_next_ifun() == -1){
        pc_in = gen_new_pc();
}
```

Cette ligne de commande permet de calculer la nouvelle valeur du compteur ordinal.

2.2 version pipe-line :

Les modifications sont très similaires à la version séquentielle. Dans le fichier pipe-std.hcl à la ligne 138, nous avons ajouté une instruction pour préciser pour quelles valeurs le processeur passera à l'instruction suivante.

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

Dans le fichier psim.c à la ligne 1327, nous avons ajouté le prototype de la fonction "gen-instr-next-ifun" qui "lit" le compteur d'instructions. On ajoute cette fameuse fonction à la ligne 1361 du fichier.

```
if(gen_instr_next_ifun () != -1)
        if_id_next->ifun = gen_instr_next_ifun();
        fetch_ok= TRUE;
else
        fetch_ok=get_byte_val(mem, valp , &instr);
```

Enfin à la ligne 1388 toujours dans le fichier psim.c, on ajoute l'instruction :

```
if (gen_instr_next_ifun() == -1){
        pc_next->pc=gen_new_F_predPC();
}
```

Cette instruction permet de calculer la nouvelle valeur du compteur ordinal.

Chapitre 3

L'ajout d'instructions

Le but de cet exercice est d'ajouter des instructions au jeu de base des instructions du processeur Y86.

3.1 L'instruction enter

Dans le fichier isa.h à la ligne 29, nous avons remplacé I-FREE1 par I-ENTER. Dans le fichier yas-grammar.lex à la ligne 5, nous avons ajouté |enter . Dans le fichier isa.c à la ligne 51, nous avons ajouté les lignes de commandes :

```
{"enter", HPACK(I_ENTER,0),1,NO_ARG,0,0,NO_ARG,0,0},
{"enter1", HPACK(I_ENTER1,0),1,NO_ARG,0,0,NO_ARG,0,0},
```

Enfin dans le fichier seq-std.hcl à la ligne 41, nous avons ajouté la ligne :

```
intsig ENTER 'I_ENTER'
```

ainsi que dans le fichier pipe-std.hcl à la ligne 40.

3.2 L'ajout du code ifun

Dans le fichier seq-std.hcl à la ligne 95, nous avons ajouté :

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

ainsi que dans le fichier pipe-std.hcl à la ligne 141.

3.3 L' instruction mul : implémenter la multiplication

Nous avons implémenté l'opération multiplication grâce à l'algorithme très connu d'additions successives. Comme demandé dans la consigne, on a supposé

que les 2 opérandes sont des nombres positifs. On supposera implicitement que le registre eax recevra le résultat de l'instruction mul.
Il faut noter que cette instruction efface le contenu du registre eax sans aucune sauvegarde. De plus, les instructions :

```
        mul %ebx, %ecx
```

et

```
        mul %ecx, %ebx
```

n'auront pas la même complexité si ebx et ecx sont différents.
Enfin, il faut noter que les instructions :

```
        mul %XXX, %eax
```

et

```
        mul %eax, %XXX
```

donneront un résultat totalement incohérent.
Une fois toutes ces remarques prises en compte, pour réaliser cette instruction nous avons créé une boucle avec ifun comme compteur, comme dans l'algorithme naïf. Il sera initialisé avec le premier opérande.
Dans la version pipeline, nous devons faire attention que l'étape "memory" soit atteinte pour soustraire 1 au champ ifun, sinon le calcul sera totalement faux.
Dans le fichier seq-std.hcl à la ligne 42, nous avons ajouté la ligne de commande :

```
intsig cc 'cc'
intsig REAX
```

ainsi que dans le pipe-std.hcl à la ligne 41.

3.4 Les instructions lods/stos/movs

L'instruction lods lit en mémoire à l'adresse esi, stocke le résultat dans eax, et ajoute 4 au pointeur esi.
L'instruction stos écrit en mémoire à l'adresse edi le contenu de eax, et ajoute 4 au pointeur edi.
Enfin, l'instruction movs lit en mémoire à l'adresse esi, écrit la valeur en mémoire à l'adresse edi, et ajoute 4 à esi et edi.
Ces trois instructions ont donc besoin des registres esi, eax et edi et une constante (4).

3.5 L'instruction repstos

L'instruction repstos répète l'instruction stos autant de fois que le contenu de ecx. A la fin de l'exécution de l'instruction repstos, le contenu du registre ecx

est égal à 0.
On note que cette instruction est possible uniquement parce que l'instruction stos n'utilise pas le registre ecx.

3.5.1 Version séquentielle

Nous créons, dans le fichier seq-std.hcl, une boucle dans la quelle l'instruction stos est effectuée et 1 est soustrait au contenu du registre ecx à chaque tour de boucle.

3.5.2 Version pipeline

De la même manière que dans la version séquentielle, nous créons, dans le fichier pipe-std.hcl, une boucle dans la quelle l'instruction stos est effectuée et 1 est soustrait au contenu du registre ecx à chaque tour de boucle.
Comme dans toute boucle dans un système pipeline,il faut faire attention que le calcul soit arrivé à l'étape "memory" avant de décompter le compteur ordinal sinon le résultat sera faux, bien-sûr.