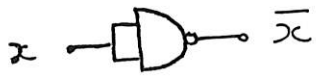
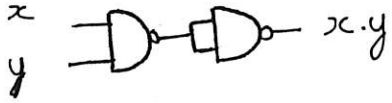
Question 1

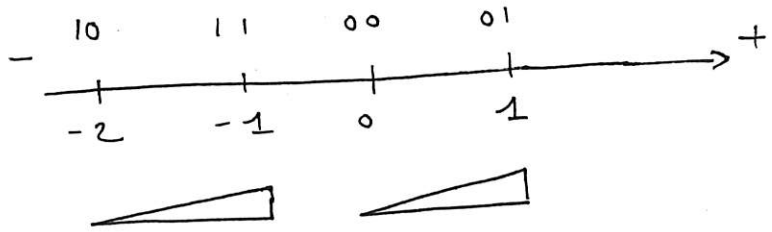
(1.1) Un ensemble de fonctions est complet s'il permet de représenter toutes les fonctions (c'est à dire tous les vecteurs de vérité) possibles. Comme toute fonction booléenne peut être exprimée comme la somme de produits de termes, il faut vérifier que, à partir des portes NAND, on peut construire les fonctions AND, OR et NOT. C'est bien le cas :



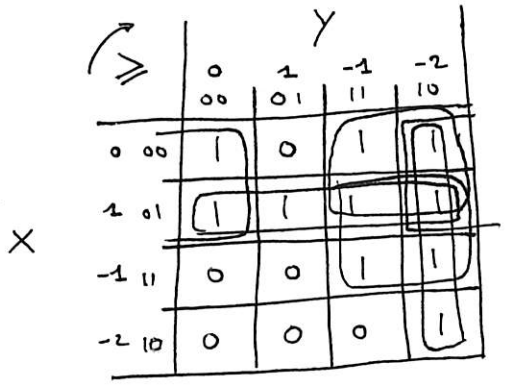


et par la loi de De Morgan : $\overline{\overline{x}.\overline{y}} = x + y$.

(1.2) Avec la notation en complément à deux :



(1.3) La table de Karnaugh est la suivante :



La forme positive est très complexe :

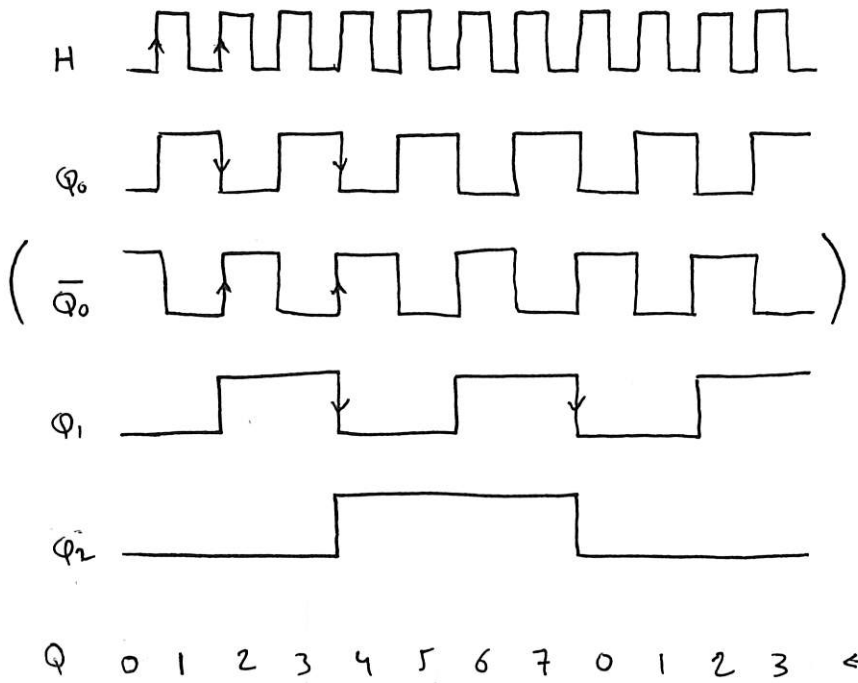
$$F(x_1,x_0,y_1,y_0) = \overline{x_1}x_0 + \overline{y_0}\,\overline{x_1} + y_1x_0 + y_1\overline{x_1} + y_1\overline{y_0}$$

La forme négative comporte moins de termes :

$$F(x_1,x_0,y_1,y_0) = \overline{\overline{x_0}\,\overline{y_1}\,y_0 + x_1\overline{x_0}\,y_0 + x_1\overline{y_1}}$$

Question 2

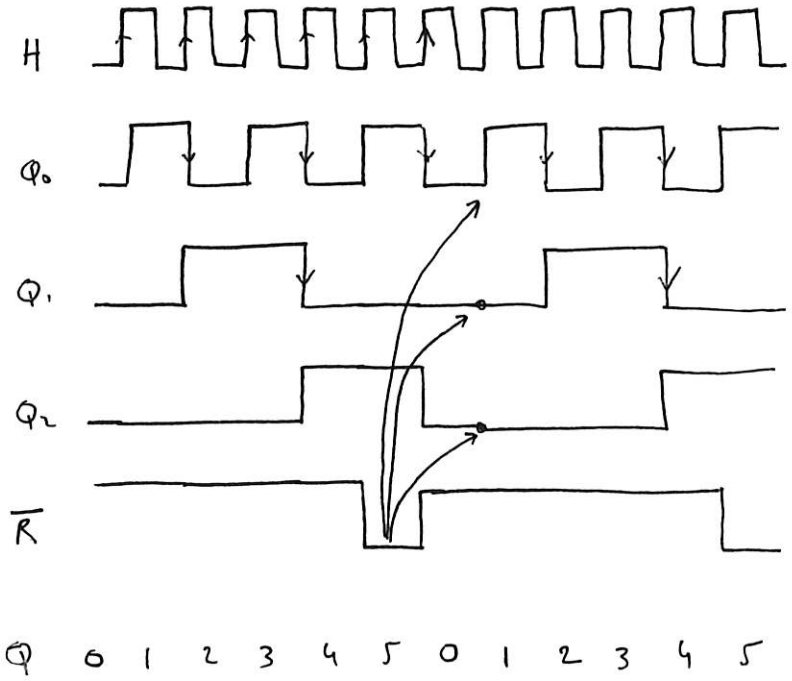
(2.1)



Le changement d'état de chaque bascule s'opère sur le front montant de son signal CLK, c'est-à-dire sur le front descendant du signal précédent

La valeur $Q = Q_2 Q_1 Q_0$ énumère toutes les configurations dans l'ordre croissant. C'est un compteur.

(2.2)

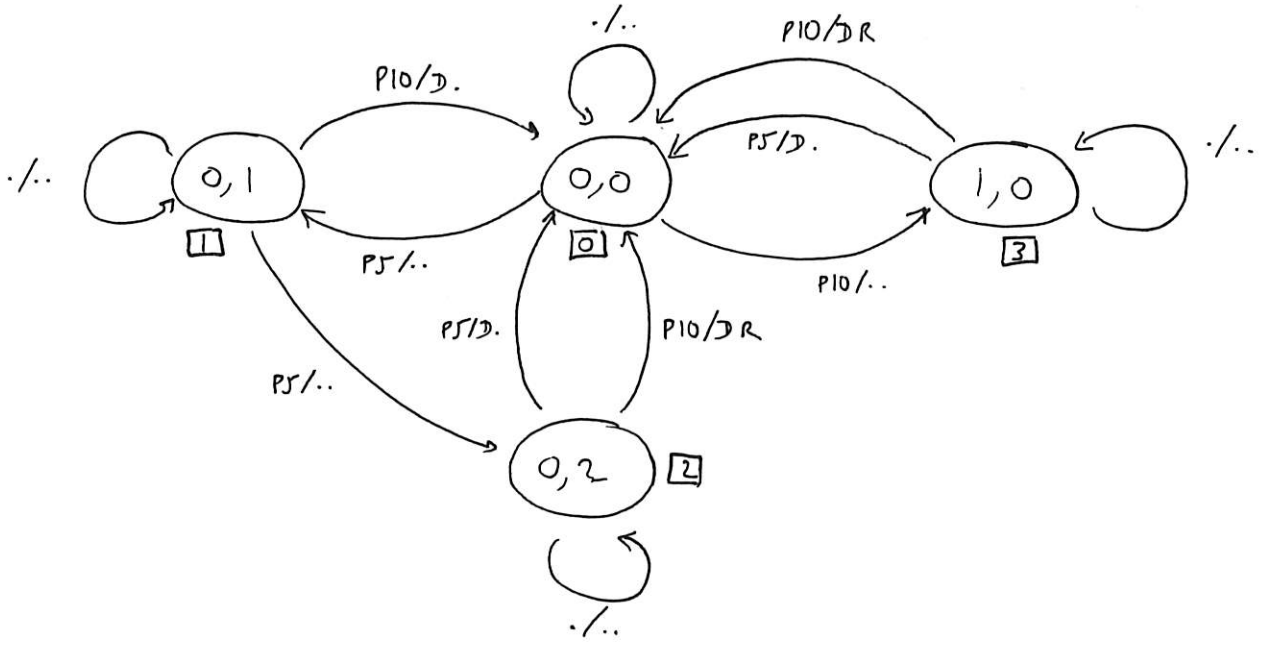


Appelons $\overline{R}$ le résultat de la porte NAND en bas à droite du circuit. $\overline{R}$ vaut 1, et le circuit fonctionne alors comme le précédent, tant que l'on n'a pas $Q_0 = Q_2 = 1$. À ce moment, $\overline{R}$ passe à 0, et toutes les cellules seront initialisées à 0 au prochain cycle.

C'est un compteur jusqu'à 5.

Question 3

(3.1) Le distributeur devant être prêt à servir une fois le bonbon distribué, il faut reboucler sur l'état initial après la distribution.

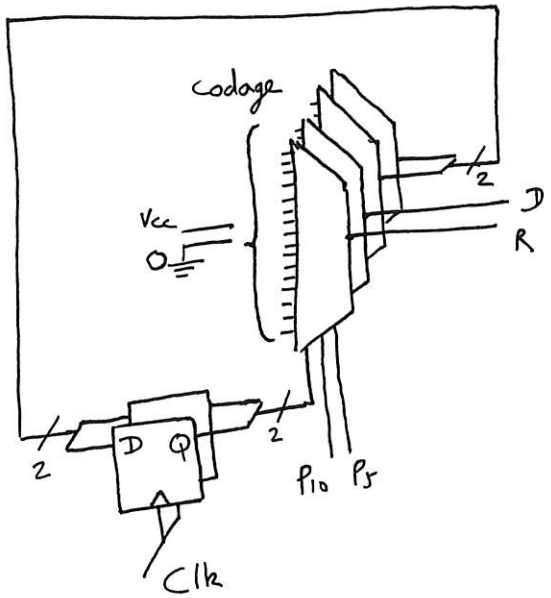


(3.2) L'automate compte quatre états.
On les numérote de 0 à 3, comme représenté dans les carrés accolés aux états. Il y a 12 flèches dans l'automate, ce qui implique qu'il y aura 12 lignes dans la table des transitions, sur les 16 possibles. Les 4 restantes correspondent à des configurations impossibles, celles où P5 et P10 sont simultanément à 1.

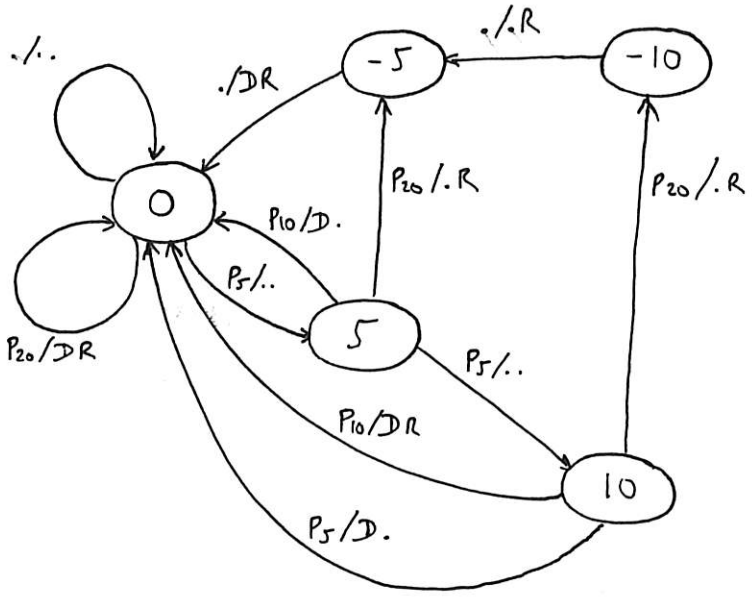
E	P_{10}	P_5	E'	D	R
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	3	0	0
1	0	0	1	0	0
1	0	1	2	0	0
1	1	0	0	1	0
2	0	0	2	0	0
2	0	1	0	1	0
2	1	0	0	1	1
3	0	0	3	0	0
3	0	1	0	1	0
3	1	0	0	1	1

4 bits 4 bits

(3.3) L'automate conserve sa valeur d'état d'un cycle sur l'autre. Il faut donc 2 bascules D pour mémoriser les 2 bits d'état. On se servira également de multiplexeurs pour coder, à la manière d'une ROM, les valeurs de la table de transition. On a donc le schéma suivant.



(3.4) Il faut rajouter, pour chaque état, une transition P_{20}, et gérer le fait de rendre plusieurs pièces de 5 Bz. On indique, dans chaque état, le nombre de Bz déjà insérés.



} Rendu de la monnaie

Question 4

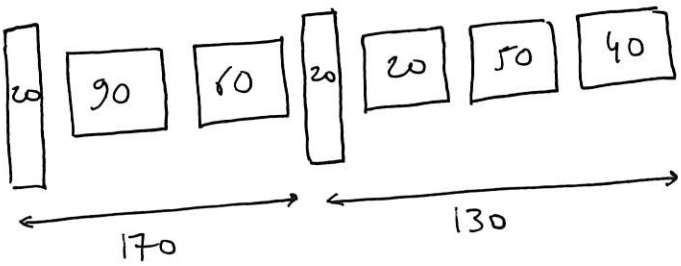
Il est possible de n'utiliser que des registres "caller save".

```
fact2:  mrmovl 4(%esp), %ecx
        irmovl 1, %eax
        subl   %eax, %ecx
        jle    fact2f
        pushl  %ecx
        call   fact2
        popl   %ecx
        mrmovl 4(%esp), %ecx
        pushl  %ecx
        pushl  %eax
        call   mul
        iaddl  8, %esp
fact2f: ret
```

```
factrielle: mrmovl 4(%esp), %ecx
            irmovl -1, %eax
            subl   %eax, %ecx
            jl     factf
            pushl  %ecx
            call   fact2
            popl   %ecx
factf:      ret
```

Question 5

(5.1)

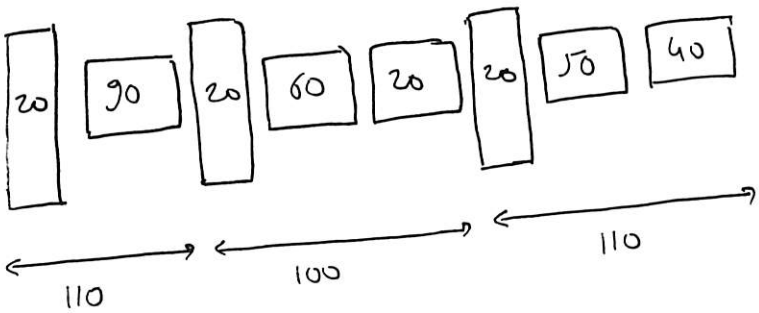


$L_m = 170$ ps

Débit $= \frac{10^{12}}{170} = 5,88$ Gop/s

Latence $= 2 \times 170 = 340$ ps

(5.2)



$L_m = 110$ ps

Débit $= \frac{10^{12}}{110} = 9,09$ Gop/s

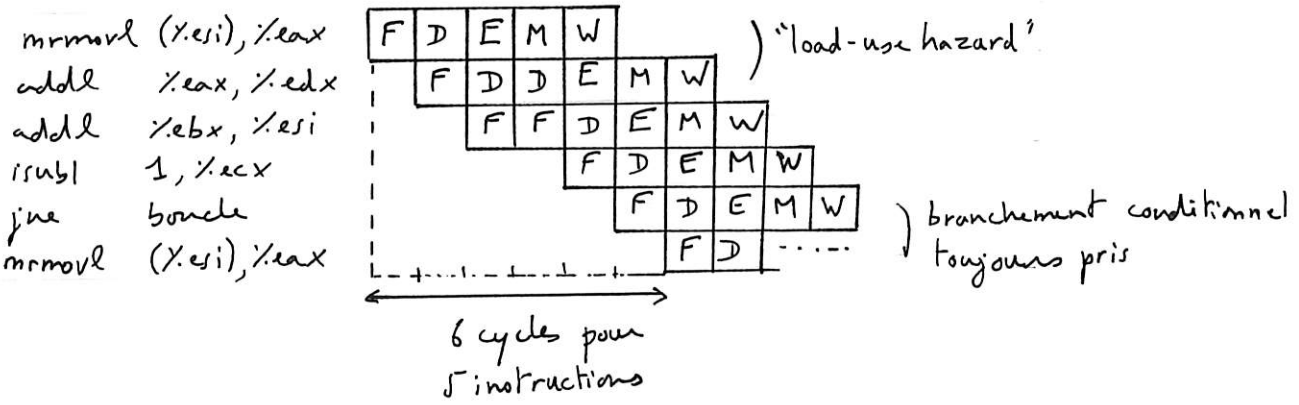
Latence $= 3 \times 110 = 330$ ps

(5.3) Dans le cas précédent, le premier étage ne peut pas être découpé plus. La profondeur minimale est donc égale à 110, et rajouter plus de registres ne ferait qu'augmenter inutilement la latence. La profondeur optimale est donc égale à 3.

Question 6

(6.1) Habituellement, dans une architecture pipe-linée, le "data-forwarding" permet d'éviter les bulles en réinjectant, dans l'étage E de l'instruction suivante, les valeurs calculées à l'étage E par l'instruction précédente.
Un "load-use hazard" se produit lorsque une instruction utilise une valeur chargée à partir de la mémoire par l'instruction qui la précède. Comme la valeur chargée n'est disponible qu'à partir de l'étage M, elle ne peut pas être transmise par "data forwarding" à l'étage E de l'instruction suivante. Il y a donc apparition d'une bulle.

(6.2) Ce programme calcule la somme des "s" valeurs contenues en mémoire à partir de l'adresse "t".



(6.3) Oui. On peut intervertir deux instructions afin d'intercaler une instruction non dépendante de la valeur en train d'être lue, et ainsi éviter le "load-use hazard".

```
mrmovl (%esi), %eax
addl   %ebx, %esi
addl   %eax, %edx
isubl  1, %ecx
jne    boucle
```

(6.4) Voir feuille jointe.

(6.5) Pour mettre en oeuvre l'instruction xchgl rA, rB, il faut faire, au niveau de la banque de registres:
- deux lectures, des registres rA et rB,
- deux écritures, des registres rB et rA.

La banque de registres permet déjà de faire deux écritures simultanées, par exemple pour l'instruction popl.
Cependant, elles font intervenir valE et valM. Il faut donc, à la place de valM, réécrire valA ou valB : si c'est valA qui passe dans valE, ce sera valB, sinon c'est l'inverse. Nous prendrons la première solution.

(6.6) Voir feuille jointe, pour les items concernés. La réécriture de valB à la place de valM n'est pas traitée ici.

— 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
```

, XCHGL

D_icode == LODSL : RESI;

, XCHGL

D_icode == LODSL : RESI;

, XCHGL # (valA+0) ira dans rB

D_icode == LODSL : RESI;

, XCHGL # Mais on n'effectuera pas l'opération mémoire

D_icode == LODSL : REAX;

```
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 };
```

pour XCHGL, valM doit prendre la valeur de valB
avant l'étage write-back