Université de Bordeaux
Licence STS

université de BORDEAUX

ARCHITECTURE DES ORDINATEURS

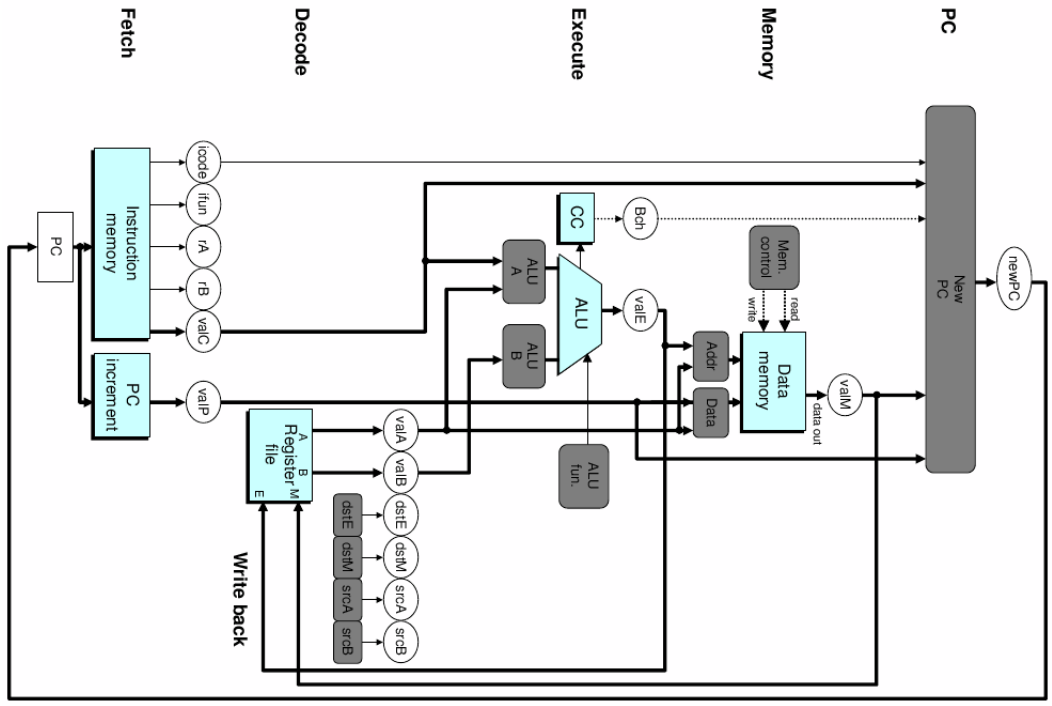
TD : **10**

Étude détaillée de l'architecture Y86(seq)

Introduction

Le jeu d'instructions Y86 peut donner lieu à différentes implémentations, plus ou moins efficaces en termes de performances (mesurées en nombre d'instructions exécutées par cycle).

Sa mise en œuvre la plus simple est la version « `seq` » (pour « *séquentielle* »), dans laquelle on exécute exactement une instruction par cycle. Le code HCL de l'architecture `seq` est fourni à la fin de cette feuille. Son chemin de données est représenté ci-dessous.



Exercice 1 : Codage des instructions Y86

On considère le fragment de code suivant, en langage d'assemblage et en langage machine Y86 :

```
0x0000:                 |       .pos 0
0x0000: 30f30b000000    |       irmovl  11,%ebx
0x0006: 30f10c000000    |       irmovl  12,%ecx
0x000c: 6031            |       addl    %ebx,%ecx
0x000e: 2010            |       rrmovl  %ecx,%eax
0x0010: 6003            |       addl    %eax,%ebx
0x0012: 6131            |       subl    %ebx,%ecx
0x0014: 00              |       halt
```

Question 1

En analysant le code ci-dessus, déterminez :
— le format des différentes instructions ;
— le code des registres utilisés ;
— le code de registre utilisé dans le cas où un registre n'est pas nécessaire.

Exercice 2 : Câblage des instructions *op*`l`

Les instructions `addl`, `subl`, etc. appartiennent à la catégorie des opérations arithmétiques et logiques.

Analysez le code HCL fourni en annexe pour déterminer comment s'exécute une instruction *op*`l` (telle que `subl`) au sein de l'architecture Y86(`seq`).

Question 1

Dans le cas de l'instruction `OPL`, mais plus généralement de toutes les instructions, quel est le sens des signaux calculés à l'étage *Fetch* ? Comment peut-on calculer `valP` à partir de ceux-ci ?

Question 2

Quelles sont les valeurs des signaux positionnés pour `OPL` pour les autres étages ?

Question 3

Quel est le nom symbolique correspondant à l'absence de registre ?

Exercice 3 : Ajout des instructions `i`*op*`l`

Le langage machine Y86 original ne possède pas d'instruction « `i`*op*`l` », c'est-à-dire d'instruction arithmétique et logique opérant entre un registre et une valeur immédiate. C'est une source d'inefficacité, car cela oblige, pour effectuer un tel calcul, à mobiliser un registre supplémentaire : celui dans lequel placer la valeur immédiate avant d'effectuer l'opération avec lui. On souhaite donc les rajouter, en touchant le moins possible au câblage existant.

Question 1

Quels seraient la taille et le format de l'instruction `i`*op*`l` ?

Question 2

Comment peut-on mettre en place une telle instruction sans avoir besoin d'un `opcode` supplémentaire ?

Question 3

Quelles sont les modifications à apporter au code HCL afin de câbler l'instruction `i`*op*`l` ?

Exercice 4 : Étude des instructions `pushl` et `popl`

Les instructions `pushl` et `popl` sont des instructions duales : effectuer un `pushl`, suivi d'un `popl` dans le même registre, doit laisser l'ordinateur dans le même état (à part le compteur ordinal, bien sûr).

Analysez le code HCL fourni en annexe pour déterminer comment s'exécutent les instructions `pushl` et `popl` au sein de l'architecture Y86(`seq`).

Question 1

Pourquoi, dans le code HCL, pour obtenir la valeur du signal `mem_addr`, utilise-t-on `valE` pour `pushl`, et `valA` pour `popl` ?

Question 2

Comment serait-il possible de « factoriser » le codage de ces deux instructions pour gagner un *opcode* ?

```
################ Fetch Stage     ###################################
# Does fetched instruction require a regid byte?
bool need_regids =
icode in { RRMOVL, OPL, PUSHL, POPL, IRMOVL, RMMOVL, MRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
icode in { IRMOVL, RMMOVL, MRMOVL, JXX, CALL };

bool instr_valid = icode in
{ NOP, HALT, RRMOVL, IRMOVL, RMMOVL, MRMOVL,
       OPL, JXX, CALL, RET, PUSHL, POPL };

################ Decode Stage    ###################################
## What register should be used as the A source?
int srcA = [
icode in { RRMOVL, RMMOVL, OPL, PUSHL } : rA;
icode in { POPL, RET } : RESP;
1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int srcB = [
icode in { OPL, RMMOVL, MRMOVL } : rB;
icode in { PUSHL, POPL, CALL, RET } : RESP;
1 : RNONE;  # Don't need register
];

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

## What register should be used as the M destination?
int dstM = [
icode in { MRMOVL, POPL } : rA;
1 : RNONE;  # Don't need register
];

################ Execute Stage   ###################################
## Select input A to ALU
int aluA = [
icode in { RRMOVL, OPL } : valA;
icode in { IRMOVL, RMMOVL, MRMOVL } : valC;
icode in { CALL, PUSHL } : -4;
icode in { RET, POPL } : 4;
# Other instructions don't need ALU
];

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

```
## Set the ALU function
int alufun = [
icode in { OPL } : ifun;
1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = (icode == OPL);

################ Memory Stage    ###################################
## Set read control signal
bool mem_read = icode in { MRMOVL, POPL, RET };

## Set write control signal
bool mem_write = icode in { RMMOVL, PUSHL, CALL };

## Select memory address
int mem_addr = [
icode in { RMMOVL, PUSHL, CALL, MRMOVL } : valE;
icode in { POPL, RET } : valA;
# Other instructions don't need address
];

## Select memory input data
int mem_data = [
# Value from register
icode in { RMMOVL, PUSHL } : valA;
# Return PC
icode == CALL : valP;
# Default: Don't write anything
];

################ Program Counter Update ############################
## What address should instruction be fetched at

int new_pc = [
# Call.  Use instruction constant
icode == CALL : valC;
# Taken branch.  Use instruction constant
icode == JXX && Bch : valC;
# Completion of RET instruction.  Use value from stack
icode == RET : valM;
# Default: Use incremented PC
1 : valP;
];
```