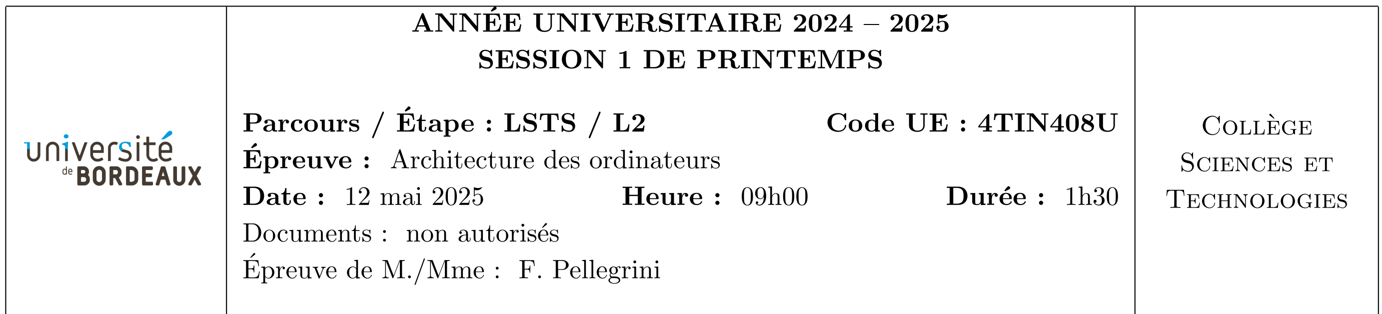


université de BORDEAUX

ANNÉE UNIVERSITAIRE 2024 – 2025
SESSION 1 DE PRINTEMPS

Parcours / Étape : LSTS / L2 **Code UE : 4TIN408U**
Épreuve : Architecture des ordinateurs
Date : 12 mai 2025 **Heure :** 09h00 **Durée :** 1h30
Documents : non autorisés
Épreuve de M./Mme : F. Pellegrini

Collège Sciences et Technologies

N.B. : - Les réponses aux questions doivent être argumentées et aussi concises que possible.
- Le barème est donné à titre indicatif.
- Inscrivez votre numéro d'anonymat sur chacune des deux feuilles à joindre avant de les insérer dans votre copie.

Exercice 1 (20 points)

(1.1) (5 points)
Au moyen des portes logiques classiques AND, OR et NOT, écrivez l'équation logique d'un multiplexeur « MUX1 » à deux entrées A et B sur un bit chacune, et un bit de contrôle C, qui renvoie sur sa sortie S la valeur A si C = 0 et B si C = 1. Dessinez le schéma de câblage de ce multiplexeur, au moyen de portes logiques.

(1.2) (5 points)
En utilisant au besoin les composants créés dans la question précédente, concevez et dessinez le schéma de câblage d'un multiplexeur « MUX4 » à un bit de contrôle C, mais prenant des entrées A et B sur 4 bits chacune.

(1.3) (10 points)
En utilisant au besoin les composants créés dans les questions précédentes, dessinez le schéma de câblage d'un circuit multiplexeur à quatre entrées E_{00}, E_{01}, E_{10} et E_{11} sur 4 bits chacune, et deux bits de contrôle C_1C_0, C_1 étant le bit de poids fort.

T.S.V.P. ⟶

Exercice 2 (80 points)

Les ordinateurs ne savent manipuler que des nombres binaires, alors que les humains expriment les nombres sous forme décimale. Lors d'un calcul interactif avec un ordinateur, il faut donc convertir les nombres décimaux en nombres binaires puis, à la fin du calcul binaire, convertir le résultat en décimal, ce qui est assez coûteux. Pour éviter cela, certains processeurs, notamment utilisés dans les premières calculatrices numériques, disposaient d'instructions pour travailler directement avec des nombres entiers en base 10, en utilisant une représentation appelée « BCD », pour « *Binary-Coded Decimal* », ou en français « *Décimal Codé en Binaire* ».

L'idée du codage BCD est de coder chaque chiffre décimal sous forme binaire et de manipuler directement ce codage en machine, pour que chaque nombre ainsi codé puisse se lire directement sous forme décimale (par exemple, grâce à un afficheur à 7 segments). Quatre bits sont nécessaires au stockage en binaire des chiffres de 0 à 9, et les autres configurations binaires de ces quatre bits ne sont pas utilisées. Ainsi, le nombre décimal $815_{(10)}$ est représenté en BCD, sur deux octets, sous la forme $0000\ 1000\ 0001\ 0101_{(2)}$, qui se lit bien $0815_{(16)}$, alors qu'en binaire pur, la valeur $815_{(10)}$ serait codée sous la forme $0000\ 0011\ 0010\ 1111_{(2)}$, qui se lit $032F_{(16)}$.

Pour additionner deux nombres en BCD, il ne faut donc pas utiliser directement les additionneurs binaires classiques, car le résultat serait faux. Par exemple, puisque $815_{(10)} + 247_{(10)} = 1062_{(10)}$, l'addition $0815_{(16)} +_{\text{BCD}} 0247_{(16)}$ doit donner $1062_{(16)}$, alors qu'une addition binaire pure donnerait $0A5C_{(16)}$.

On suppose que l'on dispose déjà d'un additionneur binaire complet classique sur 4 bits, avec une retenue d'entrée $\mathtt{c}_{in}$, une retenue de sortie $\mathtt{c}_{out}$, deux groupes d'entrées, étiquetées de $\mathtt{x}_0$ à $\mathtt{x}_3$ et de $\mathtt{y}_0$ à $\mathtt{y}_3$, pour les deux nombres binaires `x` et `y` d'entrée à additionner, et d'un groupe de sorties étiquetées de $\mathtt{s}_0$ à $\mathtt{s}_3$ pour le résultat `s` (le bit 0 est le bit de poids le plus faible). Dans la suite, cet additionneur pourra être représenté schématiquement par une boîte appelée « `FA` ». On dispose en outre des portes logiques classiques AND, OR et NOT.

(2.1) (20 points)

Construisez un tableau à cinq colonnes donnant, pour chaque chiffre décimal compris entre 0 et 9 (écrit dans la première colonne), les valeurs binaires correspondant à :

- la représentation BCD de ce chiffre, codée sur 4 bits (deuxième colonne) ;
- le résultat, appelé a, de l'ajout de la valeur $3_{(10)}$ à la représentation BCD de la deuxième colonne, codé sous la forme de deux chiffres BCD (dizaine et unité) sur 4 bits chacun (troisième colonne) ;
- le résultat, appelé b, de l'ajout de la valeur $3_{(10)}$ à la représentation BCD de la deuxième colonne, au moyen de l'additionneur binaire complet classique, codé sur 4 bits (quatrième colonne) ;
- la différence $a - b$ entre les deux résultats précédents, ceux-ci étant maintenant interprétés tous les deux comme des nombres binaires purs (cinquième colonne).

(2.2) (20 points)

Par extension des résultats contenus dans le tableau précédent, quelles sont les valeurs que peut prendre la différence entre la somme BCD a et la somme binaire b de deux chiffres BCD quelconques ? Dans quel cas la différence entre les deux résultats est-elle non nulle ?

(2.3) (20 points)

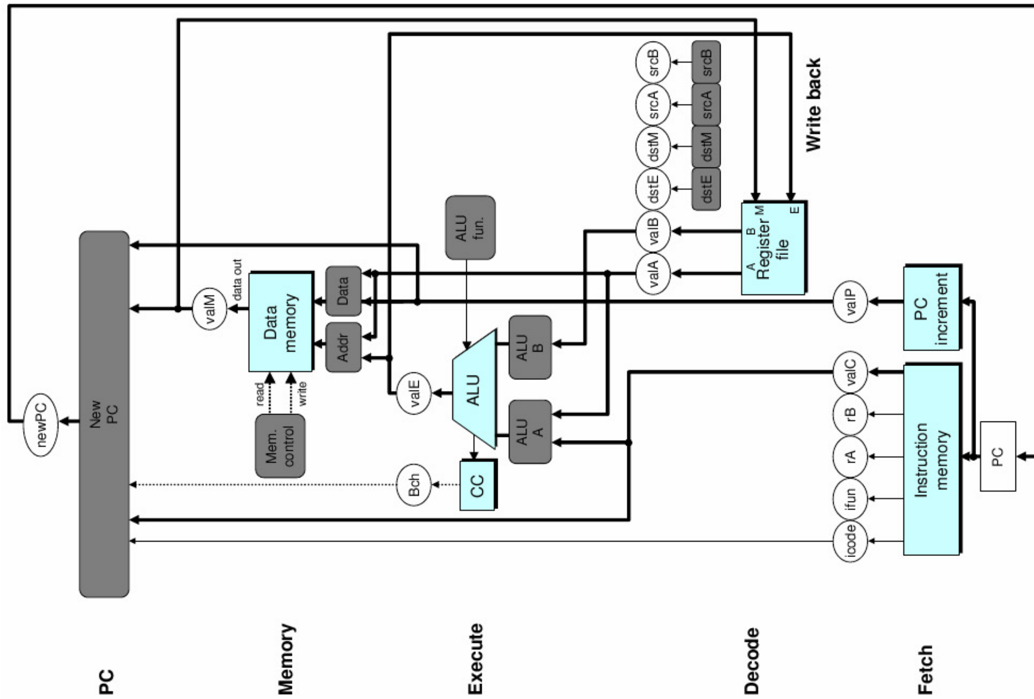
Donnez et justifiez la formule logique, puis faites le schéma, d'un circuit qui détecte quand la somme BCD calculée par un additionneur 4 bits « `FA` » classique est fausse (c'est-à-dire quand la différence étudiée à la question précédente est non nulle). Ce circuit a comme entrées la valeur `s` de la somme calculée par l'additionneur ainsi que la retenue sortante. Sa sortie `e` vaut 1 s'il y a erreur, et 0 sinon. Dans la suite, cette fonction pourra être représentée schématiquement par une boîte « `E` ».

(2.4) (20 points)

En utilisant, entre autres, le circuit de la question précédente et deux additionneurs binaires à 4 bits, dessinez le schéma d'une « *tranche* » d'additionneur BCD, qui prend en entrée deux chiffres BCD `x` et `y` sur 4 bits chacun, et une retenue d'entrée $\mathtt{c}_{in}$, et produit en sortie un chiffre BCD `s` sur 4 bits et une retenue de sortie $\mathtt{c}_{out}$.

Exercice 3 (100 points)

La figure ci-dessous est le schéma de l'architecture Y86 séquentielle.



Le tableau ci-dessous décrit le fonctionnement des différents étages de l'architecture Y86 séquentielle lors de l'exécution des deux instructions ≪ `call valC` ≫ et ≪ `rmmovl rA,valC(rB)` ≫ du jeu d'instructions Y86.

call valC	rmmovl rA,valC(rB)
icode:ifun = M_1[PC] valC = M_4[PC+1] valP = PC + 5	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valC = M_4[PC+2] valP = PC + 6
valB = R[%esp]	valA = R[rA] valB = R[rB]
valE = valB + (-4)	ValE = valB + valC
M_4[valE] = valP	M_4[valE] = valA
R[%esp] = valE	
PC = valC	PC = valP

Rappel : ≪ R[x] ≫ indique un accès à la banque de registres à l'adresse (numéro de registre) x, et ≪ M_y[x] ≫ indique un accès à la mémoire centrale (d'instructions et/ou de données) de y octets à l'adresse x. ≪ PC ≫ est le registre compteur ordinal.

(3.1) (20 points)

Expliquez le sens des opérations effectuées à chaque étage pour l'instruction ≪ `call valC` ≫.

N.B. : **on ne veut pas de paraphrase** ; il faudra, pour chaque étape, expliquer *pourquoi* les opérations sont effectuées.

(3.2) (20 points)

En vous aidant éventuellement du code HCL joint au sujet, construisez le tableau décrivant le fonctionnement des différents étages de l'architecture Y86 séquentielle pour exécuter l'instruction ≪ `ret` ≫, à l'image des tableaux présentés au début de cet exercice.

T.S.V.P. ⟶

On veut câbler en HCL l'instruction « `creg rA` » (pour « *call register* »), qui effectue un appel de fonction à l'adresse contenue dans le registre `rA`. Cette instruction permet notamment de mettre en œuvre des pointeurs de fonctions.

(3.3) (20 points)

En vous aidant éventuellement du code HCL joint au sujet, construisez le tableau décrivant le fonctionnement des différents étages de l'architecture `Y86` séquentielle pour exécuter l'instruction « `creg rA` », à l'image des tableaux présentés au début de cet exercice.

(3.4) (20 points)

Ajoutez à la première feuille HCL ci-jointe, à détacher et à rendre avec votre copie, les instructions HCL permettant de mettre en œuvre l'instruction `creg`, dans le cas où la valeur d'`icode` de cette instruction, appelée `CREG`, est différente de celle de `CALL`.

(3.5) (20 points)

Ajoutez à la seconde feuille HCL ci-jointe, à détacher et à rendre avec votre copie, les instructions HCL permettant de mettre en œuvre l'instruction `creg`, dans le cas où la valeur d'`icode` de cette instruction est celle de `CALL`, mais où son ifun est égal à `1` (celui de `call` étant égal à `0`).

Modifiez le code de la feuille afin de réaliser cette factorisation de code, en distinguant entre les nouveaux cas « `((icode == CALL) && (ifun == 0))` » et « `((icode == CALL) && (ifun == 1))` ».

Marquez par un symbole ⊗ toutes les lignes où l'usage du code `CALL` est factorisé entre les deux instructions `call` et `creg`.

N.B. *: n'oubliez pas d'inscrire votre numéro d'anonymat sur la feuille avant de l'insérer dans votre copie.*

— Première feuille à détacher et à rendre avec la copie —

Numéro d'anonymat :

```
################ 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 };

# List of all valid instructions
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 { RMMOVL, MRMOVL, OPL } : 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
];
```

TSVP .../... ⟶

```
################ 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, MRMOVL, PUSHL, CALL } : valE;
        icode in { POPL, RET } : valA;
        # Other instructions don't need address
];

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

################ Program Counter Update ############################

## What address should instruction be fetched at
int new_pc = [
        icode == CALL : valC;
        icode == JXX && Bch : valC;
        icode == RET : valM;
        1 : valP;
];
```

— Seconde feuille à détacher et à rendre avec la copie —

Numéro d'anonymat :

```
################ 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 };

# List of all valid instructions
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 { RMMOVL, MRMOVL, OPL } : 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
];
```

TSVP .../... ⟶

```
################ 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, MRMOVL, PUSHL, CALL } : valE;
	icode in { POPL, RET } : valA;
	# Other instructions don't need address
];

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

################ Program Counter Update ############################

## What address should instruction be fetched at
int new_pc = [
	icode == CALL : valC;
	icode == JXX && Bch : valC;
	icode == RET : valM;
	1 : valP;
];
```