Rapport de projet : Architecture des ordinateurs

Benjamin Saint-Sever, Steven Ratton

24 avril 2014

Table des matières

1 Introduction **2**

2 Exercice 1 **3**

2.1 implémentation de leal 3

2.2 Architecture séquentielle 5

2.3 Architecture pipelines 7

Chapitre 1

Introduction

Le but de ce projet est de montrer la possibilité d'extension de l'architecture y86, en ajoutant des instructions et en manipulant les prédictions de branchement. Ce projet a pour vocation de nous enseigner la programmation de langage machine tout en tirant parti au mieux des capacités des ordinateurs.

Chapitre 2

Exercice 1

2.1 implémentation de leal

Description

Le but de ce premier exercice est de créer une nouvelle instruction "leal" pour "load effective address", cette derniere est existante sous l'architecture x86. Cette instruction charge l'adresse de la source dans dest (leal (%regS),%regD). On souhaite que cette instruction permette un déplacement mémoire, leal depl(%regS),regD. L'avantage de cette solution est que l'on peut effectuer cette opération avec une seule instruction. Son equivalent est : "rrmovl %regS,%regD ; iaddl depl,%regD"

Assembleur

Il est possible d'ajouter cette instruction sans consommer un nouvel opcode, il suffit de reprendre le formalisme de irmovl est de donner une valeur de ifun différente afin de faire la distinction (ifun =1).

Premiere étapes, on insère dans le code assembleur la déclaration de cette nouvelle instruction :

1. isa.c

```
    instr_t instruction_set[] =
{
/*Ajout de leal avec iFun a 1*/
{"leal", HPACK(I_IRMOVL, 1), 6, M_ARG, 1, 0, R_ARG, 1, 1 },
```

2. yas-grammar.lex

```
Instr           leal|rrmovl|rmmovl|mrmovl|irmovl|
```

Fichier test_leal.ys :

On déplace de 8 à partir de l'adresse du registre %eax et on attribue l'adresse dans le registre %ebx :

```
irmovl 3, %eax
       leal 8(%eax), %ebx
       halt
```

Code HCL :

On ajoute le code d'instruction irmovl pour identifier leal :

```
intsig LEAL       'I_IRMOVL'
```

2.2 Architecture séquentielle

Etape FETCH :
On ne modifie rien puisque l'on interprète le même fonctionnement que l'instruction irmovl.

Etape Decode :
le contenue du registre 'B' (Source) est attribué à la source A.

```
int srcA = [
        #icode vaut irmovl ou leal, pour obtenir leal -> ifun=1
        (icode == LEAL) && (ifun == 1):rB;
];
```

le contenue du registre 'A' est attribué à la E destination afin d'écrire la nouvelle adresse de destination.

```
int dstE = [
        (icode == IRMOVL) && (ifun == 0) : rB;
        (icode == LEAL) && (ifun == 1) : rA;
];
```

Etape Execute :
On place ValC dans l'aluA et ValA dans l'aluB, cela permettre d'effectuer le calcul de déplacement :

```
int aluA = [
        (icode == IRMOVL) && (ifun == 0) : valC;
        (icode == LEAL) && (ifun == 1): valC;
        #Deux lignes pour la lisibilité mais on peut aussi
          utiliser seulement icode in {irmovl} : valC
];
int aluB = [
        (icode == IRMOVL) && (ifun == 0) : 0;
        (icode == LEAL) && (ifun == 1): valA;
];
```

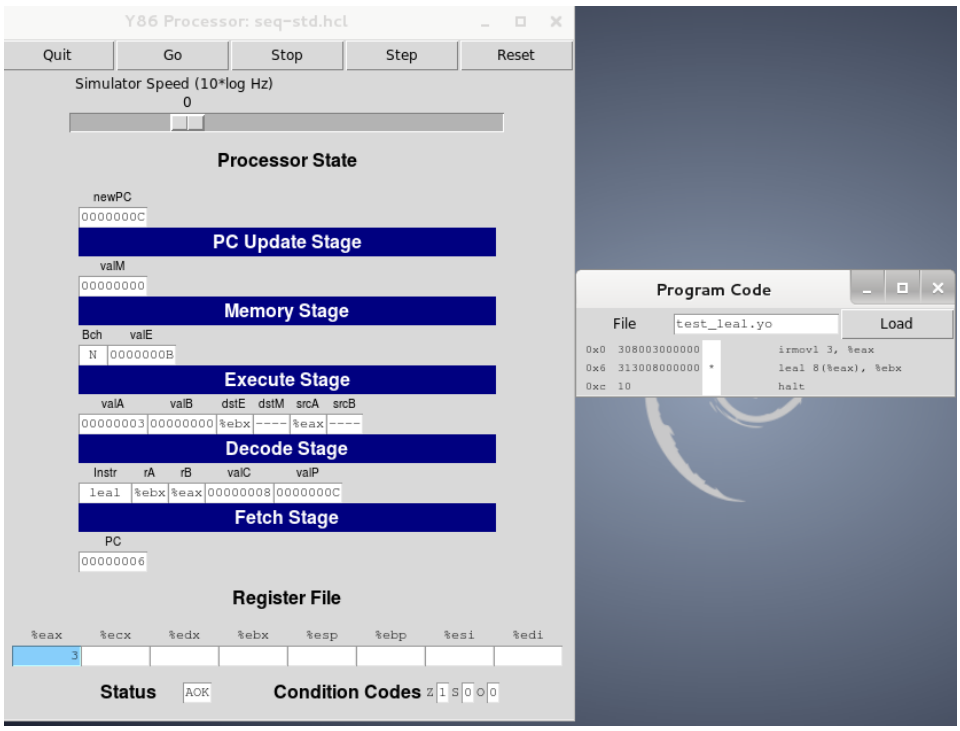

FIGURE 2.1 – Fetch

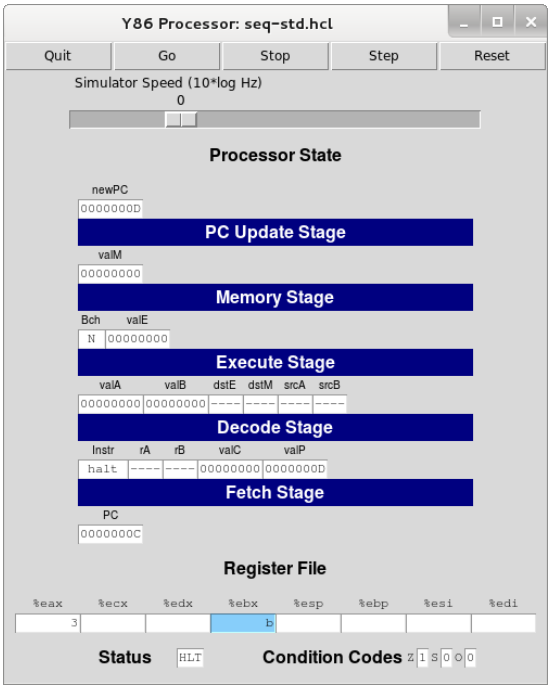

FIGURE 2.2 – Decode

2.3 Architecture pipelines

Afin de pouvoir identifier la bonne instruction à l'étage Decode il est nécessaire de récupérer la valeur de ifun qui permettra de différencier irmovl de leal, on ajoute donc la déclaration suivante :

```
intsig D_ifun 'if_id_curr->ifun'          # Instruction function
```

La prise en compte de leal dans la hcl pipeline diffère peut de l'architecture séquentielle, il faut néanmoins prendre en compte le ifun et icode des étages concerner.

Etape Decode :
Meme chose que pour l'architecture seq :

```
int new_E_srcA = [
        D_icode in { RRMOVL, RMMOVL, OPL, PUSHL } : D_rA;
        (D_icode == IRMOVL && D_ifun == 1) : D_rB;
        D_icode in { POPL, RET } : RESP;
        1 : RNONE; # Don't need register
];

int new_E_dstE = [
        D_icode in { RRMOVL, OPL, IOPL } : D_rB;
        D_icode in { PUSHL, POPL, CALL, RET } : RESP;
        (D_icode == IRMOVL && D_ifun == 1) : D_rA;
        (D_icode == IRMOVL && D_ifun == 0) : D_rB;
        1 : DNONE;  # Don't need register DNONE, not RNONE
];
```

Etape Execute :
Dans l'ALUA on utilise valC donc on ne modifie rien.

```
int aluA = [
        E_icode in { IRMOVL, RMMOVL, MRMOVL, IOPL } : E_valC;
];
```

Dans l'ALUB on souhaite utiliser valA, IRMOVL n'utilise rien donc il utilise la condition qui suis.

```
int aluB = [
        (E_icode == IRMOVL && E_ifun == 1) : E_valA;
        E_icode in { RRMOVL, IRMOVL } : 0;
        # Other instructions don't need ALU
];
```

eax vaut 3, on ajoute 8 à l'adresse de ce dernier et on stocke le résultat de dans ebx

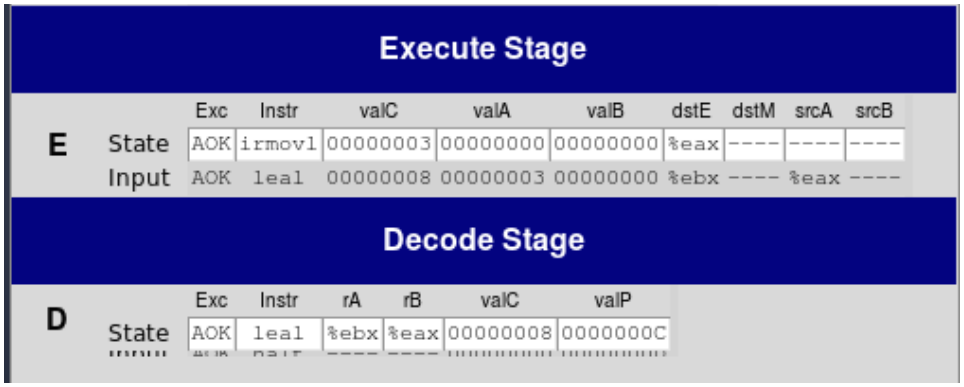

FIGURE 2.3 – Decode

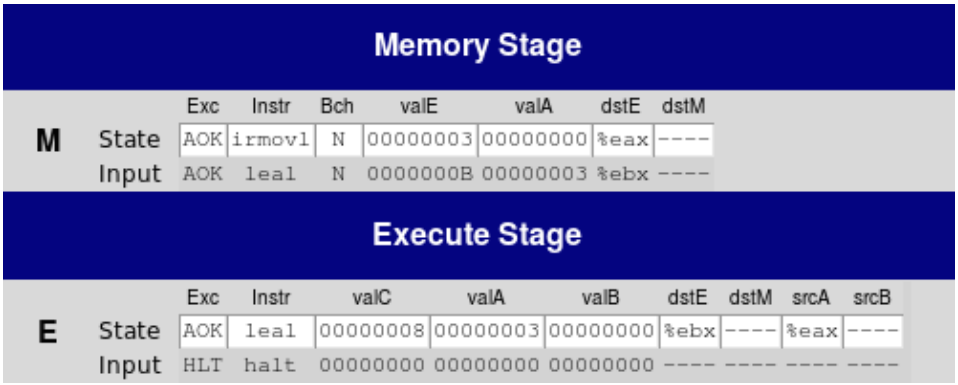

FIGURE 2.4 – Execute

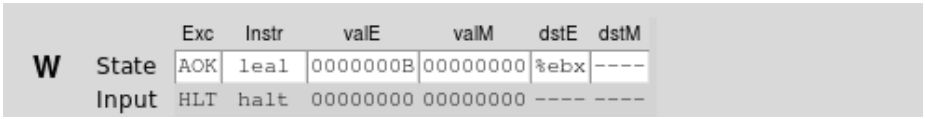

FIGURE 2.5 – Write Back

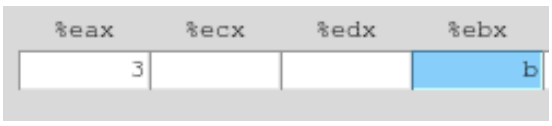

FIGURE 2.6 – Résultat

Chapitre 3

Exercice 2