université de BORDEAUX	**ANNÉE UNIVERSITAIRE 2018 – 2019** **SESSION 1 D'AUTOMNE** **Parcours / Étape : LSTS / L2** **Code UE : 4TIN304U** **Épreuve :** Architecture des ordinateurs **Date :** 11 janvier 2019 **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 la feuille à joindre avant de l'insérer dans votre copie.

Question 1 (50 points)

On veut réaliser un chenillard rotatif à trois lampes placées de gauche à droite, notées « $\mathtt{s_2 s_1 s_0}$ » et qui, à chaque cycle d'horloge, voit l'unique lumière allumée se décaler vers la droite, avant de réapparaître à gauche, selon la séquence des trois configurations suivantes : « `100` », « `010` » et « `001` ».

(1.1) (10 points)

En supposant que la valeur initiale est « `100` », créez, à base de trois bascules D à front montant, de sorties respectives $\mathtt{q_2}$, $\mathtt{q_1}$ et $\mathtt{q_0}$, d'un signal horloge `H` et éventuellement d'autres portes logiques, un circuit chenillard permettant d'allumer les sorties $\mathtt{s_2}$, $\mathtt{s_1}$ et $\mathtt{s_0}$ tel que demandé ci-dessus.

On veut maintenant réaliser un chenillard « zig-zag », qui passe par les quatre configurations suivantes : « `100` », « `010` », « `001` » et de nouveau « `010` ».

(1.2) (10 points)

Modifiez le circuit précédent, en utilisant quatre bascules D, pour créer un chenillard « zig-zag ». En particulier, donnez les équations logiques des sorties $\mathtt{s_2}$, $\mathtt{s_1}$ et $\mathtt{s_0}$ à partir de $\mathtt{q_3}$, $\mathtt{q_2}$, $\mathtt{q_1}$ et $\mathtt{q_0}$.

(1.3) (20 points)

On n'a en fait besoin que de deux bascules D pour stocker quatre états distincts. Construisez un dispositif compteur non signé à deux bits, appelé `CPT`, qui parcourt avec ses sorties $\mathtt{q_1}$ et $\mathtt{q_0}$ les quatre états différents, en changeant d'état à chaque front montant de l'horloge `H`.
N.B. : Mettez en œuvre une solution qui ne crée pas de dérive de l'horloge.

À partir de maintenant, et même si vous n'avez pas répondu aux questions précédentes, vous pouvez utiliser ce compteur non signé à deux bits `CPT`, possédant les sorties $\mathtt{q_1}$ et $\mathtt{q_0}$ et le comportement décrits ci-dessus.

(1.4) (10 points)

Donnez les équations logiques des sorties $\mathtt{s_2}$, $\mathtt{s_1}$ et $\mathtt{s_0}$ du chenillard « zig-zag » à partir des sorties $\mathtt{q_1}$ et $\mathtt{q_0}$ du compteur `CPT`, et proposez un câblage de ce chenillard.

Question 2 (110 points)

La valeur « a modulo b », parfois notée « `a % b` » dans de nombreux langages de programmation, et ici notée comme la fonction `modulo(a,b)`, désigne le reste de la division entière de a par b. De même, on notera `quotient(a,b)` le quotient de la division entière de a par b. De fait, $a = \mathtt{quotient}(a,b) * b + \mathtt{modulo}(a,b)$.

(2.1) (20 points)

Codez en Y86 la fonction `modulo(a,b)` au moyen d'une simple boucle itérative. Appelez cette fonction « `modit(a,b)` » (pour « modulo itératif »), afin de bien la différencier de la suivante.

TSVP .../... ⟶

(2.2) (10 points)

Codez en Y86 la fonction `main()` permettant d'appeler la fonction `modit`, en lui passant en paramètre deux valeurs entières stockées en mémoire aux adresses d'étiquettes `pa` et `pb`, et en sauvegardant le résultat en mémoire également, à l'adresse d'étiquette `pm`.

(2.3) (20 points)

Codez en Y86 la fonction `modre(`a`,`b`)` ci-contre, qui calcule le modulo de façon récursive.

```
int
modre (int a, int b)
{
  int i;
  if (a < b)
    return (a);
  i = modre (a, 2 * b);
  if (i >= b)
    return (i - b);
  return (i);
}
```

(2.4) (20 points)

Codez en Y86 la fonction `quotient(`a`,`b`)` au moyen d'une simple boucle itérative. Appelez cette fonction « `quotit(`a`,`b`)` » (pour « quotient itératif »), afin de bien la différencier de la suivante.

(2.5) (20 points)

En vous inspirant de la fonction `modre(`a`,`b`)`, écrivez en pseudo-langage C la fonction « `quotre(`a`,`b`)` » (pour « quotient récursif »). Dans ce cas, pour chaque appel récursif, il faut renvoyer une paire de valeurs au lieu d'une seule : le reste de la soustraction (comme pour `modre`) et la contribution de la récursion au quotient. Cela suppose également de passer une troisième valeur à la fonction : ladite contribution, initialement égale à 1, et qui double à chaque appel. On aura donc une fonction `quotre(`a`,`b`)`, qui appellera immédiatement une fonction `quotre2(`a`,`b`,1)`, la récursion portant sur la fonction `quotre2`.

(2.6) (20 points)

Codez en Y86 les fonctions `quotre(`a`,`b`)` et `quotre2(`a`,`b`,`c`)`. En Y86, lorsqu'une fonction doit retourner deux valeurs, ce sont les registres `%eax` et `%edx` qui sont utilisés.

Question 3 (40 points)

Le tableau ci-dessous représente le fonctionnement des différents étages lors de l'exécution des trois instructions « `rmmovl rA,D(rB)` », « `pushl rA` » et « `ret` ».

Étage	rmmovl rA,D(rB)	pushl rA	ret
Fetch	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valC = M_4[PC+2] valP = PC + 6	icode:ifun = M_1[PC] rA:rB = M_1[PC+1] valP = PC + 2	icode:ifun = M_1[PC] valP = PC + 1
Decode	valA = R[rA] valB = R[rB]	valA = R[rA] valB = R[%esp]	valA = R[%esp] valB = R[%esp]
Execute	ValE = valB + valC	valE = (-4) + valB	valE = 4 + valB
Memory	M_4[valE] = valA	M_4[valE] = valA	valM = M_4[valA]
Write back		R[%esp] = valE	R[%esp] = valE
PC update	PC = valP	PC = valP	PC = valM

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)

En vous basant sur les informations précédentes, remplissez le tableau correspondant aux instructions « `mrmovl D(rB),rA` », « `popl rA` » et « `call D` ».

(3.2) (20 points)

Complétez le code HCL de la feuille jointe afin de prendre en compte les trois instructions que vous venez de définir.

— 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, IRMOVL, RMMOVL };

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

# 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 { OPL, RRMOVL, RMMOVL, PUSHL } : rA;
        icode in { RET } : RESP;
        1 : RNONE; # Don't need register
];

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

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

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

TSVP .../... ⟶

```
################ Execute Stage   ###################################

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

## Select input B to ALU
int aluB = [
        icode in { RMMOVL, OPL, PUSHL, RET } : valB;
        (icode == IRMOVL) && (ifun == 1) : 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 { RET };

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

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

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

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

## What address should instruction be fetched at
int new_pc = [

        icode == JXX && Bch : valC;
        icode == RET : valM;
        1 : valP;
];
```