Exercice 1

(1.1) On écrit les tables de Karnaugh. "ϕ" correspond au cas "0000", non pris en charge. [10]

S_1	E_1E_0 = 00	01	11	10
E_3E_2 = 00	ϕ_1	0	0	0
01	1	1	1	1
11	1	1	1	1
10	1	1	1	1

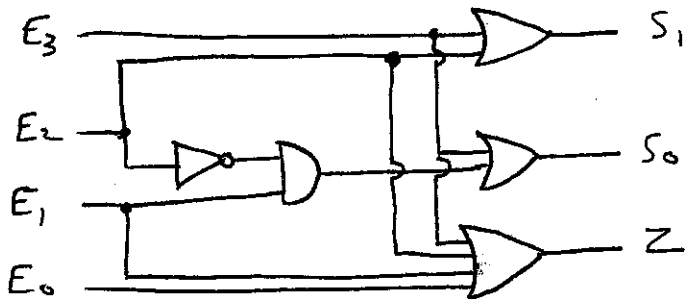
$S_1 = E_3 + E_2$

S_0	E_1E_0 = 00	01	11	10
E_3E_2 = 00	ϕ_0	0	1	1
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

$S_0 = E_3 + \overline{E_2}\,E_1$

(1.2) $Z = \overline{E_3 + E_2 + E_1 + E_0}$ [5]

(1.3) [5]



Exercice 2

(2.1) On recharge (CO=LD) le registre avec la valeur 6 chaque fois qu'il atteint la valeur 15. Les valeurs parcourues sont donc: [10]

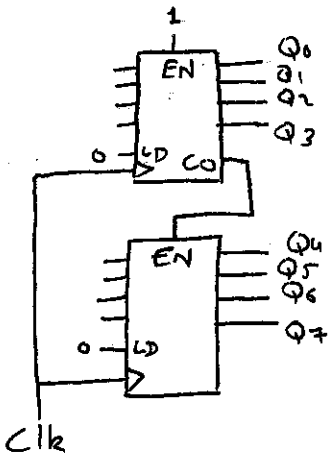
6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 6, 7, 8, 9, etc...

(2.2) On recharge (CO=X) le registre avec la valeur 0 chaque fois qu'il atteint la valeur 6 (X=1 pour 0110). Les valeurs parcourues sont donc: [10]

0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, etc...

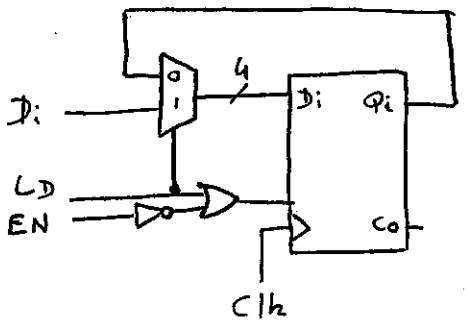
(2.3) Le compteur des 4 bits de poids fort doit être incrémenté seulement quand le compteur des 4 bits de poids faible est arrivé à 15 et doit repasser à 0. On branche donc CO_L sur EN_H.

(2) 10



(2.4) Pour ne pas avoir de dérive de l'horloge, il faut faire en sorte de renvoyer l'ancienne valeur du compteur dans la nouvelle lorsque EN=0. On peut arriver au résultat voulu au moyen d'un multiplexeur :

10



Exercice 3

(3.1)

10

```
not:  mrmovl  4(%esp), %eax
      ixorl   -1, %eax        # Tous les bits à "1"
      ret
```

(3.2) On aurait pu calculer $(0-x)$, mais l'instruction SUBL nous est interdite. On revient donc à la définition première du complément à 2 : $-x = \bar{x}+1$. Donc : [10]

```
neg:    movl    4(%esp), %eax
        ixorl   -1, %eax        # not, cf. question précédente
        iaddl   1, %eax
        ret
```

Exercice 4

(4.1) Le meilleur équilibrage est : [10]

(20 70 30) [20] (40 60) [20]

120 100

Durée minimale : 140 ps
Latence : $2 \times 140 = 280$ ps
Débit : $\frac{1}{140.10^{-12}} = 7{,}14$ Gops

(4.2) Le meilleur équilibrage est : [10]

(20 70) [20] (30 40) [20] (60) [20]

90 70 60

Durée minimale : 110 ps
Latence : $3 \times 110 = 330$ ps
Débit : $\frac{1}{110.10^{-12}} = 9{,}09$ Gops

(4.3) Le pipe-line de profondeur optimale est : [10]

(20) [20] (70) [20] (30 40) [20] (60) [20]

20 70 70 60

Durée minimale : 90 ps
Latence : $4 \times 90 = 360$ ps
Débit : $\frac{1}{90.10^{-12}} = 11{,}1$ Gops

Exercice 5

(5.1) Ce programme stocke dans la valeur res le maximum des size valeurs contenues dans le tableau d'adresse de début t. [10]

(5.2) [10]

pushl %eax	F	D	E	M	W					
mrmovl t(%esi), %edx		F	D	E	M	W				
subl %edx, %eax			F	D	D	E	M	W		
popl %eax				F	F	D	E	M	W	
jge next						F	D	E	M	W

On a un "load-use hazard" entre le "mrmovl" et le "sub".

(5.3) Il suffit de gagner un cycle supplémentaire entre les deux instructions créant le "load-use hazard". [10]

```
mrmovl t(%esi), %edx
pushl  %eax
subl   %edx, %eax
popl   %eax
jge    next
```

(5.4) Cf feuille jointe. [15]
CMPL est comme un SUBL mais qui ne met pas à jour le registre destination.

(5.5) Cf feuille jointe. [10]
On utilise l'ALU pour faire un SUBL entre Ø et la valeur du PB.

(5.6) INCL et DECL se comportent comme NEGL : elles ajoutent une constante à rB : +1 ou -1. [15]

Cf feuille jointe.

On supposera que INCL et DECL modifient les CC, pour la suite de l'optimisation.

(5.7) On peut les factoriser facilement avec NEGL, voire avec OPL. [10]

Déjà, entre INCL et DECL, seule la valeur de la constante passée à l'entrée A de l'UAL change!

(5.8) On ne touche qu'à la boucle principale : [10]

```
loop: mrmovl 8(%esi), %edx
      cmpl   %edx, %eax
      jge    next
      rrmovl %edx, %eax
next: iaddl  4, %esi
      decl   %ecx
      jne    loop
```

On gagne : 2 (pushl %eax) + 2 (popl %eax) + 4 (isubl → decl)

= 8 octets

— Feuille à détacher et à rendre avec la copie —

Numéro d'anonymat : CORRIGÉ

```
################ Fetch Stage     ###################################

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

# Predict next value of PC
int new_F_predPC = [
f_icode in { JXX, CALL } : f_valC;
1 : f_valP;
];

################ Decode Stage ######################################

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

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

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

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

## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [
D_icode in { CALL, JXX } : D_valP; # Use incremented PC
d_srcA == E_dstE : e_valE;    # Forward valE from execute
d_srcA == M_dstM : m_valM;    # Forward valM from memory
d_srcA == M_dstE : M_valE;    # Forward valE from memory
d_srcA == W_dstM : W_valM;    # Forward valM from write back
```

CMPL : 15
NEGL : 10
INCL, DECL : 15

CMPL

CMPL, NEGL, INCL, DECL

NEGL, INCL, DECL

```
d_srcA == W_dstE : W_valE;    # Forward valE from write back
1 : d_rvalA;  # Use value read from register file
];

int new_E_valB = [
d_srcB == E_dstE : e_valE;    # Forward valE from execute
d_srcB == M_dstM : m_valM;    # Forward valM from memory
d_srcB == M_dstE : M_valE;    # Forward valE from memory
d_srcB == W_dstM : W_valM;    # Forward valM from write back
d_srcB == W_dstE : W_valE;    # Forward valE from write back
1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage ######################################

## Select input A to ALU
int aluA = [                  E_icode == NEGL : E_valB;
E_icode in { RRMOVL, OPL } : E_valA;
E_icode in { IRMOVL, RMMOVL, MRMOVL } : E_valC;
E_icode in { CALL, PUSHL } : -4;        E_icode == INCL : +1;
E_icode in { RET, POPL } : 4;           E_icode == DECL : -1;
# Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
E_icode in { RMMOVL, MRMOVL, OPL, CALL, CMPL, INCL, DECL,
      PUSHL, RET, POPL } : E_valB;
E_icode in { RRMOVL, IRMOVL } : 0;          NEGL
# Other instructions don't need ALU
];

## Set the ALU function
int alufun = [          E_icode in {NEGL, CMPL} : ALUSUB;
E_icode == OPL : E_ifun;
1 : ALUADD;
];      ↖ Par défaut pour INCL et DECL

## Should the condition codes be updated?
bool set_cc = E_icode == RPL; E_icode in {OPL, CMPL, INCL, DECL}

################ Memory Stage ######################################

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

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

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