

Rapport de projet : Architecture des ordinateurs

Benjamin Saint-Sever, Steven Ratton

27 avril 2014

Table des matières

1	Introduction	2
2	Exercice 1 : Ajout de l'instruction leal	3
2.1	implémentation de leal	3
2.2	Architecture séquentielle	4
2.3	Architecture pipelines	7
3	Exercice 2 : Mise en œuvre de stratégies de prédiction sta- tique	9
4	Exercice 3 : Exécution d'instructions sur plusieurs cycles	11
4.1	Question1 :	11
5	Conclusion	15

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 : Ajout de l'instruction leal

2.1 implémentation de leal

Description

Le but de ce premier exercice est de créer une nouvelle instruction "leal" pour "load effective address", cette dernière 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 équivalent 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).

Première étape, 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
```

2.2 Architecture séquentielle**Code HCL :**

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

```
intsig LEAL      'I_IRMOVL'
```

Etape FETCH :

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

Etape Decode :

le contenu 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 contenu 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 permet de 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;
];

```

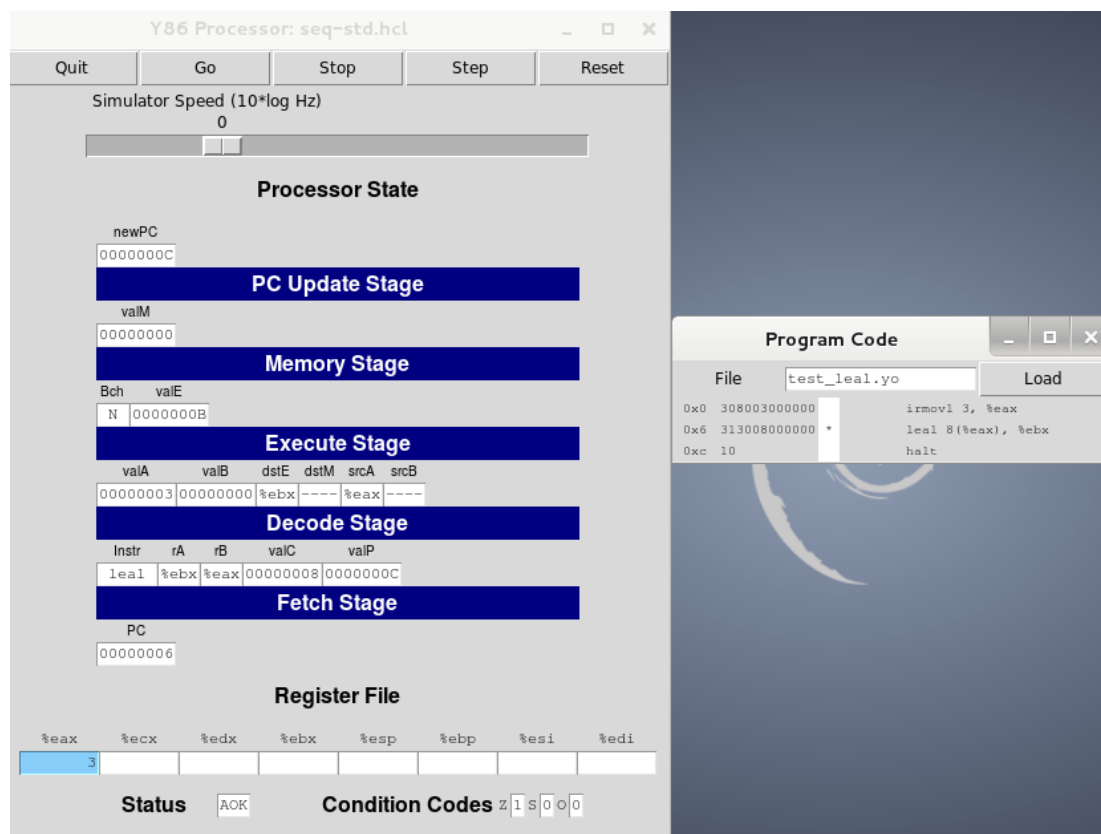


FIGURE 2.1 – Fetch

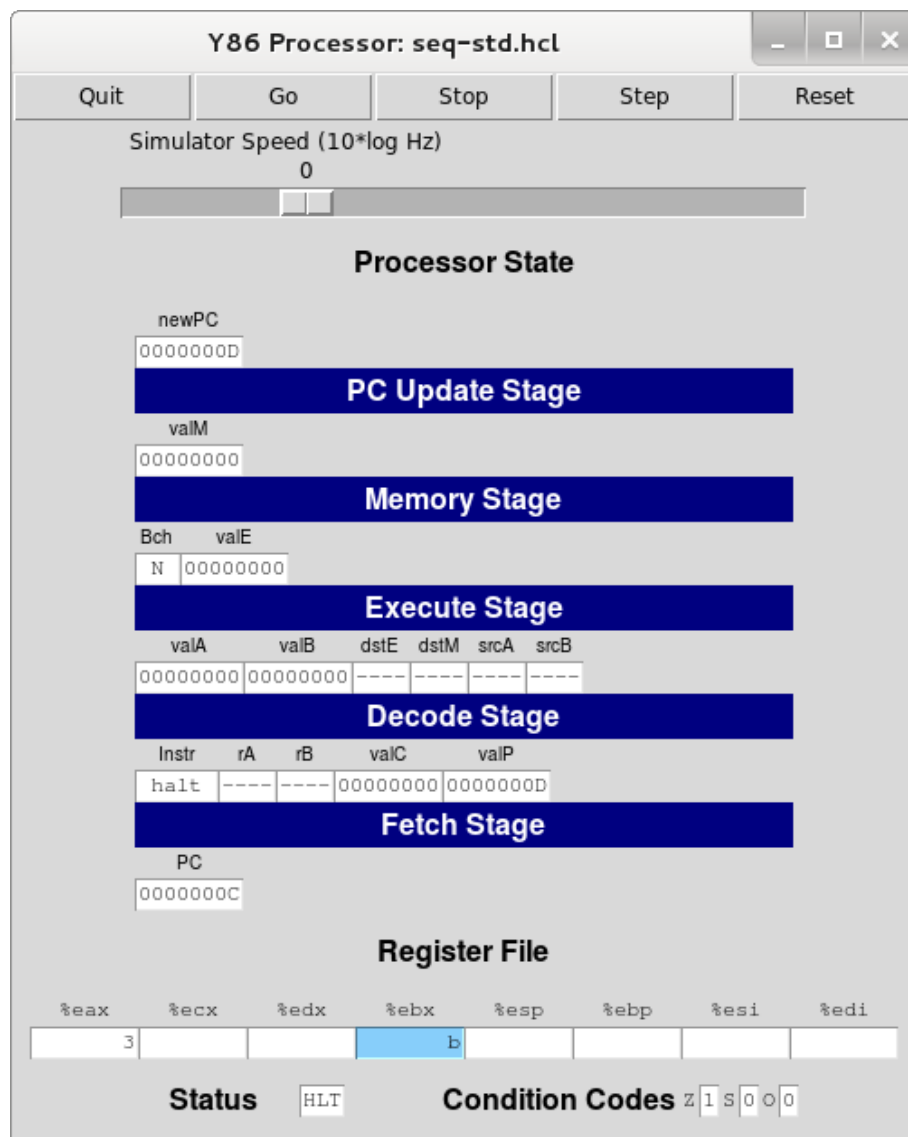


FIGURE 2.2 – Decode

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 :

Même 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

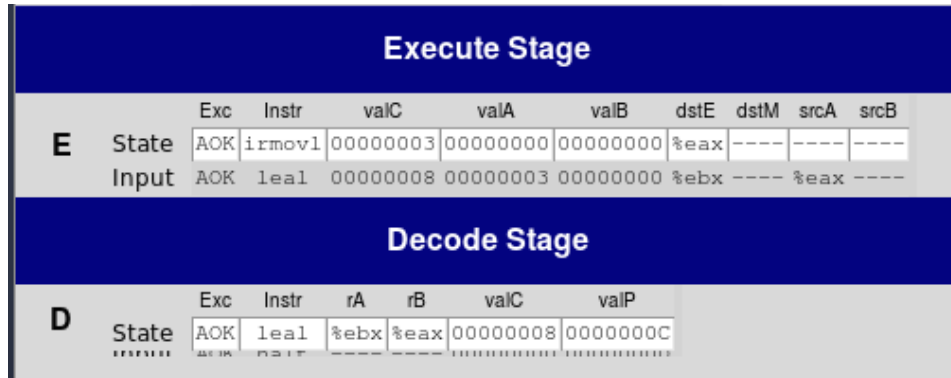


FIGURE 2.3 – Decode

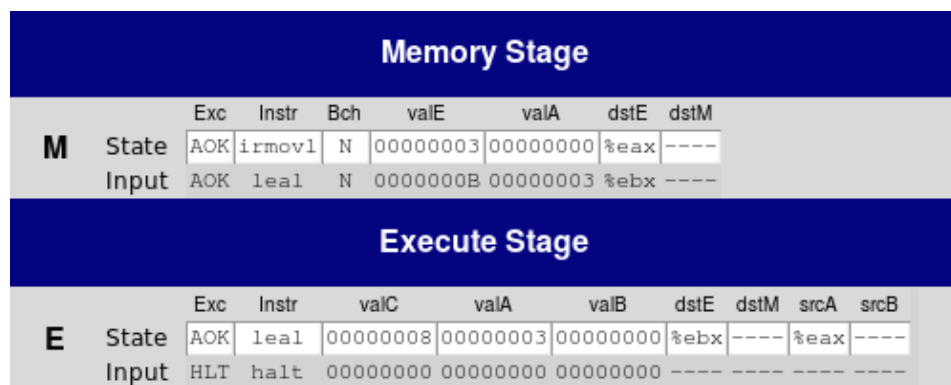


FIGURE 2.4 – Execute

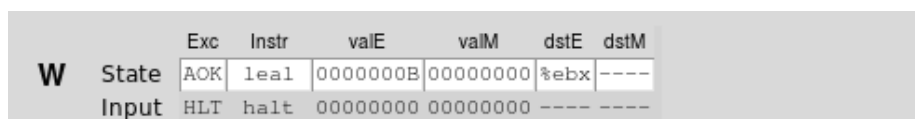


FIGURE 2.5 – Write Back

%eax	%ecx	%edx	%ebx
3			b

FIGURE 2.6 – Résultat

Chapitre 3

Exercice 2 : Mise en œuvre de stratégies de prédiction statique

Gestion des versions :

- nouveau fichier "pipe-maversion1.hcl"
- changer la variable VERSION dans le fichier Makefile

Analyse du CPI :

- ajout des fichiers test loop.hs et cpi.hs
- make loop.hs et make cpi.hs
- ./psim -g loop.hs et ./psim -g cpi.hs

La prédiction de branchement est une fonctionnalité d'un processeur qui lui permet de prédire le résultat d'un branchement. Cette technique permet à un processeur de rendre l'utilisation de son pipeline plus efficace. Avec cette technique, le processeur va faire de l'exécution spéculative : il va parier sur le résultat d'un branchement, et va poursuivre l'exécution du programme avec le résultat du pari. Si le pari échoue, les instructions chargées par erreur dans le pipeline sont annulées.

Question 1 : Expliquer pourquoi on perd du temps avec la prédiction de branchement actuelle de type « toujours ».

La prédiction de branchement de type "toujours" suppose que les branchements conditionnels vont toujours être pris, or ce n'est pas le cas de tous les programmes, lorsqu'une ou plusieurs conditions imbriquées dans une boucle ne sont vraies que rarement, il y a forcément une perte de temps. Le prédicteur de branchement prédit que les conditions sont vraies or elles ne le sont pas souvent. Exemple avec CPI.hs.

Performance Cycles Instructions CPI

Dans le cas " toujours " :

- valC utilisé comme adresse de la prochaine instruction.
- valP conservé le long du pipe-line pour devenir l'adresse de la prochaine instruction si la prédiction s'avère fausse (valP est multiplexé en tant que valA dès l'étage E).

il se pourra que ce soit valP qui serve d'adresse de la prochaine instruction, et il faudra alors préserver valC. Vous aurez donc besoin de propager valC également jusqu'à l'étage M, en plus de valP/valA. La faire cheminer dans l'UAL peut être un bon moyen pour cela, puisque celle-ci ne sert pas pour les branchements.

Prédiction de branchement " jamais " : Permet d'obtenir de meilleure performance, au fait que la plus part du temps les branchement conditionnels ne sont pas pris et les branchements inconditionnels le sont.

- valP utilisé comme adresse de la prochaine instruction.
- valC conservé.
- propager valC également jusqu'à l'étage M, en plus de valP/valA.

ajout : `intsig JUNCOND 'J_YES'`

on utilise alors `"(M_icode == JXX) && (M_ifun!= JUNCOND)"`
pour tester les branchements conditionnels

Modification des lignes 123, 145,146,213

Chapitre 4

Exercice 3 : Exécution d'instructions sur plusieurs cycles

4.1 Question1 :

On remplace dans misc, isa.h et misc, isa.c IPOP2 par ILSM. afin que lods corresponde à l'opcode ILSM avec ifun=O , et stos au même opcode, avec ifun=1 et supprimer la définition de pop2

Ajout dans pipe-exercice3.hcl :

```
- Declarations :
    intsig LSM          'I_LSM'

- Fetch Stage :
bool instr_valid = f_icode in
    { NOP, HALT, RRMovL, IRMOVL, RMMOVL, MRMOVL,
      OPL, IOPL, JXX, CALL, RET, PUSHL, POPL, LSM };

int instr_next_ifun = [
    f_icode == LSM && f_ifun == 2 : 0;
    f_icode == LSM && f_ifun == 0 : 1;
    1 : -1;
];

- Decode Stage :

int new_E_srcA = [
    D_icode == LSM && D_ifun == 1 : EAX;
    # EAX : écriture mémoire dans EDI => srcA
    D_icode == LSM && D_ifun == 0 : ESI;
```

```

        D_icode in { RRMovL, RMMovL, OPL, PUSHl } : D_rA;
        D_icode in { POPL, RET } : RESP;
        1 : RNONE; # Don't need register
];

int new_E_srcB = [
    D_icode == LSM && D_ifun == 1 : EDI;
    D_icode in { OPL, IOPL, RMMovL, MRMOVL } : D_rB;
    D_icode in { PUSHl, POPL, CALL, RET } : RESP;
    1 : RNONE; # Don't need register
];

int new_E_dstE = [
    D_icode == LSM && D_ifun == 0 : ESI;
    # ESI est incrémenté dans ALU puis réaffecté à lui-même
    D_icode == LSM && D_ifun == 1 : EDI;

    D_icode in { RRMovL, IRMOVL, OPL, IOPL } : D_rB;
    D_icode in { PUSHl, POPL, CALL, RET } : RESP;
    1 : DNONE; # Don't need register DNONE, not RNONE
];

int new_E_dstM = [
    D_icode in { MRMOVL, POPL } : D_rA;

    D_icode == LSM && D_ifun == 0 : EAX;
    # EAX prend la valeur lue en mémoire dans ESI

    1 : DNONE; # Don't need register DNONE, not RNONE
];

```

– Execute Stage:

```
int aluA = [
    E_icode == LSM && E_ifun == 0 : 4;
    E_icode == LSM && E_ifun == 1 : 4;
    E_icode in { RRMovL, OPL } : E_valA;
    E_icode in { IRMOVL, RMMovL, MRMOVL, IOPL } : E_valC;
    E_icode in { CALL, PUSHl } : -4;
    E_icode in { RET, POPL } : 4;
    # Other instructions don't need ALU
];
```

```
int aluB = [
    E_icode == LSM && E_ifun == 0 : E_valA;
    # Incrémentation de ESI
    E_icode == LSM && E_ifun == 1 : E_valB;
    # On décrémente de 4 le pointeur EDI

    E_icode in { RMMovL, MRMOVL, OPL, IOPL, CALL,
                PUSHl, RET, POPL } : E_valB;
    E_icode in { RRMovL, IRMOVL } : 0;
    # Other instructions don't need ALU
];
```

– Memory Stage:

```
bool mem_read = M_icode in { MRMOVL, POPL, RET } ||
(M_icode == LSM && M_ifun == 0);

bool mem_write = M_icode in { RMMovL, PUSHl, CALL } ||
(M_icode == LSM && M_ifun == 1);
```

```
int mem_addr = [
    #injection de esi dans eax
    M_icode == LSM && M_ifun == 0 : M_valA;

    M_icode in { RMMovL, PUSHl, CALL, MRMOVL } : M_valE;
    M_icode == LSM && M_ifun == 1 : M_valE;
    M_icode in { POPL, RET } : M_valA;
    # Other instructions don't need address
];
```

Dans pipe/psim.c :

- Ajout dans stage implementations :

```
int gen_instr_next_ifun();
```

- Ajout de conditions :

```
/* Ready to fetch instruction.  
Speculatively fetch register byte  
and immediate word  
*/
```

```
if (gen_instr_next_ifun() != -1)  
    if_id_next->ifun = gen_instr_next_ifun();  
else {...}
```

```
fetch_ok = TRUE;
```

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

Chapitre 5

Conclusion

Nous avons utilisé github qui est un logiciel de gestion de version qui nous a permis de travailler indépendamment mais aussi en collaboration. Nous n'avons pas terminé le projet principalement par manque de temps du à d'autres projet en cour.