



RAPPORT PROJET

AJOUT D'INSTRUCTIONS ET PREDICTEURS STATIQUES

Sommaire

Sommaire	3
Introduction	4
Exercice 1 : Ajout de l'instruction LEAL	5
Exercice 2 : Mise en œuvre de stratégies de prédiction statique	9
Exercice 3 : Exécution d'instructions sur plusieurs cycles	16
Conclusion	18

Introduction

SUJET DU PROJET

L'objectif de ce projet est à la fois d'étendre le jeu d'instructions du processeur y86 et d'optimiser son exécution. Le projet est en trois étapes, qui sont en fait indépendantes. En premier lieu, il s'agira de mettre en œuvre une nouvelle instruction. Ensuite, on s'intéressera au fonctionnement de la prédiction de branchement, en mettant en œuvre deux types de prédiction statique, de type « jamais » puis « backward taken, forward not taken ». En troisième lieu, on mettra en place l'exécution de certaines instructions sur plusieurs cycles, permettant au y86 d'exécuter des instructions plus complexes que ce qu'il était possible jusqu'ici.

DECOMPOSITION DU TRAVAIL

Pour réaliser ce projet, nous avons divisé le travail de cette façon : l'exercice 1 a été réalisé par Manon, l'exercice 2 par Caitlin et l'exercice 3 par nous deux ainsi que le rapport.

OUTILS

Pour pouvoir interagir sur le projet, nous avons décidé d'utiliser github (dépôt en ligne) afin de pouvoir accéder sans difficultés à tous nos fichiers n'importe quand et de n'importe où.

PROJET

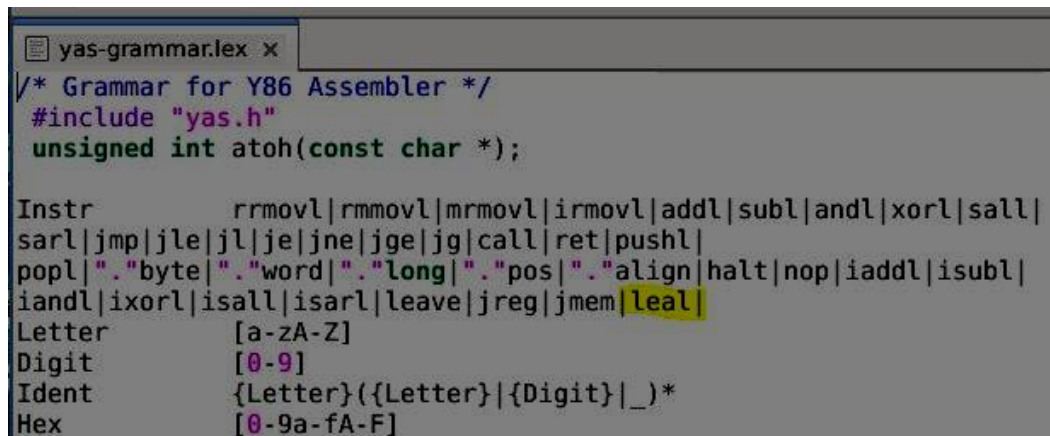
Exercice 1 : Ajout de l'instruction LEAL

QU'EST-CE QUE LEAL ?

Leal, pour « load effective address (long) » est une instruction qui permet d'effectuer deux additions (et une pseudo-multiplication) d'un seul coup.

MODIFICATION DE DEUX FICHIERS : YAS-GRAMMAR.LEX ET ISA.C

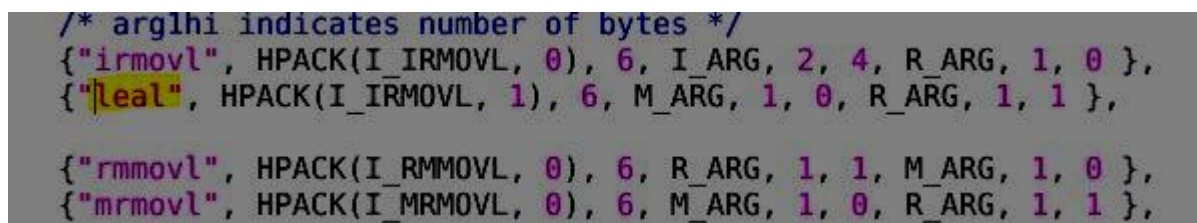
Premièrement, nous avons ajouté leal comme instruction au fichier yas-grammar.lex.



```
yas-grammar.lex x
/* Grammar for Y86 Assembler */
#include "yas.h"
unsigned int atoi(const char *);

Instr      rrmovl|rrmovl|mrmovl|irmovl|addl|subl|andl|xorl|sall|
sar|jmp|jle|jl|je|jne|jge|jg|call|ret|pushl|
popl|".byte|".word|".long|".pos|".align|halt|nop|iaddl|isubl|
iandl|ixorl|isall|isar|leave|jreg|jmem|leal|
Letter     [a-zA-Z]
Digit      [0-9]
Ident      {Letter}({Letter}|{Digit}|_)*
Hex        [0-9a-fA-F]
```

Ensuite nous avons modifié le fichier isa.c, afin que notre instruction soit prise en compte.



```
/* arglhi indicates number of bytes */
{"irmovl", HPACK(I_IRMOVL, 0), 6, I_ARG, 2, 4, R_ARG, 1, 0},
{"leal", HPACK(I_IRMOVL, 1), 6, M_ARG, 1, 0, R_ARG, 1, 1},

{"rrmovl", HPACK(I_RMMOVL, 0), 6, R_ARG, 1, 1, M_ARG, 1, 0},
{"mrmovl", HPACK(I_MRMOVL, 0), 6, M_ARG, 1, 0, R_ARG, 1, 1},
```

PROJET

FICHER TEST

Afin de pouvoir tester la nouvelle instruction leal, nous avons fait un programme de test : leal.ys

```
.pos 0x00
    irmovl 0,%eax
    irmovl 1,%ebx
    leal 4(%ebx),%eax
    halt
```

Au début de notre test, eax vaut 0 et ebx 1. A la fin, eax vaut 5 car leal permet de faire `rrmovl %ebx, %eax` et `iaddl 4, %eax` (ici).

MODIFICATION DE SEQ-STD.HCL & PIPE-STD.HCL

Leal utilisant le même opcode que `irmovl`, nous avons dû modifier la version séquentielle et pipe-linée afin de faire la distinction à l'exécution.

Fichier `seq-std.hcl` :

```
## What register should be used as the B source?
int srcB = [
    icode in { OPL, IOPL, RMMOVL, MRMOVL, JMEM } : rB;
    icode == IRMOVL && ifun == 1 : rB;
    icode in { PUSH, POPL, CALL, RET } : RESP;
    icode in { LEAVE } : REBP;
    1 : RNONE; # Don't need register
];

## What register should be used as the E destination?
int dstE = [
    icode in { RRMOVL, OPL, IOPL } : rB;
    icode == IRMOVL && ifun == 0 : rB;
    icode == IRMOVL && ifun == 1 : rA;
    icode in { PUSH, POPL, CALL, RET, LEAVE } : RESP;
    1 : RNONE; # Don't need register
];
```

PROJET

Ici nous sommes à l'étage décode : on a rajouté `icode == IRMOVL && ifun == 1 : rb` ; pour dire que quand on fait leal (`irmovl` avec `ifun` à 1) on choisit `rb` comme source. Et on fait la distinction pour la destination : `ifun` à 0 (donc `irmovl`) on choisit `ra` et `rb` quand on fait leal.

```
## Select input B to ALU
int aluB = [
    icode == IRMOVL && ifun == 1 : valB;
    icode == IRMOVL && ifun == 0 : 0;
    icode in { RMMOVL, MRMOVL, OPL, IOPL, CALL, PUSH, RET,
    POPL, JMEM, LEAVE } : valB;
    icode in { RRMOVL } : 0;
    # Other instructions don't need ALU
];
```

Maintenant nous sommes dans l'exécute. Pour l'ALU on fait la distinction également quand on fait `irmovl` (`ifun == 0`) on choisit 0 mais quand on fait leal (`ifun == 1`) on choisit `valB`.

Fichier `pipe-std.hcl` :

```
## What register should be used as the B source?
int new_E_srcB = [
    D_icode in { OPL, IOPL, RMMOVL, MRMOVL, JMEM } : D_rB;
    D_icode == IRMOVL && D_ifun == 1 : D_rB;
    D_icode in { PUSH, POPL, CALL, RET } : RESP;
    D_icode in { LEAVE } : REBP;
    1 : RNONE; # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
    D_icode == IRMOVL && D_ifun == 1 : D_rA;
    D_icode == IRMOVL && D_ifun == 0 : D_rB;
    D_icode in { RRMOVL, OPL, IOPL } : D_rB;
    D_icode in { PUSH, POPL, CALL, RET, LEAVE } : RESP;
    1 : DNONE; # Don't need register DNONE, not RNONE
];
```

Ici nous sommes à l'étage décode : on a rajouté `D_icode == IRMOVL && ifun == 1 : D_rB` ; pour dire que quand on fait leal (`irmovl` avec `ifun` à 1) on choisit `rb` comme source. Et on fait la distinction pour la destination : `ifun` à 0 (donc `irmovl`) on choisit `ra` et `rb` quand on fait leal.

PROJET

```
## Select input B to ALU
int aluB = [
    E_icode == IRMOVL && E_ifun == 1 : E_valB;
    E_icode == IRMOVL && E_ifun == 0 : 0;
    E_icode in { RMMOVL, MRMOVL, OPL, IOPL, CALL,
                PUSHL, RET, POPL, JMEM, LEAVE} : E_valB;
    E_icode in { RRM0VL} : 0;
    # Other instructions don't need ALU
];
```

Maintenant nous sommes dans l'exécute. Pour l'ALU on fait la distinction également quand on fait `irmovl` (`ifun == 0`) on choisit 0 mais quand on fait `leal` (`ifun == 1`) on choisit `valB`.

Exercice 2 : Mise en œuvre de stratégies de prédiction statique

QU'EST-CE QU'UN PREDICTION DE BRANCHEMENT ?

Une prédiction de branchement sert à améliorer l'efficacité du processeur pipeliné : il n'aura alors pas besoin d'attendre que le test de branchement arrive au stade « exécute », et peut donc gagner du temps. Ceci peut aussi faire perdre du temps lorsque la prédiction de branchement est incorrecte.

CAS D'UNE PREDICTION DE BRANCHEMENT INCORRECT

Lorsqu'une prédiction de branchement est incorrecte, alors il faut arrêter les calculs en cours et vider le pipeline et retourner calculer la valeur de branchement correct.

METHODES DE PREDICTION

Il y a plusieurs méthodes de prédire l'adresse de branchement. On peut utiliser une structure de donnée qui stocke les résultats des tests de branchement précédents (ex : 00 = branchement jamais pris, 01 = branchement parfois pris, 10 = branchement souvent pris, 11 = branchement toujours pris). Ceci est une méthode de prédiction de branchement dynamique.

On peut aussi utiliser des méthodes bien plus simples, les méthodes statiques : le simulateur y86 étudié en cours utilise une prédiction de branchement de type « toujours » : c'est à dire qu'il prédit que le test de branchement sera toujours vrai. Dans cet exercice nous sommes demandés de modifier le simulateur pour utiliser une prédiction de branchement de type « jamais », qui prédira que le test de branchement sera toujours faux.

QUESTION 1

Les branchements conditionnels de type « toujours » perdent toujours du temps car son adresse de branchement doit être calculée, ce qui n'est pas le cas pour les branchements conditionnels de type « jamais ».

PROJET

```
## What address should instruction be fetched at
int f_pc = [
    # Mispredicted branch. Fetch at incremented PC
    M_icode == IJXX && !M_Bch && M_ifun != JUNCOND : M_valA;
    # Completion of RET instruction.
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];
```

IMPLEMENTATION DE PREDICTEUR DE BRANCHEMENT DE TYPE « JAMAIS »

On travaille dans le fichier pipe-nt.hcl.

On ajoute un variable 'JUNCOND' pour pouvoir distinguer les branchements conditionnels (je, jne, ...) des branchements non-conditionnels (jmp). Ainsi le branchement « jamais » n'affectera pas jmp.

Ex :

On doit modifier la prédiction de la nouvelle valeur de PC . On distingue 4 cas (l'un est redondant, mais c'est plus clair de l'inclure comme 4e cas) :

- instruction = call : dans ce cas, la nouvelle valeur de PC sera l'adresse donné au moment de l'écriture du code ys , comme avant
- instruction = jmp : on a séparé ce cas de 'call' car il faut ajouter une condition pour vérifier si c'est un branchement conditionnel ou pas. Dans ce cas PC sera égal à l'adresse donné aussi.
- instruction = branchement conditionnel : dans ce cas, on prendra la valeur incrémentée de PC au lieu de l'adresse donné
- tous les autres cas : on prendra la valeur incrémentée de PC, comme d'habitude.

```
# Predict next value of PC
int new_F_predPC = [
    # BNT: This is where you'll change the branch prediction rule
    f_icode in {ICALL } : f_valC; #on sépare de l'instruction suivant car on a ajouté la
condition f_ifun == JUNCOND
    # si unconditionnel (jmp):
    f_icode in { IJXX } && f_ifun == JUNCOND : f_valC;
    # si conditionnel (je, jg, ...) ((nb: cette ligne est redondant, avec la dernière ligne qui
prend tous les autres cas, y compris les branchements conditionnels, mais c'est plus clair ainsi))
    f_icode in { IJXX } && f_ifun != JUNCOND: f_valP;
    # et dans tous les autres cas
    1 : f_valP;
];
```

PROJET

Dans les cas de mauvaises prédictions de branchement, il faut connaître l'adresse correcte, qui avait été donné lors de l'écriture de code. Ceci n'est pas un problème avec la prédiction « toujours » car c'est la PC de l'instruction qui sera incrémentée et utilisé. Or, avec la prédiction « jamais », on a déjà utilisé ce PC. Il faut trouver une façon de préserver l'adresse donné. En s'inspirant des instructions « rrmovl » et « irmovl », on a décidé d'utiliser l'UAL, en ajoutant une valeur de « 0 » pour ne pas modifier l'adresse. Ainsi lors d'une mauvaise prédiction de branchement, on aura facilement accès à l'adresse correcte.

```
## Select input A to ALU
int aluA = [
    E_icode in { IRRMOVL, IOPL } : E_valA;
    E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
    E_icode in { ICALL, IPUSHL } : -4;
    E_icode in { IRET, IPOPL } : 4;
    # comme suggéré dans l'énoncé, on utilise l'ALU pour faire propager valC jusqu'à l'étage M,
    où il sera utilisé en cas de mauvais prédiction de branchement
    E_icode in { IJXX } : E_valC;
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                IPUSHL, IRET, IPOPL } : E_valB;
    E_icode in { IRRMOVL, IIRMOVL } : 0;
    # pour ne pas modifier l'adresse de branchement
    E_icode in { IJXX } : 0;
    # Other instructions don't need ALU
];
```

Dans le contrôle de pipeline, il faut changer le test de mauvais branchement. Avant, dans le cas de prédiction de branchement « toujours », la prédiction était fausse lorsque la condition était fausse. Or dans ce cas de prédiction « jamais », la prédiction sera fausse lorsque la condition est vraie.

PROJET

```
bool D_bubble =
    # Mispredicted branch
    # un prédiction de branchement sera jugé mauvais si son résultat est vrai,
    # il faut donc changer la condition ici
    (E_icode == IJXX && E_ifun != JUNCOND && e_Bch) ||
    # Stalling at fetch while ret passes through pipeline
    # but not condition for a load/use hazard
    !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
    IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
    # Mispredicted branch
    # de même, on change la condition ici aussi
    (E_icode == IJXX && E_ifun != JUNCOND && e_Bch) ||
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
    E_dstM in { d_srcA, d_srcB};
```

QUESTION 2

Le prédicteur « jamais » est en général plus efficace (mais pas forcément plus précis) que le prédicteur « toujours », car il n'a pas à chercher l'adresse auquel se brancher. Lorsque la prédiction est correcte, alors il n'y a aucun coût - le processeur continue comme s'il n'y avait aucun branchement. La méthode de prédiction de branchement « toujours » fonctionne mieux en cas de boucles.

PROJET

IMPLEMENTATION DE PREDICTEUR DE BRANCHEMENT DE TYPE « BTFNT »

On travaille ici dans le fichier pipe-btfnt.hcl.

Le prédicteur « BTFNT » (backwards taken, forwards not taken) est un prédicteur statique qui prend le branchement si l'adresse donné se situe « avant » l'adresse actuel, et ne prend pas le branchement si l'adresse donné est situé « après » l'adresse actuel. Il est basé sur le principe des boucles : il suppose qu'un branche « backwards » fait partie d'un boucle, et sera donc pris plus souvent qu'il ne l'est pas, et qu'un branchement « forwards » sera plus souvent un « jmp » et non un boucle.

Lors d'un branchement conditionnel, si l'adresse donnée est avant l'adresse courante alors on prend l'adresse donné (valC). Sinon, on prend la valeur incrementée de PC (valP) :

```
# Predict next value of PC
int new_F_predPC = [
    # BBTfNT: This is where you'll change the branch prediction rule
    f_icode in { ICALL } : f_valC;
    # cas branchement inconditionnel
    f_icode in { IJXX } && f_ifun == JUNCOND: f_valC;
    # cas branchement backwards
    f_icode == IJXX && f_valP > f_valC: f_valC
    # cas branchement forwards & tous les autres
    1 : f_valP;
];
```

PROJET

Contrairement aux branchements « jamais » et « toujours », lorsqu'un branchement est incorrect il y a deux valeurs possibles pour la nouvelle valeur de PC :

```
## What address should instruction be fetched at
int f_pc = [
    # Mispredicted branch. Fetch at incremented PC
    # cas où le backwards branch a été prise et est incorrect, on cherche
    donc la valeur incremented du PC original
    M_icode == IJXX && M_ifun != JUNCOND && M_valA > M_valB && !M_Bch : M_valA
    # cas où le forwards branch a été prise et est incorrect, on cherche donc
    l'adresse donnée (valC, qui est passé par l'UAL pour devenir M_valE
    M_icode == IJXX && M_Bch && M_ifun != JUNCOND && M_valA <= M_valE :
    M_valE;
    # Completion of RET instruction.
    W_icode == IRET : W_valM;
    # Default: Use predicted value of PC
    1 : F_predPC;
];
```

Comme vous pouvez voir, valC est devenu valE, car cette valeur est passée dans l'UAL pour être disponible en cas de mauvais branchement (comme pour le branchement « jamais »)

```
## Select input A to ALU
int aluA = [
    E_icode in { IRRMOVL, IOPL } : E_valA;
    E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
    E_icode in { ICALL, IPUSHL } : -4;
    E_icode in { IRET, IPOPL } : 4;
    # comme suggéré dans l'énoncé, on utilise l'ALU pour faire propager valC jusqu'à l'étage M,
    où il sera utilisé en cas de mauvais prédiction de branchement
    E_icode in { IJXX } : E_valC;
    # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
    E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                IPUSHL, IRET, IPOPL } : E_valB;
    E_icode in { IRRMOVL, IIRMOVL } : 0;
    # pour ne pas modifier l'adresse de branchement
    E_icode in { IJXX } : 0;
    # Other instructions don't need ALU
];
```


PROJET

Enfin, lors du contrôle de pipeline, les tests utilisés changent car, comme mentionné ci-dessus, il y a plusieurs cas de mauvais branchement :

```
bool D_bubble =
    # Mispredicted branch
    # le branchement étant considéré correct sous d'autres conditions
    qu'avant, on change les tests suivants
    (E_icode == IJXX && E_ifun != JUNCOND &&
     (E_valA > E_valA && !e_Bch || E_valA <= E_valC && e_Bch)) ||
    # BBTFNT: This condition will change
    # Stalling at fetch while ret passes through pipeline
    # but not condition for a load/use hazard
    !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
    IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
    # Mispredicted branch
    # on change le test ici aussi
    (E_icode == IJXX && E_ifun != JUNCOND &&
     (E_valA > E_valA && !e_Bch || E_valA <= E_valC && e_Bch)) ||
    # BBTFNT: This condition will change
    # Conditions for a load/use hazard
    E_icode in { IMRMOVL, IPOPL } &&
    E_dstM in { d_srcA, d_srcB};
```

QUESTION 3

Les branchements conditionnels de type « BTfNT » fonctionnent efficacement lorsque la plupart des branchements « backwards » correspondent à des boucles et les branchements « forwards » correspondent à des « jmp ».

Pour illustrer ceci, on a créé deux fichiers tests : test-btfnt-good.js, où le branchement fonctionnera efficacement, et test-btfnt-bad.js, où le branchement fonctionnera inefficacement.

Exercice 3 : Exécution d'instructions sur plusieurs cycles

LES INSTRUCTIONS LODS/STOS/MOVS

Les instructions lods, stos et movs n'ont pas de paramètre. Elles utilisent implicitement esi et edi comme adresse source et comme adresse destination, et eax comme valeur. Lods lit en mémoire à l'adresse esi, stocke le résultat dans eax, et ajoute 4 au pointeur esi. Stos écrit en mémoire à l'adresse edi le contenu de eax, et ajoute 4 au pointeur edi. Movs lit en mémoire à l'adresse esi, écrit la valeur en mémoire à l'adresse edi, et ajoute 4 à esi et edi.

Question 1 :

Afin de prendre en compte nos nouvelles instructions, nous avons modifié les fichiers isa.h, isa.c et psim.c.

Voici les modifications qui ont été apportés :

Isa.h

```
typedef enum { I_NOP, I_HALT, I_RRMOVL, I_IRMOVL, I_RMMOVL, I_MRMOVL,  
               I_ALU, I_JXX, I_CALL, I_RET, I_PUSHL, I_POPL,  
               I_ALUI, I_LEAVE, I_JREG, I_JMEM, I_LSM } itype_t;
```

Nous avons enlevé l'instruction I_POP2 et nous avons ajouté I_LSM afin que nos nouvelles instructions soit prises en comptes.

Isa.c

```
{ "lods",  HPACK(I_LSM, 0) , 0, NO_ARG, 0, 0, NO_ARG, 0, 0 },  
{ "stos",  HPACK(I_LSM, 1) , 0, NO_ARG, 0, 0, NO_ARG, 0, 0 },
```

Ici nous avons simplement supprimé l'instruction pop2 et nous avons créé les instructions Lods et Stos en reprenant l'opcode de pop2 et en changeant ifun qui est à 1 pour lods.

PROJET

Psim.c

```
/* Performance monitoring */  
if (mem_wb_curr->exception != EXC_BUBBLE && mem_wb_curr->icode != I_LSM) {  
    starting_up = 0;  
    instructions++;  
    cycles++;  
} else {  
    if (!starting_up)  
        cycles++;  
}  
  
sim_report();  
return mem_wb_curr->exception;
```

Ici nous avons juste remplacé I_POP2 par I_LSM afin de prendre en compte nos 3 instructions.

Nous avons également ajouté les deux instructions aux fichiers pipe-std.hcl et yas-grammar.lex de la même façon que dans l'exercice 1.

Par la suite, nous avons créé un programme de test afin de vérifier si l'assembleur génère bien les bons codes machine, et que ce programme s'exécute comme prévu. Voici notre programme :

```
.pos 0x00  
    irmovl 1, %esi  
    lods  
    stos  
    halt
```

SUPPORT DES INSTRUCTIONS SUR PLUSIEURS CYCLES

A finir.

Conclusion

Nous avons rencontré quelques problèmes lors de ce projet (erreur de syntaxe, fonction qui ne réalisait pas ce que l'on voulait ...) mais nous avons réussi à surmonter ces problèmes. Actuellement le projet n'ai pas fini (nous n'avons pas terminé l'exercice 3).

Ce projet nous a permis de mieux comprendre le langage y86 et les structures seq et pipe.