Question 1

(1.1) A et B sont des entiers non signés sur 8 bits, donc $A \leq 2^8-1$ et $B \leq 2^8-1$. Il en découle que

$$P \leq (2^8-1)(2^8-1)$$
$$\leq 2^8.2^8 - 2.2^8 + 1$$
$$\leq 2^{16} - 2^9 + 1$$

Donc, comme $(2^{16}-1) \geqslant P \geqslant 2^{15}-1$, P peut et doit être câblé sur $x = 16$ bits.

(1.2)

a \ b	0	1
0	0	0
1	0	1

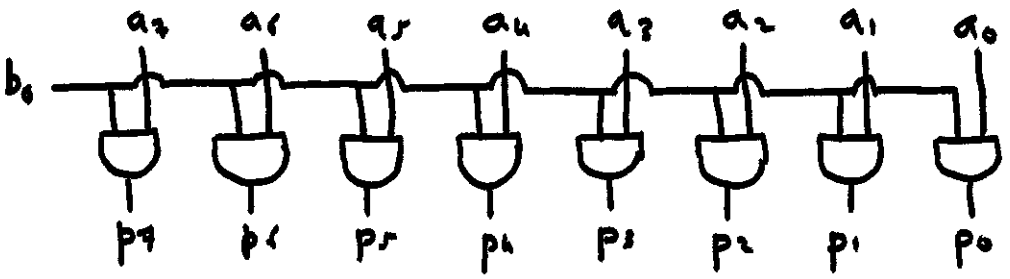
C'est un AND.

(1.3)

```
                1 0 1 1 0 0 0 1
              x 1 1 0 1 0 0 1 1
            -----------------------
                1 0 1 1 0 0 0 1     ← x 1 : copie
              1 0 1 1 0 0 0 1 .
            0 0 0 0 0 0 0 0 . .     ← x 0 tout à 0
          0 0 0 0 0 0 0 0 . . .
        1 0 1 1 0 0 0 1 . . . .
      0 0 0 0 0 0 0 0 . . . . .
    1 0 1 1 0 0 0 1 . . . . . .
  1 0 1 1 0 0 0 1 . . . . . . .
-------------------------------
1 0 0 1 0 0 0 1 1 1 1 0 0 0 1 1
```

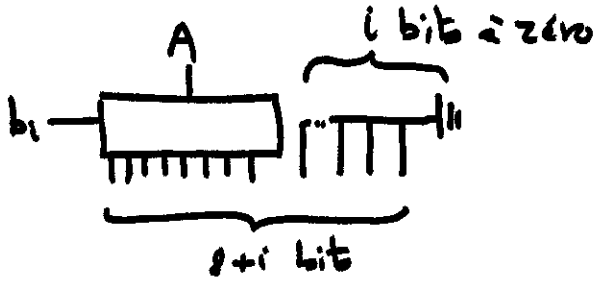
10

(1.4)



(1.5)

5



Pour calculer P_i, il suffit de décaler P_0 de i bits vers la gauche, en ajoutant i fils à zéro pour les bits de poids les plus faibles.

(1.6)

5

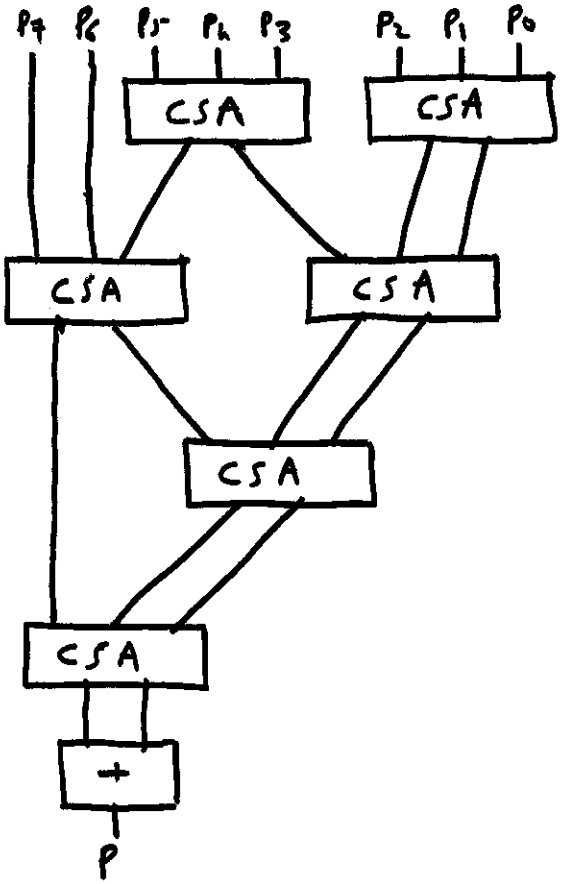
x	y	z	c	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(1.7) Soit $m = \max(f,g,h)$. On sait que $m \leq x$ (question 1.1). Comme il n'y a pas de propagation de retenue, la taille de s est celle du plus grand mot, donc $s = m$.

Comme c accumule les retenues, dont celle du bit de poids le plus fort de la somme, $c = m+1$, sauf si $m = x$, car $c \leq 16$ (question 1.1). Donc $c = \min(m+1, x)$.

10

(1.8)

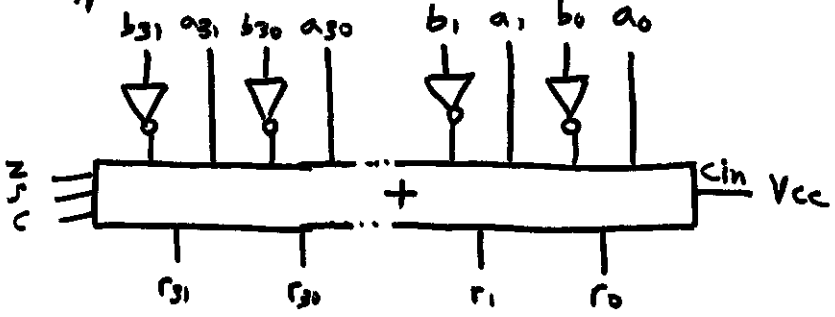


Pour minimiser le chemin critique, il faut utiliser une structure arborescente.
A chaque niveau, on utilise autant de circuits CSA que possible, pour réduire le nombre de fils

Question 2

(2.1) En notation "complément à deux", l'opposé d'un nombre s'obtient en inversant tous les bits du nombre, puis en ajoutant 1 au résultat. 10

On peut effectuer cette opération grâce à l'additionneur fourni :



Il suffit pour cela d'inverser tous les bits de B en entrée, et d'utiliser la valeur Cin de retenue entrante pour ajouter 1 au résultat.

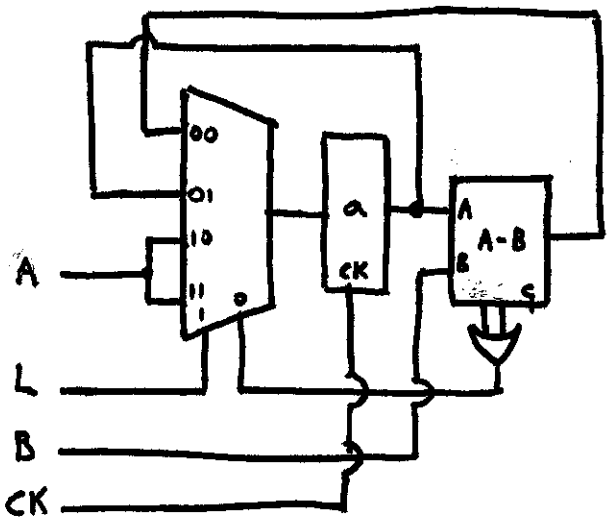
(2.2) On a trois cas d'entrée possibles, donc il faut utiliser un multiplexeur 32 x 2, qui a 4 entrées. On a donc deux entrées identiques. Comme le signal L est prioritaire, c'est lui que l'on va utiliser pour les deux entrées identiques. 15

Un nombre résultat est strictement positif si Z=0 et S=0. A l'inverse, ce résultat est négatif ou nul si Z=1 ou S=1, et dans ce cas c'est l'ancienne valeur de a qu'il faut réinjecter.

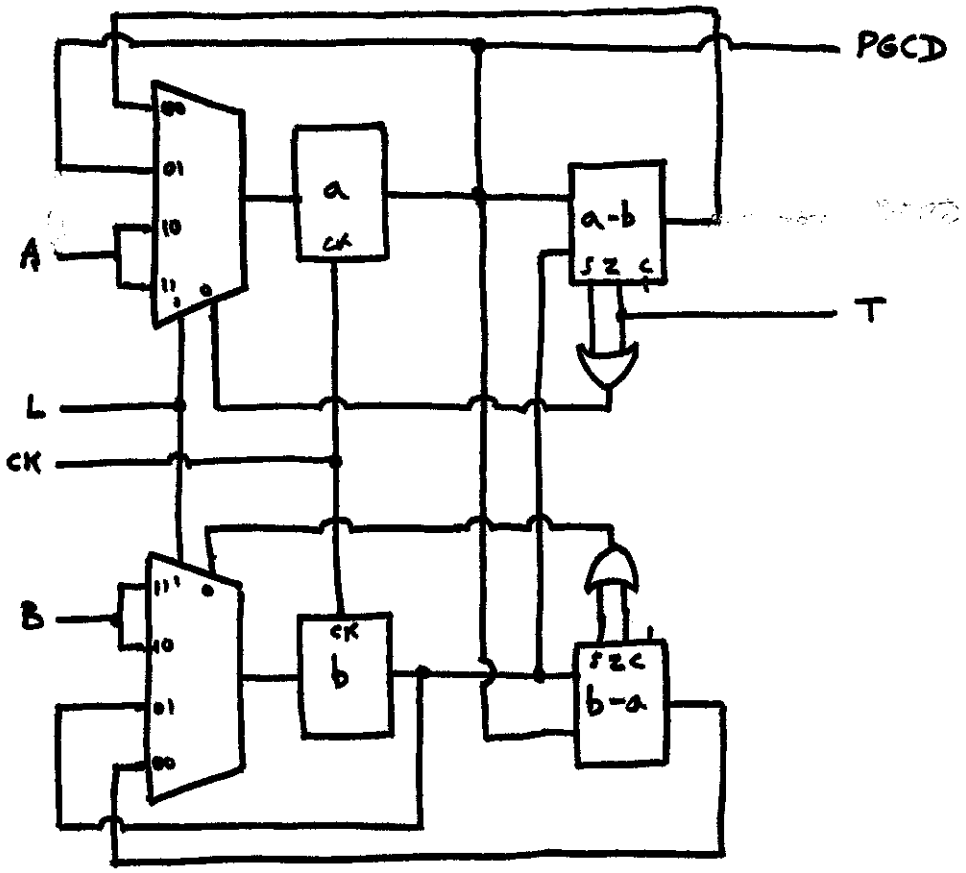
La logique de commande du multiplexeur 32 x 2 est donc :

bit 1 \ bit 0 (Z+S)	0	1
0	a-B	a
1	A	A

L

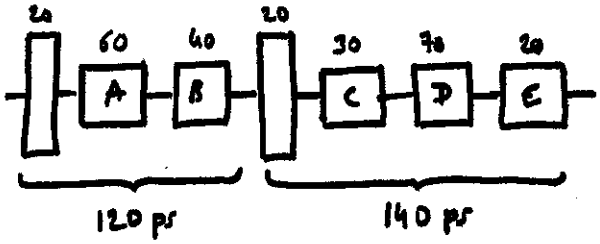


(2.3) Pour ce nouveau circuit, on a besoin de deux registres, deux multiplexeurs et deux soustracteurs. La sortie T est à 1 quand le bit Z d'un soustracteur (en fait, des deux) passe à 1. Dès que c'est le cas, les valeurs des deux registres ne seront plus modifiées, puisque quand Z=1, on copie l'ancienne valeur du registre. 15



Question 3

(3.1)



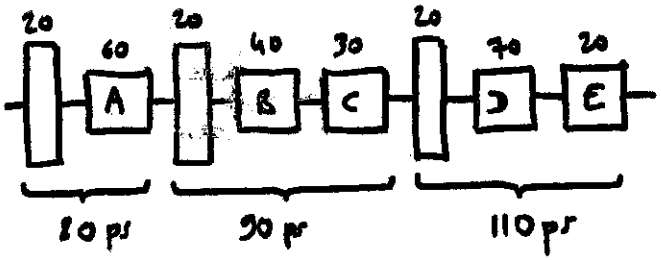
Si on mettait le latch après C, on aurait 150 et 110 ps, ce qui serait moins efficace

10

Durée minimale : 140 ps

Latence : 2 × 140 ps = 280 ps

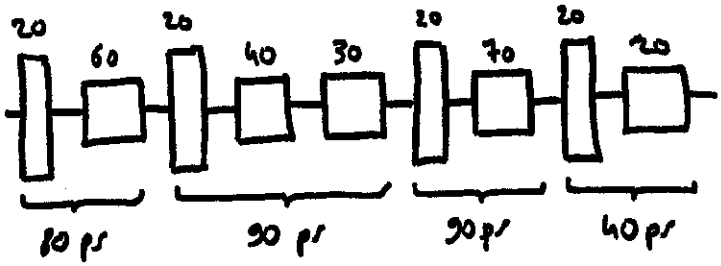
Débit : $\frac{10^{12}}{140} = 7{,}14$ Gop/s

(3.2) 

10

Durée minimale : 110 ps
Latence : $3 \times 110 = 330$ ps
Débit : $\frac{10^{12}}{110} = 9{,}09$ Gop/s

(3.3) Le pipe-line de profondeur optimale est celui-ci : 10



Il est inutile de rajouter un étage, car on ne pourra jamais avoir une durée inférieure à 90 ps, du fait de l'étage 3.

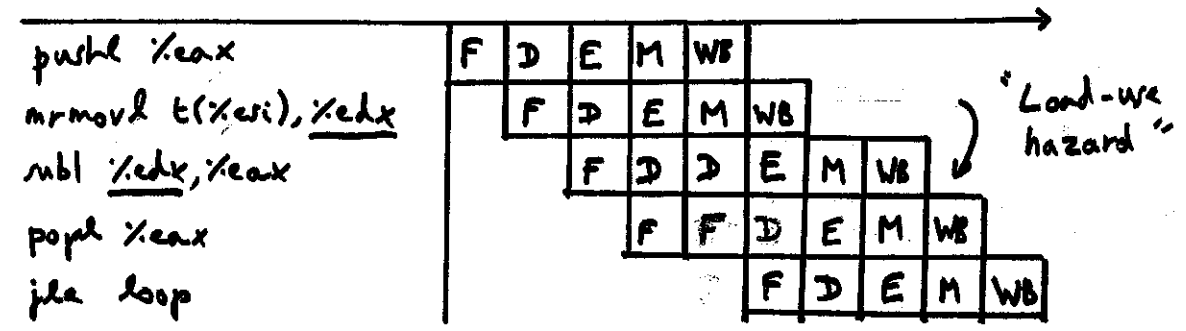
Durée minimale : 90 ps
Latence : $4 \times 90 = 360$ ps
Débit : $\frac{10^{12}}{90} = 11{,}1$ Gop/s

Question 4

10

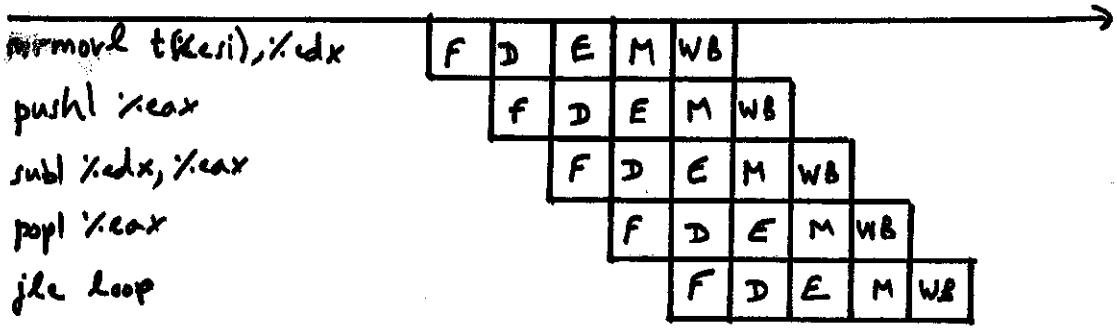
(4.1) Ce programme place dans la mémoire à l'adresse "res" la valeur de la plus petite case du tableau commençant à l'adresse "t" et de taille contenue à l'adresse "size".
C'est le registre EAX qui contient la plus petite valeur trouvée jusqu'à présent. Il est chargé avec la valeur de la première case du tableau. Ensuite, lors de la boucle "loop", on avance à la case suivante, on teste la fin de la boucle, on sauvegarde la valeur de EAX pour faire la comparaison avec la valeur lue depuis la mémoire, et on modifie EAX si elle est plus petite, avant de reboucler.

(4.2) 10



(4.3) 10

Il suffit d'intercaler une instruction "utile" entre l'instruction qui change la valeur dans EDX et celle qui l'utilise :



(4.4) Voir feuille jointe. CMPL est une sorte d'OPL. 10

(4.5) Voir feuille jointe. NEGL est une sorte d'OPL. Cependant, comme elle ne mobilise qu'un seul registre, la source et la destination sont identiques (rB), ce qui impose un traitement spécifique pour l'entrée aluA. 10

(4.6) Voir feuille jointe. INCL et DECL sont des sortes d'OPL. 10

(4.7) INCL et DECL ne diffèrent que par la valeur d'incrément passée à l'UAL. Il est donc facile de les factoriser avec un opcode commun et un ifun distinct. 10
Également, il est très facile de les factoriser avec OPL, car elles ont le même fonctionnement général.

(4.8) Avec CMPL, on n'a plus besoin de faire les "PUSHL - POPL" pour sauvegarder EAX lors de chaque test avec la valeur courante du tableau. On re-perd donc un cycle à cause du "load-use hazard", mais on gagne le cycle du "POPL". On n'a plus besoin de la pile. [10]

- pas ∅

```
        mrmovl  size,%ecx
        xorl    %esi,%esi
        mrmovl  t(%esi),%eax
loop:   iaddl   4,%esi
        decl    %ecx
        je      endl
        mrmovl  t(%esi),%edx
        cmpl    %edx,%eax
        jle     loop
        rmmovl  %edx,%eax
        jmp     loop
endl:   rmmovl  %eax,res
        halt
```

Gains :

- suppression du "irmovl 200,%esp" : 6
- "cmpl" au lieu de "push-sub-pop" : 4 (2 pour push, 2 pour pop)
- "decl" au lieu de "isubl" : 4 (6 pour isubl - 2 pour decl)

— Feuille à détacher et à rendre avec la copie —

Numéro d'anonymat : CORRIGÉ

CMPL : 10
NEGL : 10
INCL } : 10
DECL

```
################ 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

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 in { RRMOVL, OPL } : E_valA;
E_icode in { IRMOVL, RMMOVL, MRMOVL } : E_valC;
E_icode in { CALL, PUSHL } : -4;
E_icode in { RET, POPL } : 4;
# Other instructions don't need ALU
];

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

## Set the ALU function
int alufun = [
E_icode == OPL : E_ifun;
1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode == OPL;

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

E_icode == NEGL : E_valB

E_icode == INCL : +1

E_icode == DECL : -1

CMPL, INCL, DECL,

NEGL

E_icode in {NEGL, CMPL} : ALUSUB;

Par défaut pour INCL et DECL

E_icode in {OPL, CMPL, INCL, DECL}