ELECTRONIQUE NUMERIQUE

+

ARCHITECTURE DES ORDINATEURS

CORRIGES

ELECTRONIQUE

NUMERIQUE

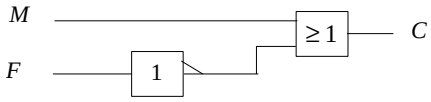
CORRIGES

TD 1 CORRIGE. LOGIQUE COMBINATOIRE 1

4. Synthèse d'un système logique combinatoire

M	F	C
0	0	1
0	1	0
1	0	1
1	1	1

F \ C M	0	1
0	(1	1)
1	0	1



$$C = M + \overline{F}$$

5. Synthèse d'une Fonction logique

a	b	c	s
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

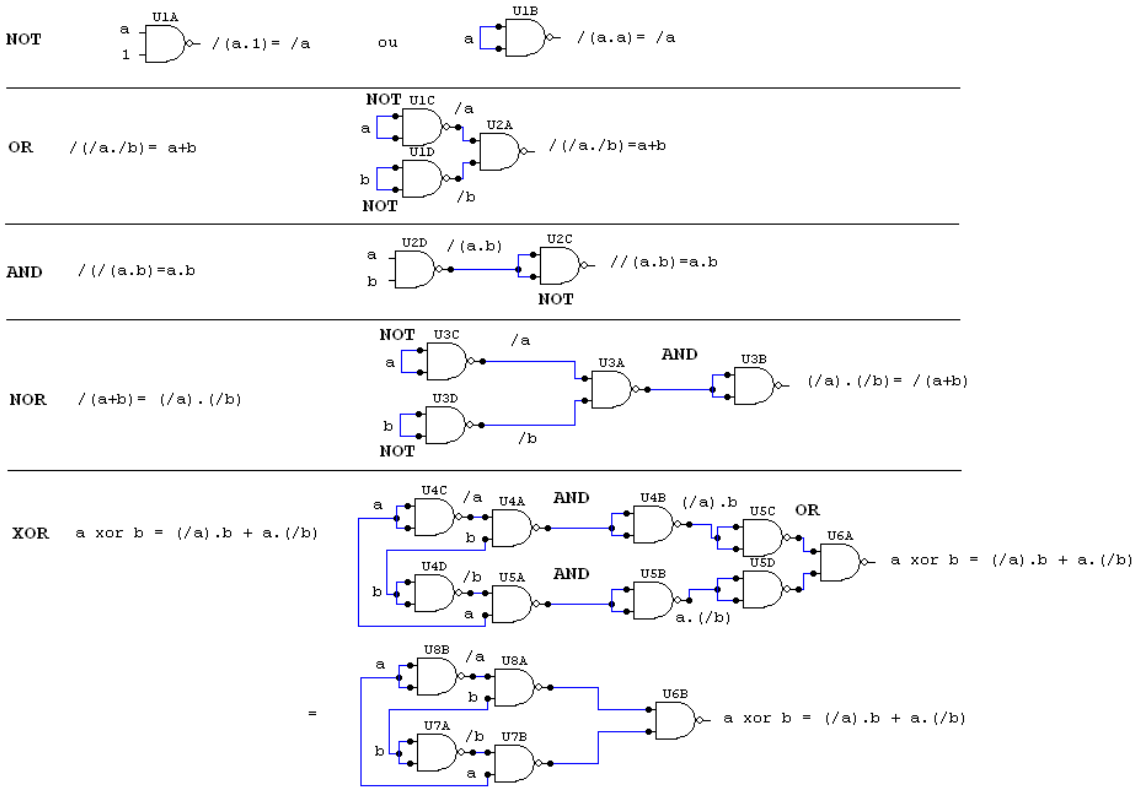
d'où : $s = \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$

Simplification de s *:*

- algébrique : $s = c\left(\overline{a}b + a\overline{b}\right) + ab(c + \overline{c}) = c(a \oplus b) + ab$
- graphique (Karnaugh) : $s = ab + bc + ab$

c \ ab	00	01	11	10
0	0	0	1	0
1	0	1	1	1

6. Réalisation d'une fonction logique à l'aide de portes NAND exclusivement



TD 1 ANNEXE CORRIGE. LOGIQUE COMBINATOIRE 1

1. Analyse d'un système logique combinatoire

- Expression de *S* : $\boxed{S = \overline{(a_0 \oplus b_0)}.\overline{(a_1 \oplus b_1)}.\overline{(a_2 \oplus b_2)}.\overline{(a_3 \oplus b_3)}}$ ou : $S = (a_0 \odot b_0).(a_1 \odot b_1).(a_2 \odot b_2).(a_3 \odot b_3)$

- Rôle du montage comparateur :

Détecter l'égalité des 2 mots binaires de 4 bits : $A = a_3a_2a_1a_0$ et : $B = b_3b_2b_1b_0$

2. Simplification d'une fonction logique

S *AB* \ *CD*	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$S = \overline{B}C + B\overline{C}$

$\boxed{S = B \oplus C}$

Corrigé TD Electronique n°1

I Algèbre de Boole – Logique Combinatoire 1

TD1 Logique Combinatoire1

1.a) $S = \underbrace{P_L \cdot P_P}_{1} + \underbrace{\bar{P}_L \cdot \bar{P}_P}_{2} + \underbrace{\bar{P}_L \cdot P_P}_{3}$

(*) → pour avoir le moins de circuits logiques possibles

$1+3 = P_P \cdot (P_L + \bar{P}_L) = P_P$ (1)

b) b1) avant de câbler il faut simplifier la fonction logique

$S = \underbrace{P_L \cdot P_P + \bar{P}_L \cdot P_P}_{1+3} + \underbrace{\bar{P}_L \cdot \bar{P}_P}_{2}$

autre méthode de simplification algèbre de Boole

$S = P_L + P_L P_P$

$S = \bar{\bar{S}} = \overline{\bar{P}_L + P_L P_P}$

$= \overline{P_L \cdot (\bar{P}_L + \bar{P}_P)}$

$= \overline{P_L \cdot \bar{P}_P}$

$S = P_P + \bar{P}_L$

ou une fonction logique n'est pas changée si on rajoute un terme déjà existant. (car $a + a = a$)

→ $S = \overline{P_L \cdot P_P}^{1} + \overline{\bar{P}_L \cdot \bar{P}_P}^{2} + \overline{\bar{P}_L \cdot P_P}^{3} + \overline{\bar{P}_L P_P}^{3}$

rajoutons 3 à 2

$2+3 = \bar{P}_L \bar{P}_P + \bar{P}_L P_P = \bar{P}_L (P_P + \bar{P}_P) = \bar{P}_L$

→ $S = P_P + \bar{P}_L$

b2)

P_P \ P_L	0	1
0	1	0
1	1	1

→ $\bar{P}_L$

→ P_P

→ $S = \bar{P}_L + P_P$

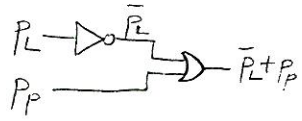
si on regroupe les 0 : (→ $\bar{S}$ obtenue)

P_P \ P_L	0	1
0	1	0
1	1	1

$\bar{S} = P_L \cdot \bar{P}_P$

→ $S = \overline{P_L \cdot \bar{P}_P} = \bar{P}_L + P_P$

c)



dans les tableaux de Karnaugh à 2 variables, les coins ne constituent pas des cases adjacentes comme pour les tableaux de Karnaugh d'ordre supérieur.

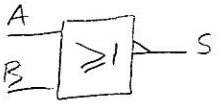
P_P \ P_L	0	1
0	①	
1		②

cases ① et ② non adjacentes.

2. a) $f = A \text{ NOR } B$ car $f = \overline{(A+B)} = \bar{A} \cdot \bar{B}$ on peut le voir avec la table de vérité (ou mieux Karnaugh & simplification)

A	B	$S = \overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

b)



B \ A	0	1
0	1	0
1	0	0

$S = \bar{A} \cdot \bar{B} = \overline{A+B}$

(*) $S = P_L P_P + \bar{P}_L$ (non simplifié)

→ $S = P_L P_P + \bar{P}_L \cdot P_P + \bar{P}_L \cdot \bar{P}_P$ (idempotence)

→ $S = P_L P_P + \bar{P}_L P_P + \bar{P}_L \bar{P}_P + \bar{P}_L \cdot P_P = \underline{\bar{P}_L + P_P}$

on a aussi : $\bar{S} = P_L \cdot \bar{P}_P \rightarrow S = \underline{\bar{P}_L + P_P}$ ← simplifié

4 a) $S = \overline{C \oplus D} = C\bar{D} + \bar{C}.\bar{D}$ (fonction coïncidence)

S (AB \ CD)	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	X	X	X	X
10	1	0	X	X

C, D, A, B

$S = \bar{C}\bar{D} + CD$

b) $\bar{S} = C \oplus D$

S (AB \ CD)	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	X	X	X	X
10	1	0	X	X

C, D, A, B

$\bar{S} = \bar{C}D + C\bar{D}$

3 a) NOT : $A \rightarrow \bar{A}$; A — $\overline{A.A} = \bar{A}$ ou A, 1 — $\overline{A.1} = \bar{A}$

b) OR : $A, B \rightarrow A+B$: A, B — $\bar{A}$, $\bar{B}$ — $\overline{\bar{A}.\bar{B}} = A+B$

c) AND : $A, B \rightarrow A.B$: A, B — $\overline{A.B}$ — AB

d) XOR : $A, B \rightarrow A \oplus B = A.\bar{B} + \bar{A}.B$

A, B, $\bar{A}$, $\bar{B}$, $\overline{\bar{A}.B}$, $\overline{A.\bar{B}}$

$\overline{(A+\bar{B}).(\bar{A}+B)}$
$= \overline{(A+\bar{B})} + \overline{(\bar{A}+B)}$
$= \bar{A}.B + A.\bar{B}$

e) NOR : $A, B \rightarrow \overline{A+B}$

A, B, $\bar{A}$, $\bar{B}$, $\overline{\bar{A}.\bar{B}} = A+B$, $\overline{A+B}$

ou : $\bar{A}.\bar{B}$

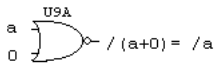
A, B, $\bar{A}$, $\bar{B}$, $\overline{\bar{A}\,\bar{B}} = A+B$, $\overline{A+B}$

TP 1 CORRIGE. LOGIQUE COMBINATOIRE 1

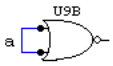
4. Etude Expérimentale

4.3. Facultatif - Réalisation d'une fonction logique à l'aide de portes NOR exclusivement

NOT

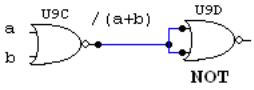


ou



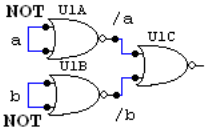
/(a+a) = /a

OR /(/(a+b) = a+b



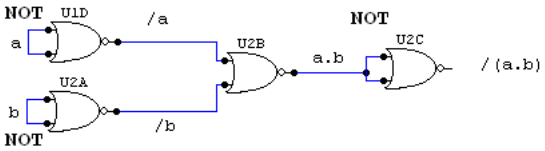
//(a+b) =a+b

AND /[(/a)+(/b)] = a.b

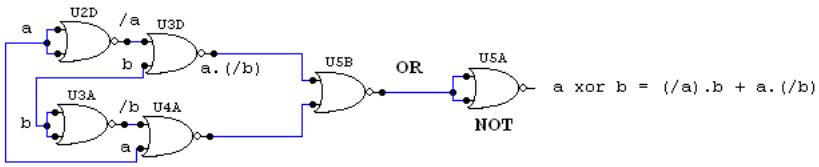


/[(/a)+(/b)] = a.b

NAND /a + /b = /(a.b)



XOR a xor b = (/a).b + a.(/b)



a xor b = (/a).b + a.(/b)

TD 2 CORRIGE. LOGIQUE COMBINATOIRE 2

Transmission

3. Construction d'un code détecteur d'erreur

a) $k_1 = \overline{m_2 \oplus m_1 \oplus m_0}$

b) $d = \overline{m_2' \oplus m_1' \oplus m_0' \oplus k_1}$

4. Système de transmission numérique avec correction d'erreur

Rappels des points importants de l'énoncé :

- *Code de Hamming*
- *4 éléments binaires correspondant à un chiffre du système décimal* ***m_1, m_2, m_3, m_4.***
- *3 éléments binaires de contrôle de parité* ***k_1, k_2, k_3.***
- *La structure du mot :*

Position du bit	1	2	3	4	5	6	7
Nom du bit	k_1	k_2	k_3	m_1	m_2	m_3	m_4
Exemple	0	1	1	0	0	1	0

L'exemple numérique donné dans le tableau ci-dessus correspond à N=2 dans le code de Hamming !

- *3 tests de parité*
  - *T_1(test sur k_1) se fait sur les bits : 1, 4, 5, 7 (c-à-d : k_1, m_1, m_2, m_4)*
  - *T_2(test sur k_2) se fait sur les bits : 2, 4, 6, 7 (c-à-d : k_2, m_1, m_3, m_4)*
  - *T_3(test sur k_3) se fait sur les bits : 3, 5, 6, 7 (c-à-d : k_3, m_2, m_3, m_4)*
- *Test de parité* ***paire*** *(le résultat du test est égal à 0 si le nombre de 1 dans la zone considéré est pair, ce qui revient à dire qu'il n'y a pas d'erreur !)*
- *($T_3T_2T_1$) est un mot en code BCD qui donne en décimal la position du bit erroné*
- *1 seul bit au maximum peut-être erroné et l'erreur est cantonné au niveau des bits d'information « m_i » (pas d'erreur sur les k)*

Questions

4.1. Donner le schéma du dispositif « émetteur » permettant de générer les bits k_1, k_2 et k_3 :
On va utiliser le tableau du code de Hamming pour établir les expressions des bits de contrôle de parité k_i en fonction des bits d'information m_1, m_2, m_3 et m_4. On trace les tableaux de Karnaugh des bits de contrôle de parité. On remplit les cases avec les données de Hamming.

K_1

m_1m_2 \ m_4	0	1
00	0	1
01	1	0
11	X^0	X^1
10	1	0

k_2

m_1m_3 \ m_4	0	1
00	0	1
01	1	0
11	X	X^1
10	1	0

k_3

m_2m_3 \ m_4	0	1
00	0	1
01	1	0
11	0	1
10	1	0

Explication de remplissage de tableau de Karnaugh de k_1 : on sait que k_1 est regroupé avec les bits numéros 4, 5 et 7 (le test de parité T_1 sur k_1 se fait sur les bits 1, 4, 5 et 7), c'est-à-dire m_1, m_2 et m_4. Pour la 1ère case (1ère ligne et 1ère colonne) on repère dans le code de Hamming la ligne où $m_1m_2m_4 = 000$ et on inscrit dans la case la valeur de k_1 correspondante, pour la 2ème case (1ère ligne et 2ème colonne) on repère la ligne où $m_1m_2m_4 = 001$ et on inscrit dans la case la valeur de k_1 correspondante, ainsi de suite… Lorsque vient le tour des 5ème et 6ème cases, on doit rechercher les lignes du code de Hamming où $m_1m_2m_4 = 110$ et 111, or ces lignes n'existent pas ! Alors dans ce cas on peut se permettre d'y inscrire X et lui attribuer ensuite la valeur (0 ou 1) qui convient la mieux pour une plus grande simplification de l'expression recherchée.

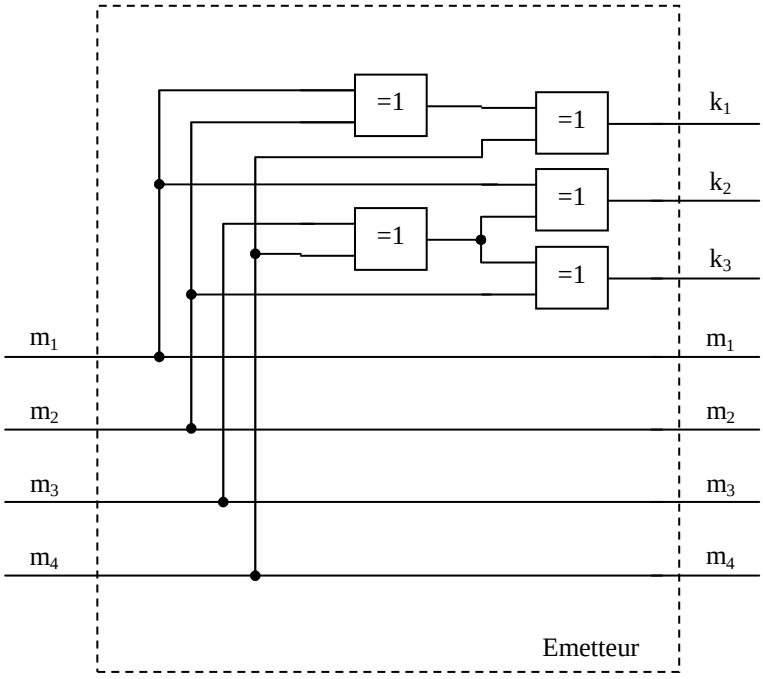
On avait vu que les 1 qui se disposent en diagonal dans un tableau de Karnaugh correspondent à une expression qui relie les variables avec l'opérateur OU Exclusif. Donc ici on a intérêt à remplacer le X de la case 5 par un 0 et le X de la case 6 par un 1.

D'où $\mathbf{k_1 = m_4 \oplus m_2 \oplus m_1}$

Il en va de même pour k_2 et le résultat est semblable : $\mathbf{k_2 = m_4 \oplus m_3 \oplus m_1}$
Toutes les combinaisons possibles de $m_2m_3m_4$ étant disponibles dans le code de Hamming le tableau de Karnaugh de k_3 ne contient pas de X (vérifiez !), on a donc :

$$\mathbf{k_3 = m_4 \oplus m_3 \oplus m_2}$$

Les expressions obtenues nous permettent de proposer le schéma du dispositif émetteur permettant de générer les bits de contrôle de parité.



4.2. Donner le schéma du dispositif « récepteur » permettant de générer les bits T_1, T_2 et T_3 :
On va tracer les tableaux de Karnaugh des bits de détection d'erreur T_1, T_2 et T_3 afin d'obtenir leur expression en fonction des k_i et m_j. On sait que T_1 dépend de k_1, m_1, m_2, m_4, T_2 dépend de k_2, m_1, m_3, m_4 et T_3 dépend de k_3, m_2, m_3, m_4. On va donc remplir les tableaux de Karnaugh pour chaque T_i en fonction des 4 bits auxquels il est lié ; et on mettra 0 pour chaque combinaison de bits contenant un nombre pair de 1, et on mettra 1 pour le cas inverse (ce qui correspond à une erreur car on est dans le contrôle de parité paire)

T_1

$k_1m'_1$ \ $m'_2m'_4$	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

T_2

$k_2m'_1$ \ $m'_3m'_4$	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

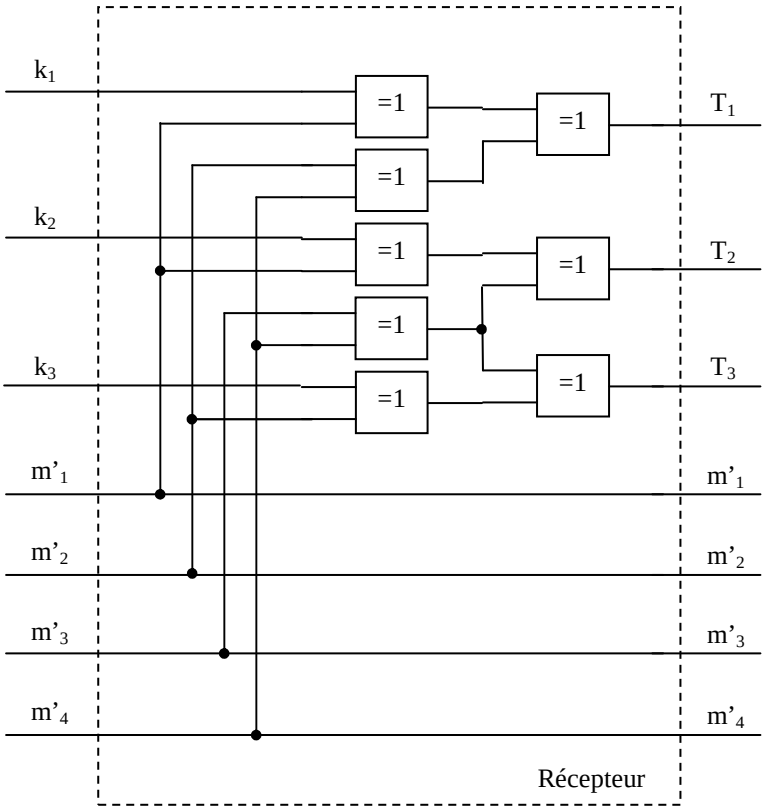
T_3

$k_3m'_2$ \ $m'_3m'_4$	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

On a encore une structure qui donne ***XOR***.

$T_1 = k_1 \oplus m'_1 \oplus m'_2 \oplus m'_4$
$T_2 = k_2 \oplus m'_1 \oplus m'_3 \oplus m'_4$
$T_3 = k_3 \oplus m'_2 \oplus m'_3 \oplus m'_4$

Les m_i à la sortie de l'émetteur ayant « voyagé » lors de la transmission ont subi des perturbations et sont devenus m'_i.



4.3. Dispositif simple réalisant la correction du bit erroné (1 seul bit erroné au maximum parmi uniquement les m_i). Nous savons que $(T_2T_1T_0)_2$ indique le numéro de l'élément binaire erroné, donc celui qu'il faut corriger. Si par exemple $(T_2T_1T_0)_2 = (101)_2$ ça veut dire que c'est l'élément numéro 5 qui est erroné (m_2). La correction consiste simplement à changer l'élément binaire faux à l'aide d'un circuit complément.

Pour transmettre l'information du numéro de l'élément binaire erroné du code BCD en décimal il faut un décodeur BCD-Décimal. Donc s'il y a erreur, c'est la sortie correspondant au numéro du bit erroné qui sera à l'état haut (1) toutes les autres resteront à l'état bas (0). Dans ce cas il faut rechercher l'opération qui inverse l'entrée lorsque la référence (la deuxième entrée connecté à la sortie du décodeur) est à 1 (donc si erreur) et qui la recopie lorsque la référence est à 0 (pas d'erreur). C'est le « OU exclusif » qui est soit un opérateur transparent soit un opérateur inverseur suivant sa commande (ou référence).

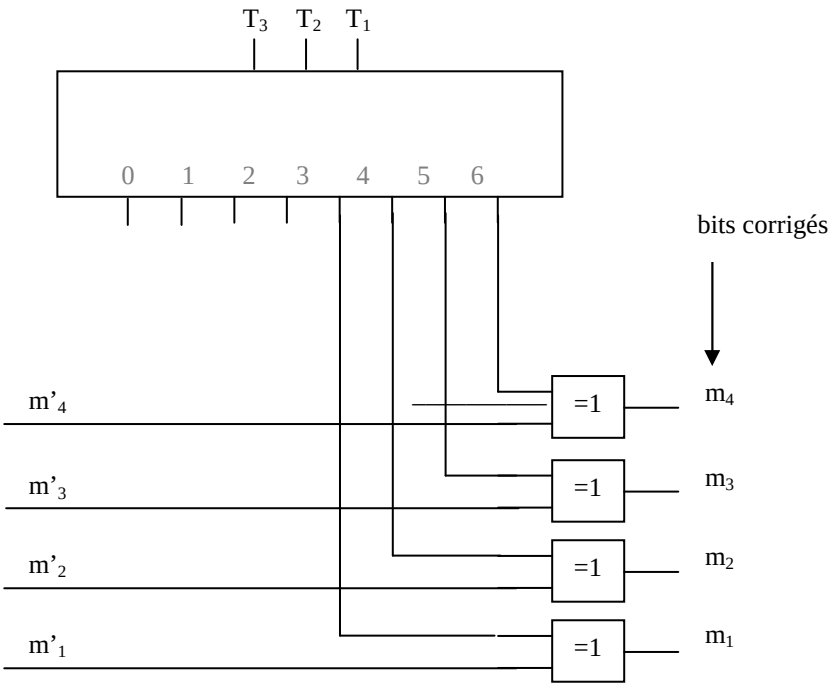
En effet, soit E = entrée et R = Référence, la sortie S = E $\oplus$ R obéit au tableau de Karnaugh suivant :

S

E \ R	0	1
0	0	1
1	1	0

On voit bien sur ce tableau que la 1ère colonne (la référence est égale à 0) est identique à E donc c'est une opération transparente et la 2ème colonne est bien l'inverse de E (R = 1).

D'où le circuit correcteur :



Corrigé TD d'électronique n°2

Logique combinatoire (2)

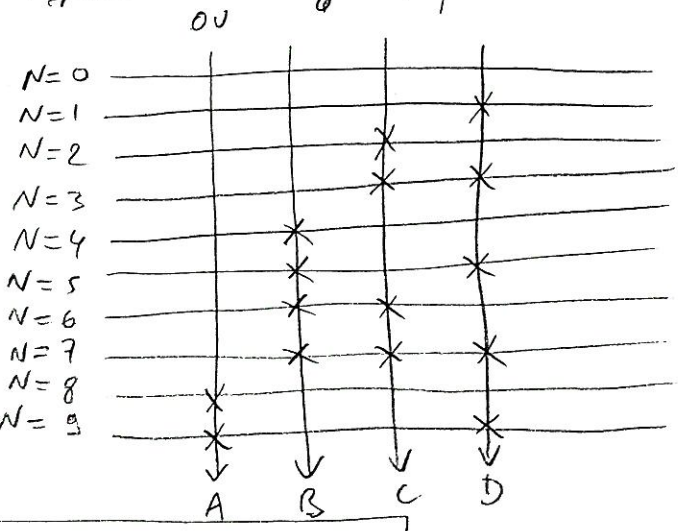
TD2 — Logique Combin.2

1

Table de codage

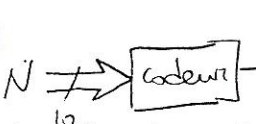
N	ABCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Représentation symbolique



1 seule entrée active à la fois ≡ Codeur ≡ Matrice OU

2



≡ de l'exercice précédent (1)

↑ 1 seule entrée active (1 parmi 10 ici)

3

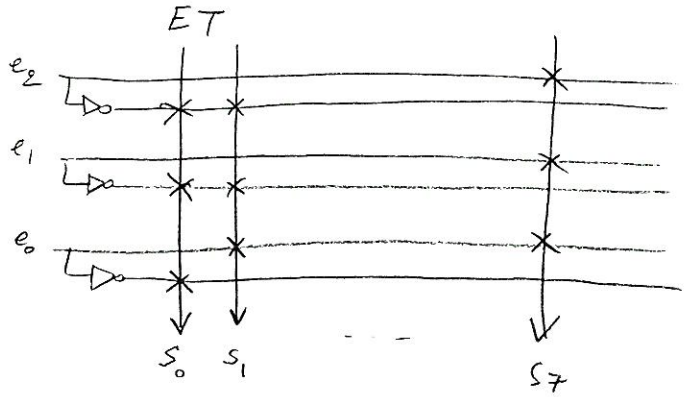
1 seule sortie active à la fois ≡ Décodeur ≡ Matrice ET

Table de Décodage

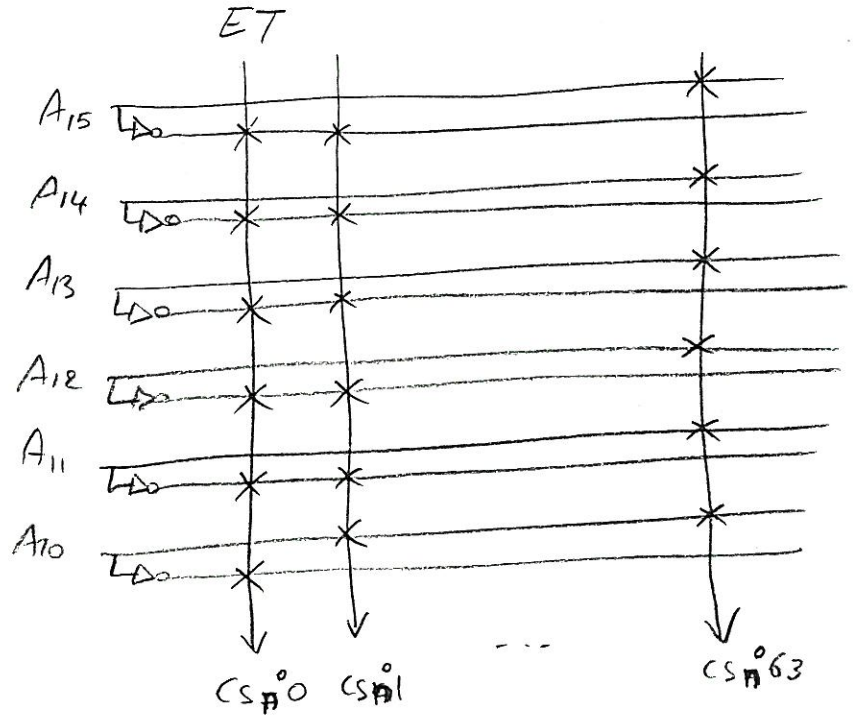
e_2 e_1 e_0	S_i
0 0 0	$S_0 = 1$ ($S_i = 0$ de $i = 1$ à 7)
0 0 1	$S_1 = 1$ ($S_0 = 0$; $S_i = 0$ de $i = 2$ à 7)
0 1 0	$S_2 = 1$ ($S_0 = S_1 = 0$; $S_i = 0$ de $i = 3$ à 7)
0 1 1	$S_3 = 1$
1 0 0	$S_4 = 1$
1 0 1	$S_5 = 1$
1 1 0	$S_6 = 1$
1 1 1	$S_7 = 1$

etc...

Représentation symbolique



4



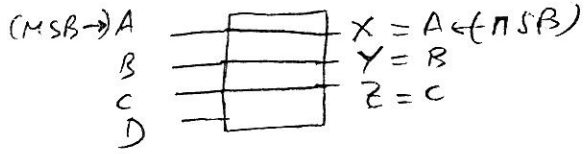
5

Multiplier par 2 ≡ Décaler le mot binaire vers la gauche
Diviser par 2 ≡ ——— droite

car. N : [... | b_i | ...] = ... + $b_i \cdot 2^i$ + ...

$N \times 2 \longrightarrow b_i \cdot 2^{i+1} = 2 \times b_i \cdot 2^i$ ($b_i \cdot 2^{i+1}$, 2^i)

Le bit introduit par décalage étant à 0. ex : 2 sur 4 bits = 0010 → (b_i) 4 = 0100 et 1 = 0...
→ Pas besoin de faire la table de transcodage, il suffit de prendre les bits décalés de 1 cran :



6

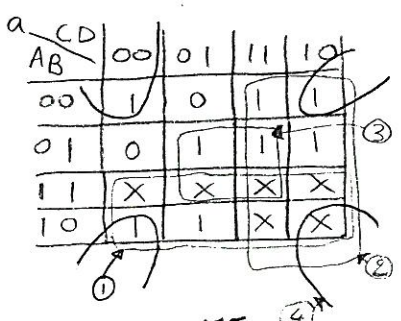
a) idem que ~~l'ex.~~ 7. mais les 6 dernières lignes sont indéfinies (X) pour a b c d e f g

	A B C D	a b c d e f g
0	0 0 0 0	1 1 1 1 1 1 0
	idem	idem
9	1 0 0 1	1 1 1 1 0 1 1
A	1 0 1 0	X X X X X X X
B	1 0 1 1	X X X X X X X
C	1 1 0 0	X X X X X X X
D	1 1 0 1	X X X X X X X
E	1 1 1 0	X X X X X X X
F	1 1 1 1	X X X X X X X

(6) ~~abis~~ abis).

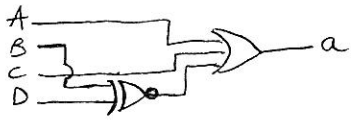
N	N(BCD) A	B	C	D	N(7 segments) a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
A	1	0	1	0	1	1	1	0	1	1	1
B	1	0	1	1	0	0	1	1	1	1	1
C	1	1	0	0	1	0	0	1	1	1	0
D	1	1	0	1	0	1	1	1	1	0	1
E	1	1	1	0	1	0	0	1	1	1	1
F	1	1	1	1	1	0	0	0	1	1	1

b)

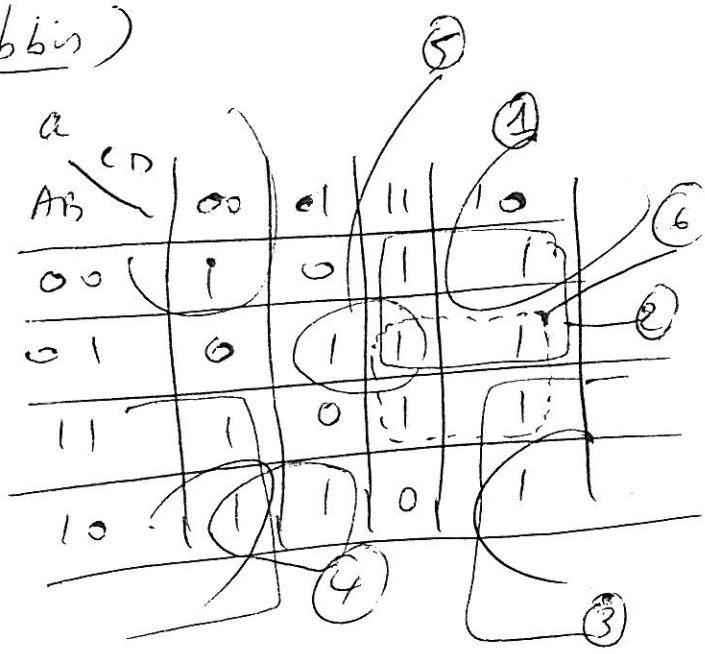


$a = A + C + BD + \overline{B}\,\overline{D} = A + C + \overline{B \oplus D}$

① ② ③ ④



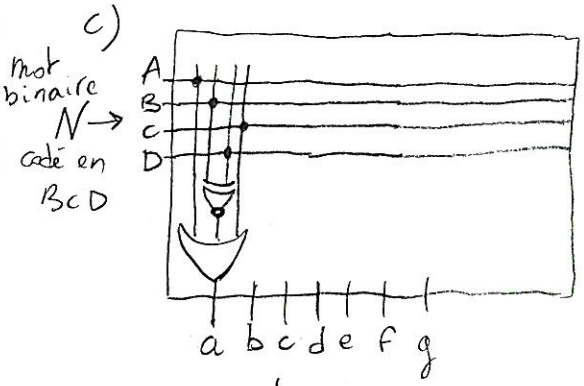
bbis)

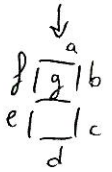


$a = \overline{B}\,\overline{D} + \overline{A}C + A\overline{D} + A\overline{B}\overline{C} + \overline{A}BD + BC$

0 ① ② ③ ④ ⑤ ⑥

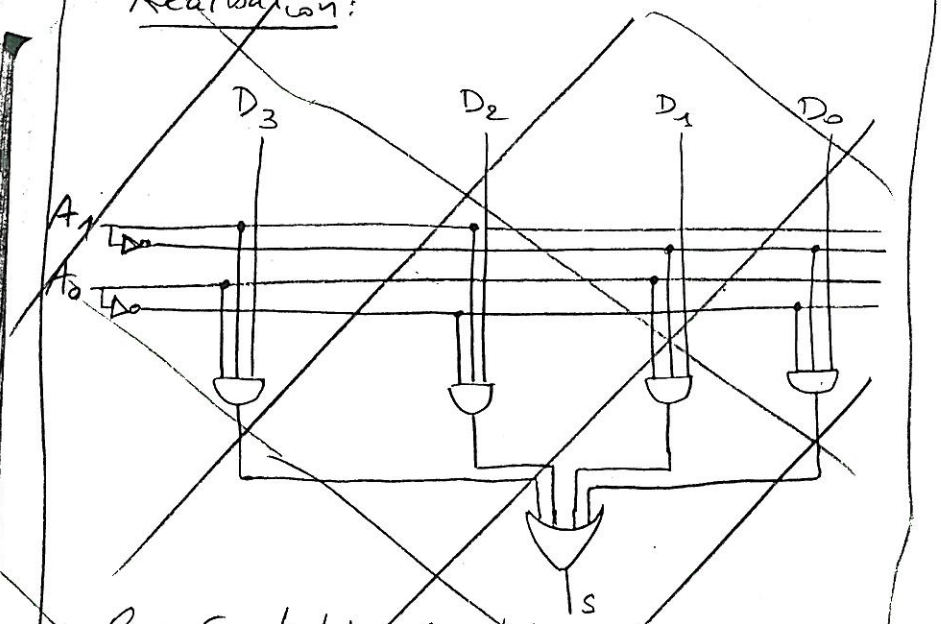
c)



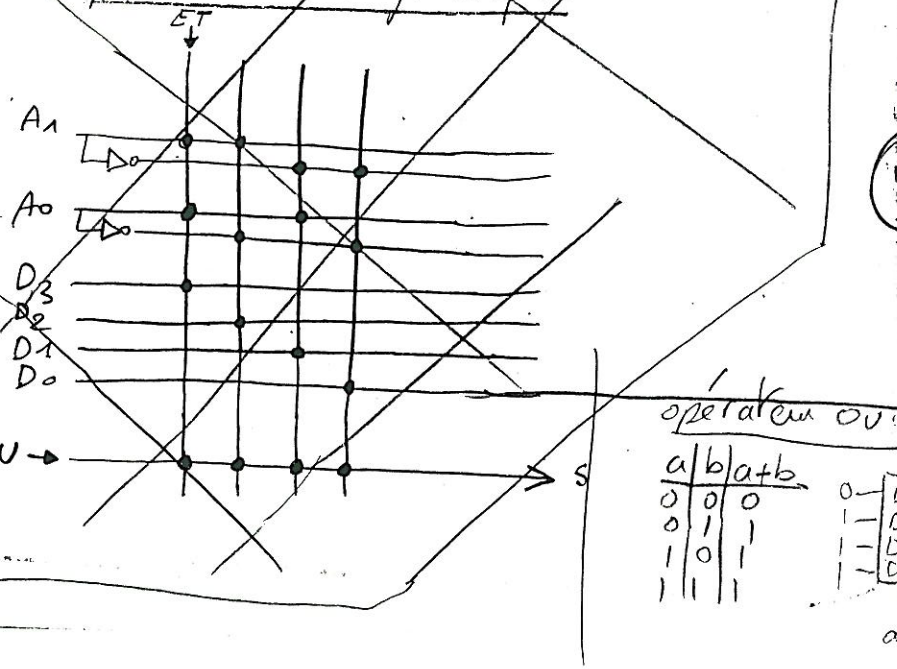


(7)

4.4.2 Réalisation :



Représentation symbolique



opérateur OU :

a	b	a+b
0	0	0
0	1	1
1	0	1
1	1	1

8

9

12.4.4.3 Applications :

12.4.4.3.1 Sélection d'un bit parmi plusieurs bits

mais aussi :

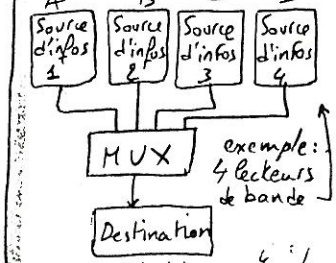
Sélection d'un mot binaire parmi plusieurs mots binaires

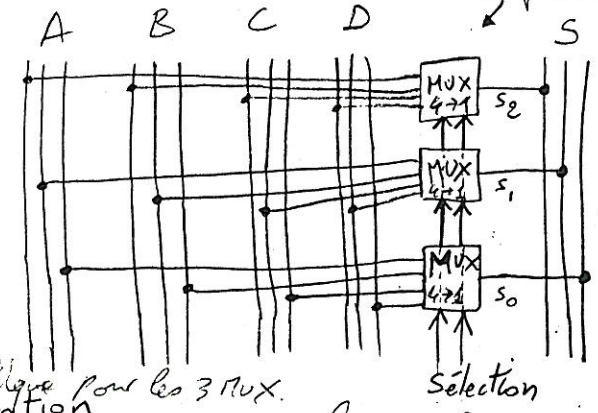
ex : sélection d'un mot de 3 bits parmi 4 mots de 3 bits :

⟹ il faut autant de multiplexeurs qu'il y a de bits dans le mot (ici 3 multiplexeurs) :

4 mots de données :

A, B, C, D





La sélection est identique pour les 3 MUX.

12.4.4.3.2 Génération

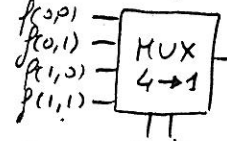
Matérialisation d'une fonction logique :

Toute fonction logique peut se mettre sous forme canonique

→ ex : à 2 variables : $f(a,b) = \bar{a}\bar{b}.f(0,0) + \bar{a}b f(0,1) + a\bar{b} f(1,0) + ab f(1,1)$

avec : $f(i,j)$ = valeur particulière de la fonction logique f lorsque $a = i$ et $b = j$

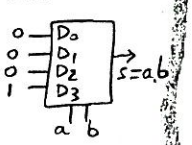
→ utilisation d'un multiplexeur à 4 entrées de données, donc 2 entrées d'adresse :



ab =
$\bar{a}\bar{b}.0 + \bar{a}b.0$
$+ a\bar{b}.0 + ab.1$

ex : opérateur ET :

a	b	S = ab
0	0	D0 = 0
0	1	D1 = 0
1	0	D2 = 0
1	1	D3 = 1



→ Le multiplexeur est un opérateur programmable.

Pour une fonction de n variables, il faut un multiplexeur à 2^n entrées de données donc n entrées d'adresse : MUX $2^n \to 1$

(10)

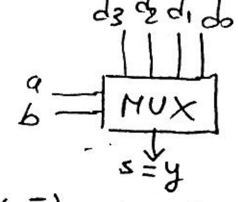
On ne fait pas les regroupements pour pouvoir utiliser un multiplexeur :

On a :

$y = ab\bar{c}\bar{d} + a\bar{b}\bar{c}\bar{d} + \bar{a}\bar{b}cd + \bar{a}bcd + \bar{a}bc\bar{d} + \bar{a}\bar{b}c\bar{d}$

On choisit par exemple a b en entrées d'adresse :

a b : a noté a_1 (MSB), b noté a_0 (LSB)



$y = \bar{a}\bar{b}.(cd + c\bar{d}) + \bar{a}.b.(cd + c\bar{d}) + a\bar{b}(\bar{c}.\bar{d}) + ab(\bar{c}.\bar{d})$

$= \sum d_i . m_i$ $\quad y = \bar{a}\bar{b}c + \bar{a}bc + a\bar{b}\bar{c}\bar{d} + ab\bar{c}\bar{d}$

Sur d_0 : on doit appliquer : $cd + c\bar{d} = c$

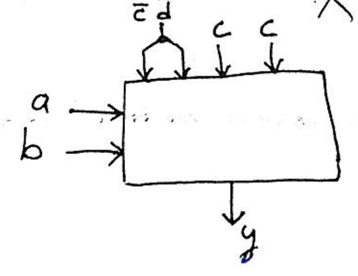
— d_1 : ——— : $cd + c\bar{d} = c$

— d_2 : ——— : $\bar{c}\bar{d}$

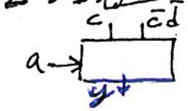
— d_3 ——— : $\bar{c}\bar{d}$

avantage : réduction du nombre de variables :

on est passé d'un problème à 4 variables abcd à un problème à 2 variables ~~ab~~ . cd



Vu que b se simplifie, on pourrait même utiliser un MUX 2 → 1 : $y = c\bar{a} + \bar{c}\bar{d}a$



(6)

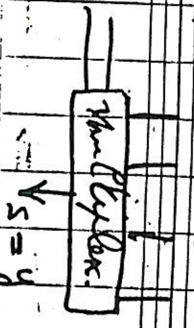
On pourrait toujours faire comme dans l'exercice précédent (6) utiliser 4 variables pour les adresses du MUX (le MUX est alors un MUX 16→1 avec en entrées de données des cst des 1, mais pour réduire la table du MUX, on peut entrer des données variables)

Soit un pb donné : voir sa table de Karnaugh →

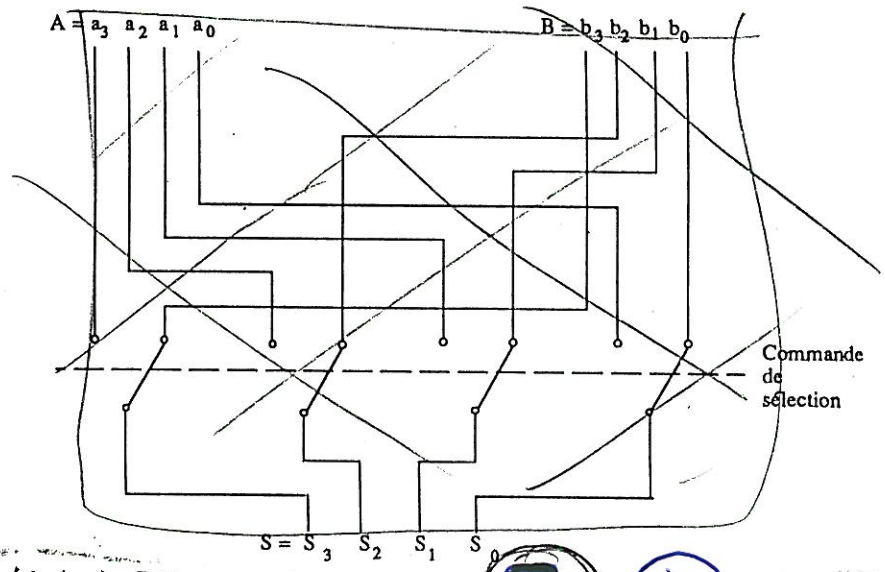
(y) ab / cd

0	0	1	1
0	0	0	0
1	1	0	0
1	1	0	0

On peut utiliser un multiplexeur qui donne en sortie (y) voulue.



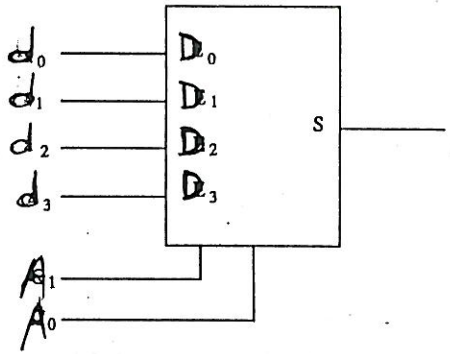
$y = c\bar{a} + \bar{c}\bar{d}a$



Conversion parallèle-série

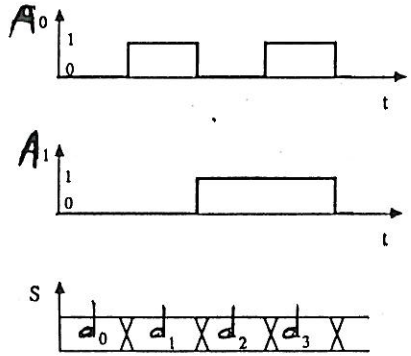
Soit un mot binaire $D = d_3 d_2 d_1 d_0$ disponible en mode parallèle, c'est-à-dire sur quatre fils, chaque fil étant affecté à un élément binaire du mot.

Pour transmettre les éléments binaires en série, c'est-à-dire les uns à la suite des autres sur un seul fil, il faut d'abord transmettre d_0, puis d_1 puis d_2 et enfin d_3. Ceci revient à sélectionner (ou à aiguiller) l'un des éléments binaires de D sur le fil unique de sortie série. Le multiplexeur est capable d'effectuer cette tâche si les combinaisons correspondantes sont placées successivement sur les commandes de sélection.



Dans le premier temps il faut que $A_1 = A_0 = 0$ pour que $S = D_0 = d_0$.
Ensuite A_0 passe à 1 ce qui impose $S = D_1 = d_1$.
Puis $A_1 = 1$ et $A_0 = 0$ d'où $S = D_2 = d_2$.
Et enfin $A_1 = A_0 = 1$ alors $S = D_3 = d_3$.

D'où le chronogramme :



Génération de fonctions

Toute fonction logique combinatoire peut se mettre sous forme canonique (*cf.* § 2.2.1, théorèmes d'expansion de Shannon). Par exemple, une fonction de deux variables A et B se développe suivant l'expression

$$f(A,B) = \overline{A}\,\overline{B} f(0,0) + \overline{A} B f(0,1) + A \overline{B} f(1,0) + A B f(1,1)$$

où $f(i,j)$ est la valeur particulière de la fonction logique lorsque $A = i$ et $B = j$.

Un multiplexeur à quatre entrées, donc deux commandes, délivre une sortie S reliée aux commandes C_1 C_0 par la relation

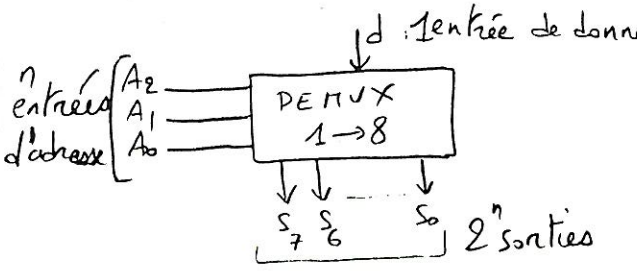
$$S = \overline{C_1}\,\overline{C_0} E_0 + \overline{C_1} C_0 E_1 + C_1 \overline{C_0} E_2 + C_1 C_0 E_3$$

En comparant les deux relations précédentes, il est facile de voir qu'il est possible de réaliser toutes les fonctions de deux variables en identifiant S et $f(A,B)$, C_1 et A, C_2 et B, les valeurs des entrées du multiplexeur et celles de la fonction.

Les entrées de sélection du multiplexeur sont alors les variables de la fonction, et les entrées du multiplexeur permettent de sélectionner la fonction à réaliser.

4.5 Le Démultiplexeur

12.4.5.1 Définition : aiguille 1 donnée sur 1 parmi 2^n sorties :



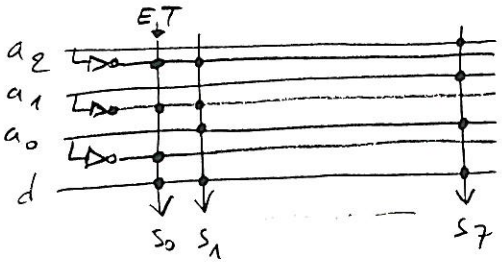
$A_2 A_1 A_0$	S_i
0 0 0	$S_0 = d$
0 0 1	$S_1 = d$
1 1 1	$S_7 = d$

Les sorties non aiguillées sont non activées par convention on les suppose au niveau 0 (en fait, ça peut être 0 ou 1 suivant la technique utilisée par le constructeur)

12.4.5.2 Equation des sorties : $S_i = m_i . d$ ← minterme

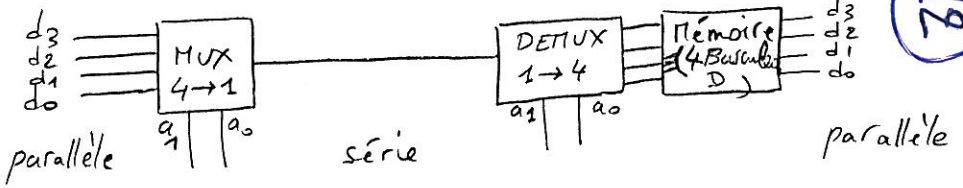
ex : $S_3 = \bar{a}_2 a_1 a_0 . d$

12.4.5.3 Réalisation : (Représentation Symbolique)



12.4.5.4 Applications

12.4.5.4.1 Conversion série / parallèle

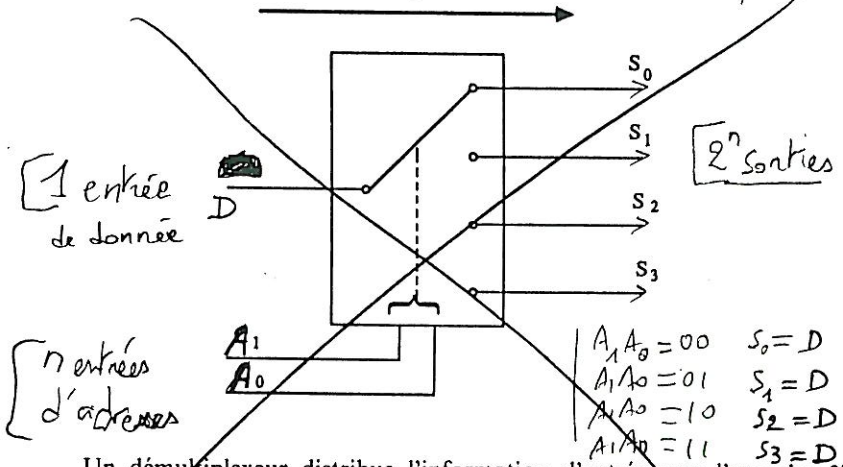


12.4.5.4.2 Affichage multiplexé

Soit 4 chiffres à afficher ; on peut afficher les chiffres l'un après l'autre très vite pour donner l'impression de simultanéité à l'oeil

12

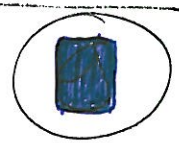
Le démultiplexeur aiguille 1 Donnée sur 1 parmi 2^n sorties



Un démultiplexeur distribue l'information d'entrée vers l'une des 2^n sorties, la sélection de la sortie concernée étant effectuée par n variables de commande. Les autres sorties sont alors dans un état de repos.

Lorsque l'entrée est toujours égale à 1, le démultiplexeur fonctionne comme un décodeur binaire.

12.4.6 3.10. LES CIRCUITS INTÉGRÉS ARITHMÉTIQUES



13

12.4.6.1 3.10.1. L'ADDITIONNEUR

C'est le circuit réalisant l'addition de deux nombres binaires. La table d'addition de deux nombres à un élément binaire est la suivante :

a \ b	0	1
0	0	1
1	1	10

r Σ

Le résultat de l'opération comporte deux parties :

— la somme Σ :

a \ b	0	1
0	0	1
1	1	0

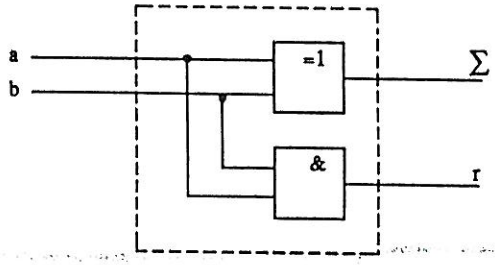
$\Sigma = a \oplus b$

— et la retenue r :

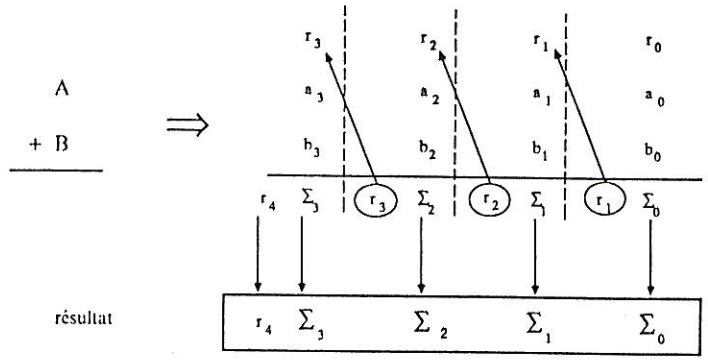
a \ b	0	1
0	0	0
1	0	1

r = ab

Le circuit élémentaire réalisant cette opération est le **demi-additionneur :**



Pour effectuer une addition de deux nombres binaires à plusieurs chiffres, une première technique consiste à additionner successivement les chiffres de même poids avec éventuellement la retenue de l'addition précédente.

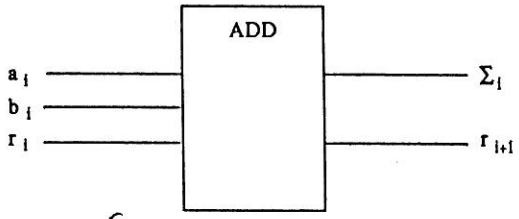


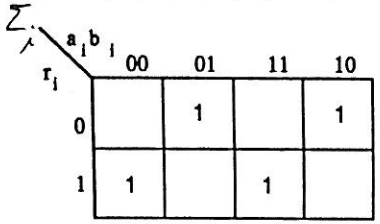
La structure de l'additionneur de deux nombres est alors répétitive. Une cellule élémentaire peut donc être utilisée pour chaque poids. Elle est appelée **additionneur complet.** L'addition globale est réalisée par la mise en cascade des cellules au sens des retenues.

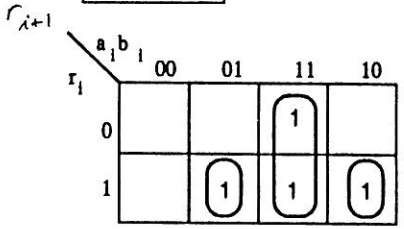
L'additionneur complet est défini par la table de vérité ci-après :

somme = $a_i + b_i + r_i$ (plus)

a_i	b_i	r_i	r_{i+1}	Σ_i
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



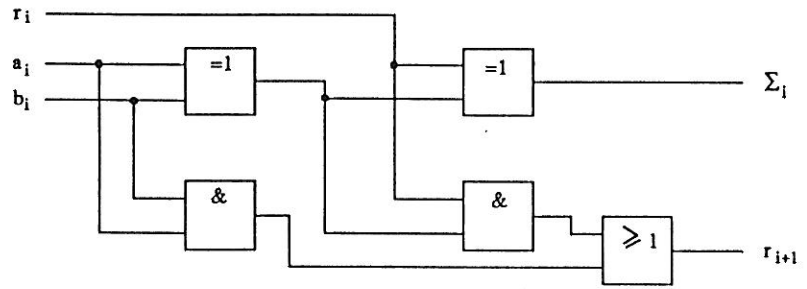




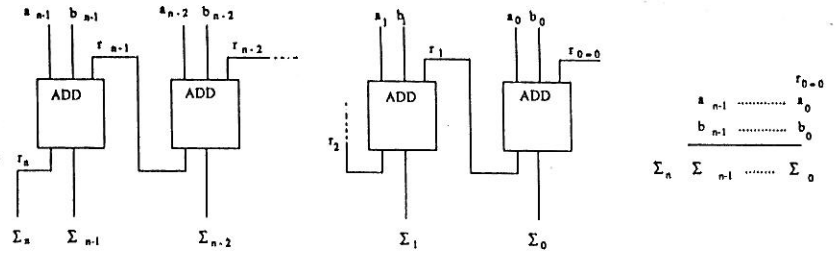
$\Sigma_i = a_i \oplus b_i \oplus r_i$

$r_{i+1} = a_i b_i + r_i (a_i \oplus b_i)$

Ce qui donne le schéma :



L'addition de deux nombres de n éléments binaires nécessite n additionneurs complets, la retenue appliquée sur les plus faibles poids est nulle et chaque retenue calculée est appliquée au chiffre de poids immédiatement supérieur.

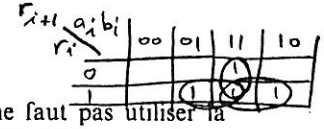


Cette solution est intéressante d'un point de vue du matériel parce que répétitive. Par contre, comme le résultat d'une addition ne peut pas être obtenu instantanément, le temps maximum mis pour obtenir le résultat est directement proportionnel au nombre d'additionneurs. En effet, après le premier temps de calcul la retenue r_1 est appliquée au second additionneur. Ce n'est qu'après le second temps de calcul que la retenue r_2 est délivrée et ainsi de suite, jusqu'au dernier additionneur. Pour cette raison, l'addition ainsi réalisée porte le nom d'« **additionneur à propagation de la retenue** » ou « **additionneur à retenue série** ».

Pour éliminer cet inconvénient, la seconde technique consiste à calculer toutes les retenues en parallèle, directement à partir des données sans même calculer les sommes partielles. Le circuit ainsi réalisé est alors appelé « **additionneur à retenue anticipée** ».

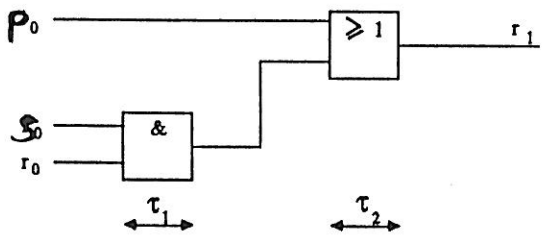
En reprenant le tableau de Karnaugh relatif au calcul de la retenue il vient :

$$r_{i+1} = a_i b_i + r_i(a_i + b_i)$$



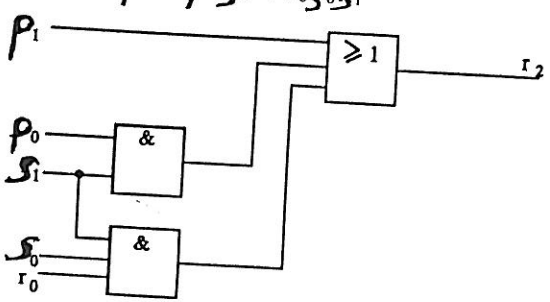
Afin d'éviter des temps de calcul cumulatifs, il ne faut pas utiliser la relation en tant que relation de récurrence, c'est-à-dire qu'il ne faut pas utiliser un résultat de calcul pour le calcul suivant. Il faut systématiquement recalculer chaque terme, ce qui donne en posant $\Sigma_i = a_i + b_i$ et $P_i = a_i b_i$:

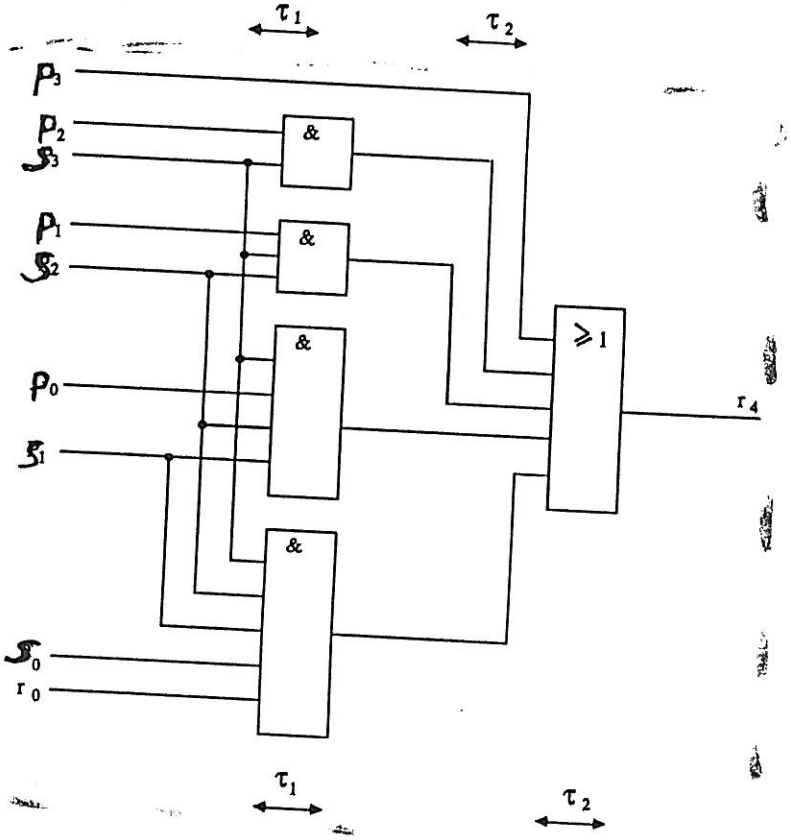
$$r_1 = P_0 + r_0 \Sigma_0$$



De même :

$$r_2 = p_1 + r_1 s_1 = p_1 + (p_0 + r_0 s_0) s_1$$
$$r_2 = p_1 + p_0 s_1 + r_0 s_0 s_1$$





Et ainsi de suite :

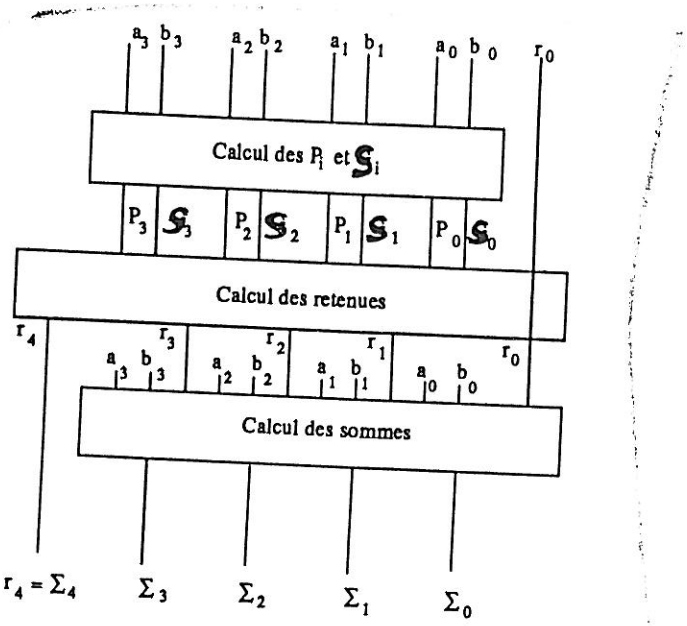
$$r_3 = p_2 + r_2 s_2 = p_2 + (p_1 + p_0 s_1 + r_0 s_0 s_1) s_2$$
$$= p_2 + p_1 s_2 + r_0 s_0 s_1 s_2$$

et :

$$r_4 = p_3 + r_3 s_3$$
$$= p_3 + p_2 s_3 + p_1 s_2 s_3 + p_0 s_1 s_2 s_3 + r_0 s_0 s_1 s_2 s_3$$

On constate que les temps de calcul des retenues sont tous égaux. Ils correspondent au temps de transit de l'information dans une porte ET (τ_1) et une porte OU (τ_2) en cascade (le nombre d'entrée d'une porte n'affectant pas son temps de transit).

La structure d'un additionneur quatre éléments binaires utilisant la technique de calcul anticipé des retenues est la suivante :



Afin d'illustrer le gain apporté par le principe de la retenue anticipée, le tableau ci-après donne les temps de calcul pour une addition de deux nombres de formats différents :

Format de chaque nombre en bits	Temps de calcul en nS (logique TTL série N)		
	Propagation de la retenue	Retenue anticipée	
4	24	24	avec un additionneur 4 bits intégrés
8	36	36	avec utilisation d'un générateur de retenue
12	48	36	
16	60	36	
64	192	60	avec deux générateurs de retenue en cascade

3. LE COMPARATEUR

Un comparateur est un dispositif capable de détecter l'égalité de deux nombres et éventuellement d'indiquer le nombre le plus grand ou le plus petit.

3.1. Principe

Pour effectuer la comparaison de deux nombres, deux techniques sont couramment utilisées :

— La soustraction des deux nombres. Si le résultat de l'opération $A - B$ est positif, cela signifie que A est supérieur à B. Si le résultat est nul, les deux nombres sont égaux.

— Une comparaison bit à bit. C'est cette méthode qui est utilisée dans la plupart des circuits intégrés commercialisés. La comparaison s'effectue poids à poids en commençant par le chiffre le plus significatif.

Les nombres A et B ayant le même format, le nombre A est forcément supérieur à B si son élément binaire le plus significatif (MSB) est supérieur à celui de B. Si ces deux éléments binaires sont égaux, la supériorité (ou l'infériorité) ne peut être déterminée que par l'examen des bits de poids immédiatement inférieur et ainsi de suite.

L'examen des poids successifs s'arrête dès que l'un des éléments binaires est supérieur ou inférieur à l'autre. Les deux nombres A et B sont égaux si, après avoir examiné tous les éléments binaires, il n'a pas été détecté de supériorité ou d'infériorité.

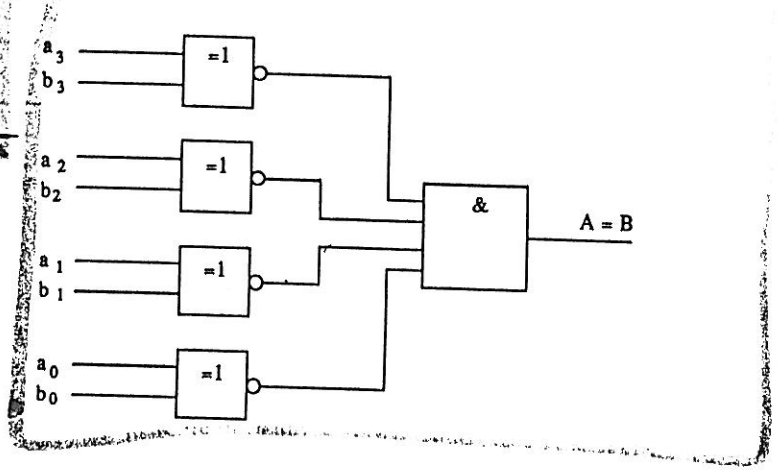
3.1.1. Comparateur donnant l'égalité de deux nombres

C'est le comparateur le plus simple. Deux nombres sont égaux si tous les chiffres sont égaux deux à deux. Pour détecter l'égalité de deux éléments binaires, un opérateur OU exclusif complémenté est indispensable. Un opérateur ET indique la simultanéité de toutes les inégalités partielles.

Soient deux nombres A et B de quatre éléments binaires chacun, $A = a_3a_2a_1a_0$ et $B = b_3b_2b_1b_0$:

$A = B$ si $(A_3 = b_3)$ ET $(a_2 = b_2)$ ET $(a_1 = b_1)$ ET $(a_0 = b_0)$

Ce qui donne le schéma :



3.2. Comparateur complet

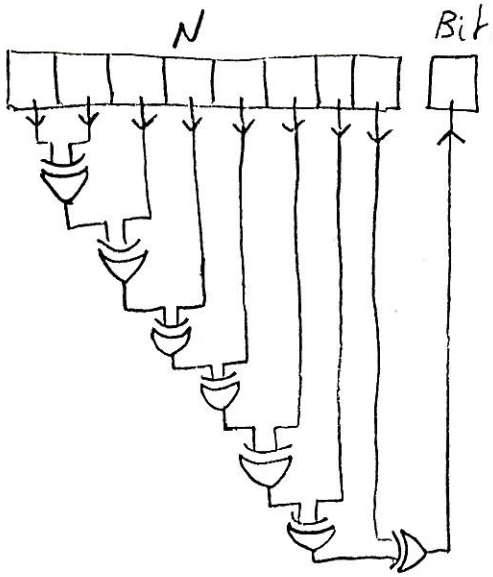
Par analogie avec l'additionneur, la conception d'un comparateur complet pour des nombres de quatre éléments binaires peut se faire de deux façons différentes.

— **Première solution :** En cascade, c'est-à-dire avec propagation des égalités partielles. Les poids de A et de B sont comparés en commençant par le plus élevé. La comparaison sur les poids faibles ne peut être faite que si tous les bits de poids plus élevés sont égaux deux à deux.

12.4 6.4. Générateur de Parité

On appelle parité d'un mot binaire N le nombre de 1 contenus dans le mot le mot a une parité paire si ce nombre de 1 est pair. Afin de rendre les transmissions numériques plus robustes au bruit, on adjoint un bit à tous les mots transmis. Ce bit, dit de parité, est choisi de façon à ce que le mot complet formé du mot et du bit de parité soit pair.

Le principe utilisé pour générer ce bit de parité repose sur la propriété du OU exclusif : $a \oplus b \oplus c \oplus \dots \oplus m$ vaut 1 si un nombre impair de variables est au niveau 1 :



rappel : $\overline{a \oplus b} = a\bar{b} + \bar{a}b$

$\overline{a \oplus b} = \bar{a}\bar{b} + ab = a \odot b$

k_1 : $m_2 m_1$ \ m_0	0	1
0 0	1	0
0 1	0	1
1 1	1	0
1 0	0	1

$k_1 = \bar{m}_2 \bar{m}_1 \bar{m}_0 + \bar{m}_2 m_1 m_0 + m_2 m_1 \bar{m}_0 + m_2 \bar{m}_1 m_0$

$= \bar{m}_2 (\bar{m}_1 \bar{m}_0 + m_1 m_0) + m_2 (m_1 \bar{m}_0 + \bar{m}_1 m_0)$

$= \bar{m}_2 (\overline{m_0 \oplus m_1}) + m_2 (m_0 \oplus m_1)$

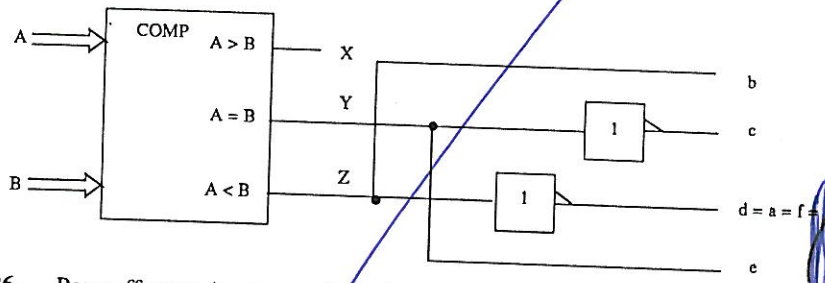
$$\boxed{k_1 = \overline{m_0 \oplus m_1 \oplus m_2}}$$ ($\oplus$ est associatif)

25. De la même façon que pour l'exercice 24, on obtient :

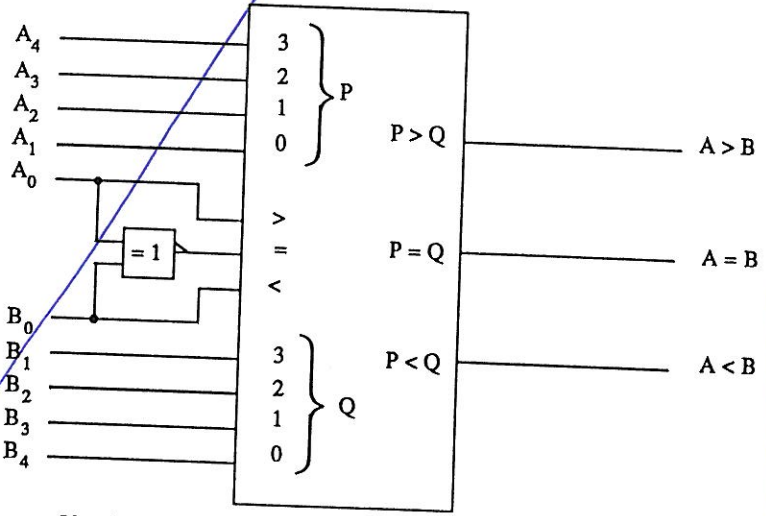
	X $(A > B)$	Y $(A = B)$	Z $(A < B)$	a	b	c	d	e	f	g
S	1	0	0	1	0	1	1	0	1	1
E	0	1	0	1	0	0	1	1	1	1
I	0	0	1	0	1	1	0	0	0	0

d'où les équations logiques :

$$a = X + Y = \overline{Z} = d = f = g$$
$$b = \underline{Z}$$
$$c = \overline{Y}$$
$$e = Y$$



26. Pour effectuer la comparaison de deux nombres de 5 bits à l'aide d'un seul comparateur 4 bits cascadable, il suffit d'utiliser ses entrées de mise en cascade pour l'une des variables. Compte tenu du schéma du 7485, c'est la variable de plus faible poids qui doit y être connecté.



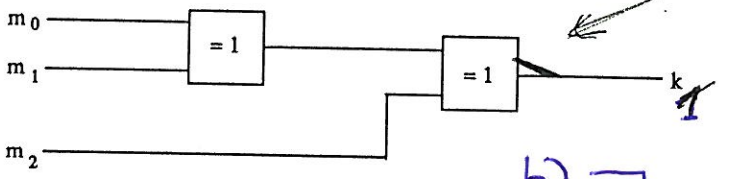
Un circuit de détection d'égalité doit néanmoins être rajouté pour valider l'égalité globale.

Sécuriser une communication se fait en ajoutant de la redondance au message.

14

m_2	m_1	m_0	k_1
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

a) $k_1 = \overline{m_0 \oplus m_1 \oplus m_2}$



b)

15

1°) k_1 est le seul élément binaire de contrôle entrant dans le test de parité T_1, donc k_1 doit être calculé à l'émission de tel façon que le nombre de 1 dans les éléments binaires 1, 3, 5, 7 soit pair.

m_1	m_2	m_4	k_1
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	∅
1	1	1	∅

m_1 \ $m_2 m_4$	00	01	11	10
0		1	∅	1
1	1		∅=1	

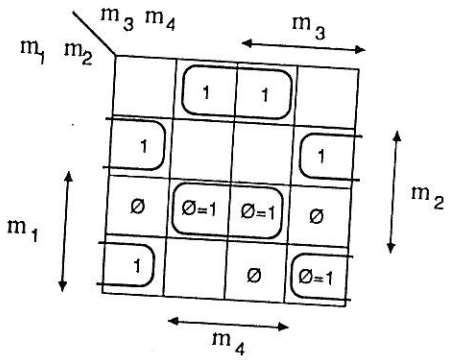
$$k_1 = m_1\overline{m}_2\overline{m}_4 + m_1m_2m_4 + \overline{m}_1\overline{m}_2m_4 + \overline{m}_1m_2\overline{m}_4$$
$$= m_1(\overline{m}_2\overline{m}_4 + m_2m_4) + \overline{m}_1(\overline{m}_2m_4 + m_2\overline{m}_4)$$
$$= m_1(\overline{m_2 \oplus m_4}) + \overline{m}_1(m_2 \oplus m_4)$$
$$= m_1 \oplus m_2 \oplus m_4$$

b) Convention :
T = 1 si erreur
T = 0 sinon

T \ $m'_0 k_1$ / $m'_2 m'_1$	00	01	11	10
00	1	0	1	0
01	0	1	0	1
11	1	0	1	0
10	0	1	0	1

$$T = \overline{m'_2 \oplus m'_1 \oplus m'_0 \oplus k_1}$$

Autre solution k_1 dépend des quatre éléments binaires du message. Le tableau de Karnaugh correspondant est rempli à partir du code de Hamming.



$$k_1 = m_1\overline{m}_2\overline{m}_4 + m_1m_2m_4 + \overline{m}_1m_2\overline{m}_4 + \overline{m}_1\overline{m}_2m_4$$
$$k_1 = m_1 \oplus m_2 \oplus m_4$$

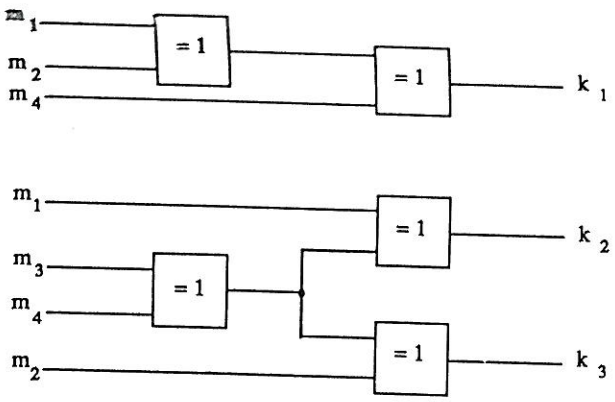
k_2 résulte du test de parité sur les éléments binaires 3. 6. 7. d'où :

$$k_2 = m_1 \oplus m_3 \oplus m_4$$

De même

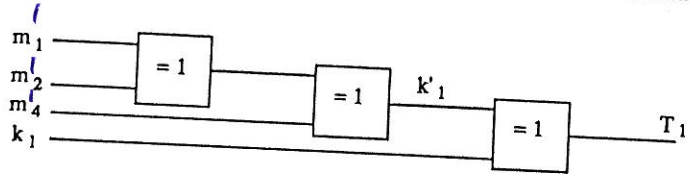
$$k_3 = m_2 \oplus m_3 \oplus m_4$$

Le schéma



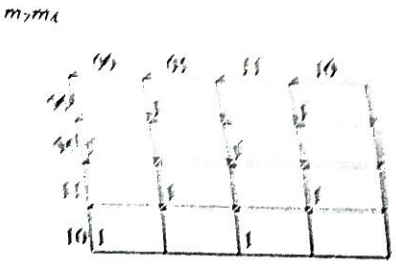
2°) Il y a deux solutions :

a) Fabriquer k'_1 à partir de m_1, m_2 et m_4 et comparer avec k_1 reçu de façon que $T_1 = 0$ si $k_1 = k'_1$ (et $T_1 = 1$ si $k_1 = k'_1$); d'où le schéma :



b) Calculer T_1 à partir des différentes possibilités de $m_1m_2m_4k_1$ (*eb* 1, 3, 5 et 7).

K_1	m_1	m_2	m_4	T_1	
0	0	0	0	0	nombre de 1 pair
0	0	0	1	1	nombre de 1 impair
0	0	1	0	1	
0	0	1	1	0	
0	1	0	0	1	
0	1	0	1	0	
0	1	1	0	0	
1	1	1	1	0	



Ici toutes les combinaisons sont prévues. Certaines sont impossibles s'il n'y a qu'une seule erreur.

Les 1 placés en diagonale indiquent une solution comportant des OU exclusifs.

$$T_1 = k_1 \oplus m_1 \oplus m_2 \oplus m_4$$

De même

$$T_2 = m_1 \oplus m_3 \oplus m_4 \oplus k_2$$

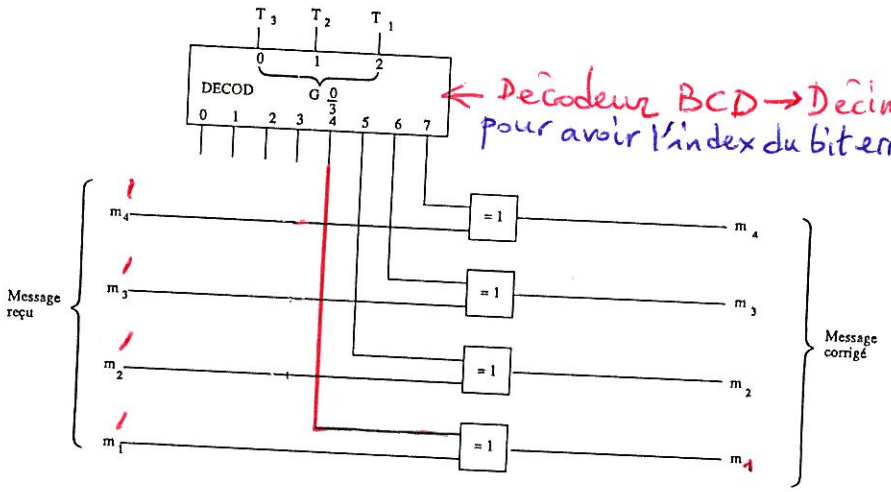
et

$$T_3 = m_2 \oplus m_3 \oplus m_4 \oplus k_3$$

Le schéma est identique à celui de l'émetteur auquel on ajoute trois OU exclusifs.

3°) $(T_3T_2T_1)_2$ indique le numéro de l'élément binaire erroné, donc celui qu'il faut corriger. La correction consiste simplement à changer l'élément binaire faux à l'aide d'un circuit complément.

Le OU exclusif permet d'être soit un circuit transparent soit un circuit inverseur suivant sa commande.



← Décodeur BCD → Décimal pour avoir l'index du bit erroné

Il n'est pas utile de corriger les éléments de contrôle.

Rappel : Pour inverser un bit b selon une commande c (c = 0 = pas d'inversion de b ; c = 1 = inversion du bit b), on utilise un circuit OU exclusif :

$s = b \oplus c = b\bar{c} + \bar{b}c$

| $c = 0 : s = b$
| $c = 1 : s = \bar{b}$

15

Bibliographie

Boole G., *The Mathematical Analysis of Logic*, Cambridge (réédité par Blackwell, 1948).

Boole G., *An Investigation of the Laws of Thought*, Londres (réédité par Dover Publications, 1954).

Karnaugh M., « The Map Method of Synthesis of Combinational Logic Circuits », *Communications and Electronics*, n° 9, November 1953, p. 593-599.

McCluskey E.J. Jr., « Minimization of Boolean Functions », *Bell System Technical Journal*, vol. 35, November 1956, p. 1417-1444.

McCluskey E.J. Jr., « Transients in Combinational Logic Circuits », *Redundancy Techniques for Computing Systems*, Spartan Books Co, 1962, p. 9-46.

McCluskey E.J. Jr., *Introduction to the Theory of Switching*, McGraw-Hill Book Co, New York, 1965.

McCluskey E.J. Jr. and Bartee T.C., *A Survey of Switching Circuits Theory*, McGraw-Hill Book Co, New York, 1962.

McCluskey E.J. Jr. and Schorr H., « Essential Multiple Output Prime Implicants », *Proceeding of the Symposium on the Mathematical Theory of Automata*, 1962, Polytechnic Press of the Polytechnic Institute of Brooklyn, New York, p. 437-457.

Quine W.V., « The Problem of Simplifying Truth Functions », *Am. Math. Monthly*, vol. 59, October 1952, p. 521-531.

Quine W.V., « A Way to Simplify Truth Functions », *Am. Math. Monthly*, vol. 62, November 1955, p. 627-631.

Quine W.V., « On Cores and Prime Implicants of Truth Functions », *Am. Math. Monthly*, vol. 66, November 1959, p. 755-760.

Shannon C.E., « A Symbolic Analysis of Relay and Switching Circuits », *Transactions of the AIEE*, vol. 57, 1938, p. 713-723.

Shannon C.E., « The Synthesis of Two-Terminal Switching Circuits », *Bell System Technical Journal*, vol. 28, 1949, p. 59-98.

Tison P., « Recherche de termes premiers d'une fonction booléenne », *Automatisme*, tome IX, n° 1, janvier 1964.

Tison P., *Théorie des consensus et algorithmes de recherche des bases premières*, Colloque d'algèbre de Boole, Grenoble, janvier 1965.

Tison P., « Algèbre booléenne : théorie des consensus. Recherche des bases premières d'une fonction booléenne », *Automatisme*, tome X, n° 6, juin 1965.

Veith E.W., « A Chart Method for Simplifying Truth Functions », *Proc. ACM*, May 2-3, 1952, p. 127-133.

Documents construits

TP2. LOGIQUE COMBINATOIRE 2

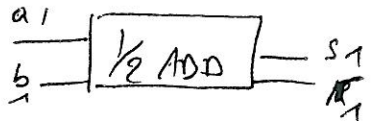
Logique Combin. 2

Corrigé Log 2. [redacted]

[redacted]

3.1. Les additionneurs

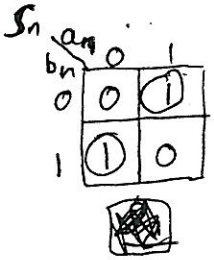
a) additionneurs 2 bits

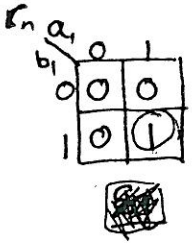


a_n	b_n	S_n	R_n
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

r_n [redacted] $= a_n \cdot b_n$

$S_n = a_n \oplus b_n$ car $= a_n \bar{b}_n + \bar{a}_n b_n$

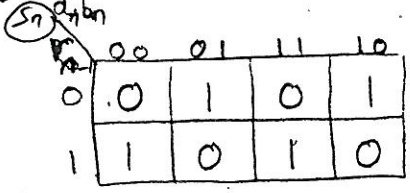




[redacted]

b) additionneur 3 bits

a_n	b_n	[redacted] r_0	S_n	[redacted] r_n
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

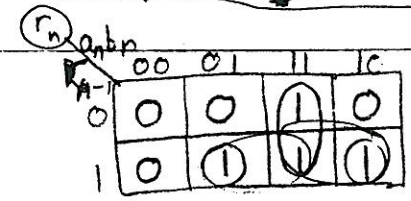


[redacted]

$S_n = \bar{a}_n \bar{b}_n r_{n-1} + \bar{a}_n b_n \bar{r}_{n-1} + a_n \bar{b}_n \bar{r}_{n-1} + a_n b_n r_{n-1}$

$= \bar{r}_{n-1}(a_n \bar{b}_n + \bar{a}_n b_n) + r_{n-1}(a_n b_n + \bar{a}_n \bar{b}_n)$

$S_n = a_n \oplus b_n \oplus r_{n-1}$



[redacted]

r_n [redacted] $= r_{n-1} a_n \bar{b}_n + r_{n-1} b_n \bar{a}_n + a_n b_n$

r_n [redacted] $= a_n b_n + r_{n-1}(a_n \oplus b_n)$

(erreur sur le schéma : opérateur de sortie de R_1 est un "ou" et non pas un "et")

en regroupant au maxi, on n'a pas le ou exclusif :

$r_n = a_n b_n + b_n r_{n-1} + a_n r_{n-1}$

$\rightarrow r_n = a_n b_n + r_{n-1}(a_n + b_n)$

transcodeur BCD. Affichage 7segments

un afficheur peut afficher des nombres allant de 0 à 9.

Pour coder 10 chiffres, il faut au moins 4 bits : $2^3 = 8 < 10$

$2^4 = 16 > 10$.

A	B	C	D	N	a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	1	0	1	1	0	0	0	0
0	0	1	0	2	1	1	0	1	1	0	1
0	0	1	1	3	1	1	1	1	0	0	1
0	1	0	0	4	0	1	1	0	0	1	1
0	1	0	1	5	1	0	1	1	0	1	1
0	1	1	0	6	1	0	1	1	1	1	1
0	1	1	1	7	1	1	1	0	0	0	0
1	0	0	0	8	1	1	1	1	1	1	1
1	0	0	1	9	1	1	1	1	0	1	1
1	0	1	0	A							
1	0	1	1	B							
1	1	0	0	C							
1	1	0	1	D							
1	1	1	0	E							
1	1	1	1	F							

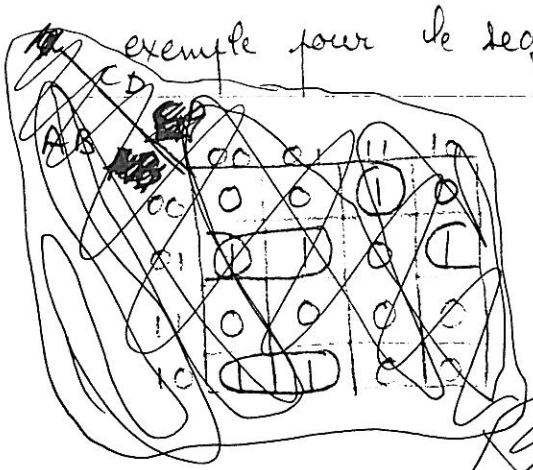
X

(sans importance)

(a)

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

exemple pour le segment a :



$g = \bar{D}C\bar{B}\bar{A} \vee \bar{D}B\bar{A} \vee B\bar{C}D \vee D\bar{C}\bar{B}$

$= A\bar{B}\bar{C}\bar{D} \vee (\bar{A} \vee \bar{B})C\bar{D} \vee D\bar{C}\bar{B}$

$a = A + C + BD + \bar{B}\bar{D} = A + C + \overline{B \oplus D}$

$g = \bar{D}C\bar{B}\bar{A} + \bar{D}B\bar{A} + C\bar{B}\bar{A} + AB\bar{C}$

$\rightarrow g = \bar{D}C\bar{B}\bar{A} + (\bar{D} + \bar{C})B\bar{A} + AB\bar{C}$

3.5- Les Comparateurs (logigramme : cf énoncé)

A		B		A>B	A=B	A<B
a_1	a_0	b_1	b_0			
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

A>B

b_1b_0 \ a_1a_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$(A>B) = \bar{b}_1 a_1 + \bar{b}_1 \bar{b}_0 a_0 + \bar{b}_0 a_1 a_0$

EWB: $\left(\begin{smallmatrix} A \\ ab \end{smallmatrix} > \begin{smallmatrix} B \\ cd \end{smallmatrix}\right) = \bar{c}\, a + \bar{c}\, \bar{d}\, b + \bar{d}\, a\, b$

A=B

b_1b_0 \ a_1a_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$(A=B) = \bar{a}_0 \bar{a}_1 \bar{b}_0 \bar{b}_1 + a_0 \bar{a}_1 b_0 \bar{b}_1 + \bar{a}_0 a_1 \bar{b}_0 b_1 + a_0 a_1 b_0 b_1$

EWB: $\left(\begin{smallmatrix} A \\ ab \end{smallmatrix} = \begin{smallmatrix} B \\ cd \end{smallmatrix}\right) = \bar{b}\,\bar{a}\,d\,\bar{c} + b\,\bar{a}\,d\,\bar{c} + \bar{b}\,a\,d\,\bar{c} + b\,a\,d\,\bar{c}$

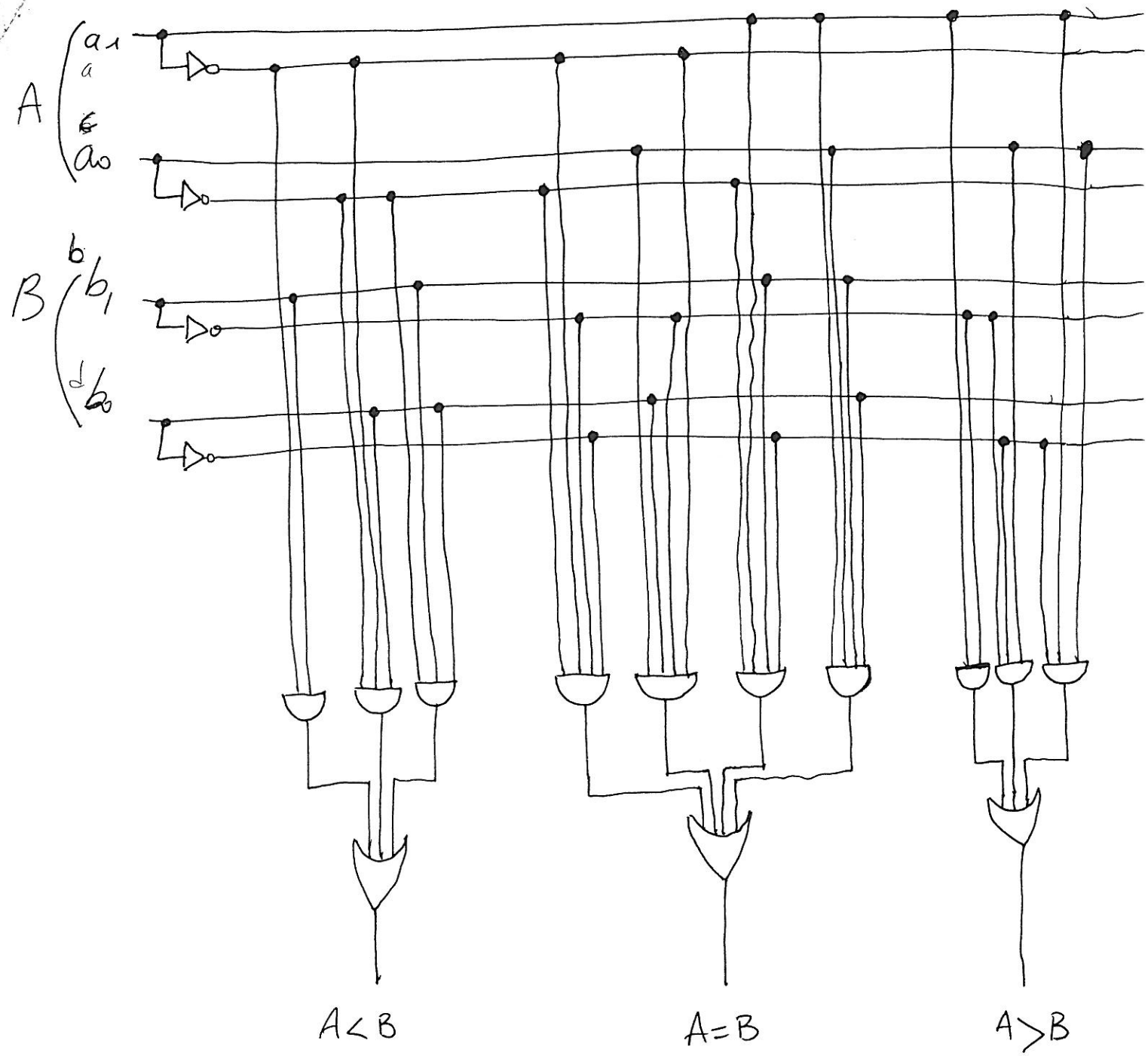
A<B

b_1b_0 \ a_1a_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$(A<B) = \bar{a}_1 b_1 + \bar{a}_0 \bar{a}_1 b_0 + \bar{a}_0 b_1 b_0$

EWB: $\left(\begin{smallmatrix} A \\ ab \end{smallmatrix} < \begin{smallmatrix} B \\ cd \end{smallmatrix}\right) = \bar{a}\,\bar{c} + \bar{b}\,\bar{a}\,d + \bar{b}\,\bar{c}\,d$

Représentation symbolique :



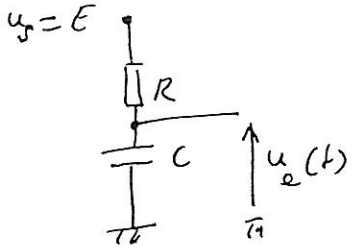
TP Electronique n°2 (suite) CORRIGÉ

Logique Combinatoire (2ème). Multiplexeur-Démultiplexeur

3. Etude théorique

3.0. Horloge :

* origine des temps : instant où u_s passe de 0 à E. $t=0$: on a alors : $u_e = U_1$



C se charge à E à travers R avec comme C.i. : $u_e(0) = U_1$

$$u_e(t) = E(1 - e^{-t/RC}) + U_1 \cdot e^{-t/RC} \quad : \quad 0 < t < t_1$$

$$\left(\begin{array}{l} u_e(0) = U_1 \\ "u_e(\infty)" = E \end{array} \right)$$

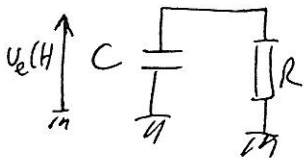
$$\rightarrow \boxed{u_e(t) = E(1 - e^{-t/\tau}) + U_1 e^{-t/\tau}} \quad 0 < t < t_1$$

* $t_1 = ?$ à $t = t_1$: $u_e(t_1) = U_2$

$$\rightarrow U_2 = E(1 - e^{-t_1/\tau}) + U_1 e^{-t_1/\tau} = e^{-t_1/\tau}[U_1 - E] + E$$

$$\rightarrow \boxed{t_1 = \tau \ln \frac{U_1 - E}{U_2 - E}}$$

* origine des temps : instant où u_s passe de E à 0 : $t=0$: on a alors : $u_e = U_2$



C se décharge dans R avec comme C.i. : $u_e(0) = U_2$:

$$u_e(t) = U_2 \cdot e^{-t/RC} \quad : \quad 0 < t < t_2$$

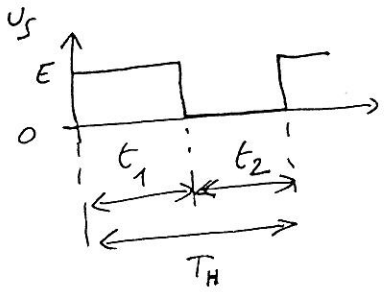
$$\left(\begin{array}{l} u_e(0) = U_2 \\ "u_e(\infty)" = 0 \end{array} \right)$$

$$\rightarrow \boxed{u_e(t) = U_2 \cdot e^{-t/\tau}} \quad 0 < t < t_2$$

* $t_2 = ?$ à $t = t_2$: $u_e(t_2) = U_1$

$$\rightarrow U_1 = U_2 \cdot e^{-t_2/\tau} \rightarrow \boxed{t_2 = \tau \ln \frac{U_2}{U_1}}$$

* Période de l'oscillation : $T_H = t_1 + t_2 = \tau \ln \frac{U_1 - E}{U_2 - E} + \tau \ln \frac{U_2}{U_1}$



$$\boxed{T_H = \tau \ln \frac{U_2 (U_1 - E)}{U_1 (U_2 - E)}}$$

* Rapport cyclique : $K_0 = \frac{t_1}{T_H} = \frac{1}{2} \rightarrow \frac{\tau . \ln\left(\frac{U_1 - E}{U_2 - E}\right)}{\tau . \ln\left(\frac{U_2 (U_1 - E)}{U_1 (U_2 - E)}\right)} = \frac{1}{2}$

$$\rightarrow 2 \ln\left(\frac{U_1 - E}{U_2 - E}\right) = \ln\left(\frac{U_2 (U_1 - E)}{U_1 (U_2 - E)}\right)$$

$$\rightarrow \ln\left[\left(\frac{U_1 - E}{U_2 - E}\right)^2\right] = \ln\left(\frac{U_2 (U_1 - E)}{U_1 (U_2 - E)}\right)$$

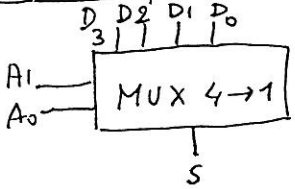
$$\rightarrow \left(\frac{U_1 - E}{U_2 - E}\right)^2 = \frac{U_2}{U_1}\left(\frac{U_1 - E}{U_2 - E}\right) \rightarrow \frac{U_1 - E}{U_2 - E} = \frac{U_2}{U_1}$$

$$\rightarrow \boxed{U_1 (U_1 - E) = U_2 (U_2 - E)}$$

* A.N : ...

3.4. Multiplexeur

a)

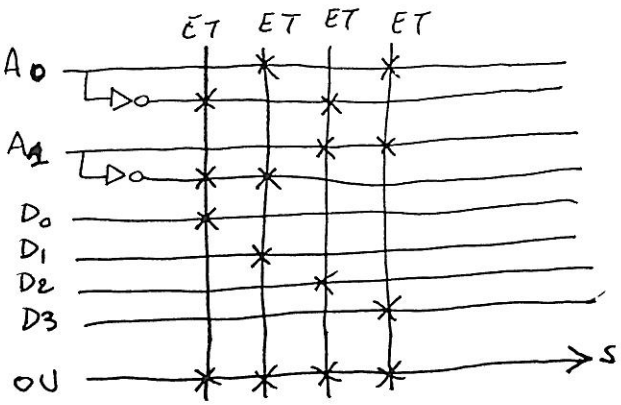


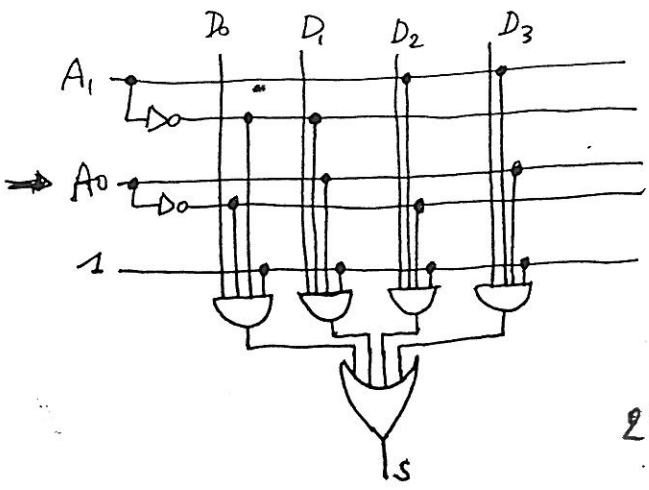
b) Table de vérité :

A_1	A_0	S
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

c) $S = \bar{A}_1 \bar{A}_0 D_0 + \bar{A}_1 A_0 D_1 + A_1 \bar{A}_0 D_2 + A_1 A_0 D_3$

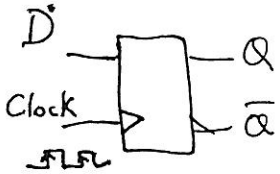
d)



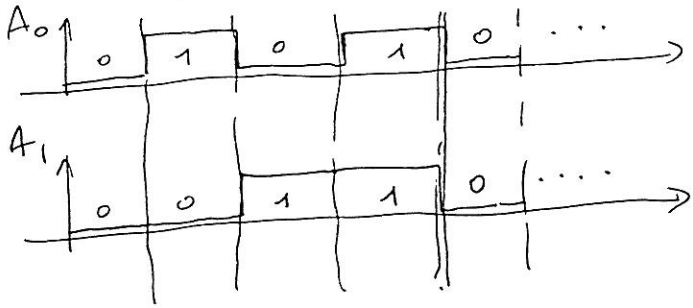


c) Rappel : bascule D ↓ :

Clock	D	Q_n
X	X	Q_{n-1}
↓	0	0
↓	1	1

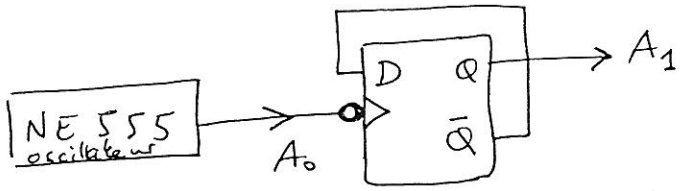


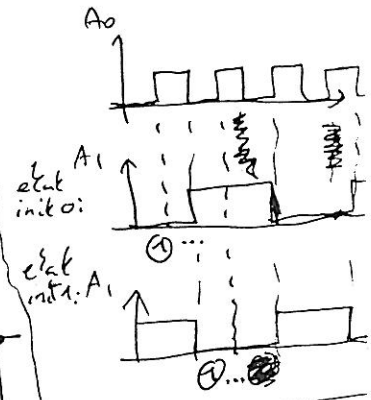
On désire la séquence suivante pour les adresses A_1, A_0 :



A_1 : division par 2 de A_0.

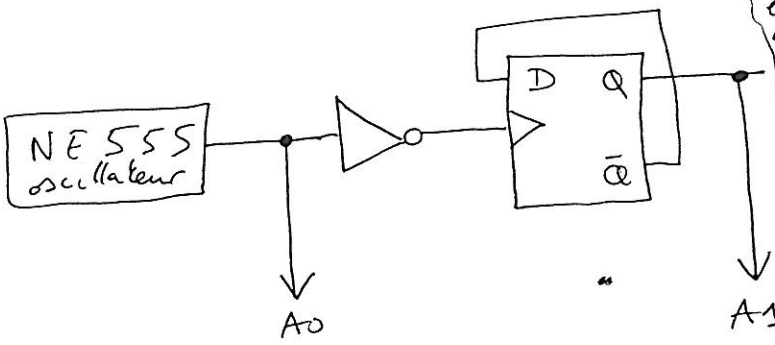
Si A_0 représente l'horloge de la bascule D ↓ générant A_1, il suffit d'une part de complémenter l'horloge car on voit que le changement d'état de A_1 a lieu pour un front descendant de A_0, et d'autre part d'envoyer la sortie $\bar{Q}$ de la bascule D en entrée de D, car à chaque front descendant de l'horloge A_0, A_1 change d'état. On a donc le schéma suivant : (l'état initial de la bascule D peut être pris quelconque)





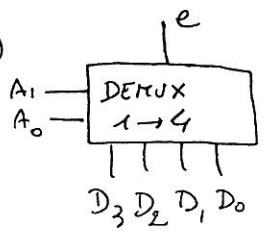
① : 1ère phase de la séquence

ou encore :



3.5. Démultiplexeur

a)



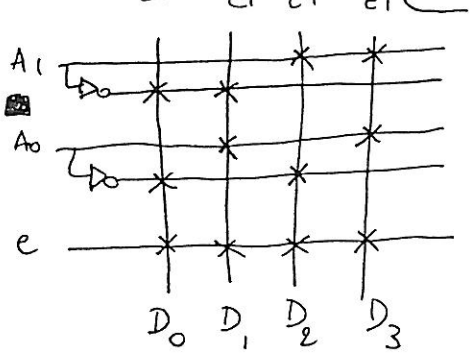
b) Table de vérité :

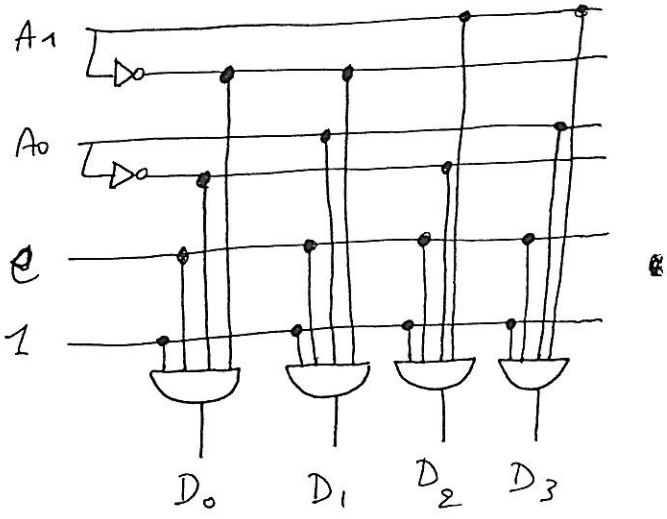
A_1 A_0	D_i
0 0	$D_0 = e$
0 1	$D_1 = e$
1 0	$D_2 = e$
1 1	$D_3 = e$

c) Equation des sorties :

$$D_0 = e\,\bar{A}_1\,\bar{A}_0$$
$$D_1 = e\,\bar{A}_1\,A_0$$
$$D_2 = e\,A_1\,\bar{A}_0$$
$$D_3 = e\,A_1\,A_0$$

d)





4. Etude expérimentale

4.4 Multiplexeur

a) On vérifie que si $A_1 A_0 = 00$, les LEDs L et L_0 clignotent à la même fréquence.

_____ = 01, _____ L_1 _____.

_____ = 10, _____ L_2 _____.

_____ = 11, _____ L_3 _____.

4.5 Démultiplexeur

a) On vérifie que si $A_1 A_0 = 00$, les LEDs L et L_0 clignotent ensemble.

_____ = 01, _____ L_1 _____.

_____ = 10, _____ L_2 _____.

_____ = 11, _____ L_3 _____.

4.6 Multiplexeur d'affichage

4.6.1. Affichage d'un digit :

Avec le Transcodeur TTL 7447, on a : (← idem →)

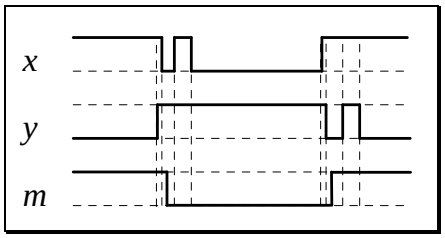
- pin 4 : $\overline{BI}/\overline{RBo} = 1$
- pin 5 : $\overline{RBi} = 1$ (affichage de ∅)
- pin 3 : $\overline{LT} = 1$ (test des sorties)

sorties complémentées

TD 3 CORRIGE. LOGIQUE SEQUENTIELLE 1

3. Bascule RS à entrées complémentées

COMMENTAIRES

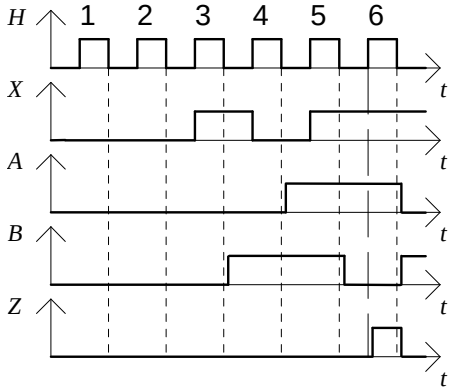


$x = \overline{R}$	$y = \overline{S}$	$m = Q_n$	***Fonction***
1	1	Q_{n-1}	*Mémorisation*
0	1	0	*RESET (Mise à 0 de Q)*
1	0	1	*SET (Mise à 1 de Q)*
0	0		*Combinaison interdite*

Application anti-rebond de la bascule RS

4. Système séquentiel

L'état initial des bascules *JK* est l'état 0 :



TD 3 ANNEXE CORRIGE. LOGIQUE SEQUENTIELLE 1

1. Détection synchrone d'une séquence (serrure électronique)

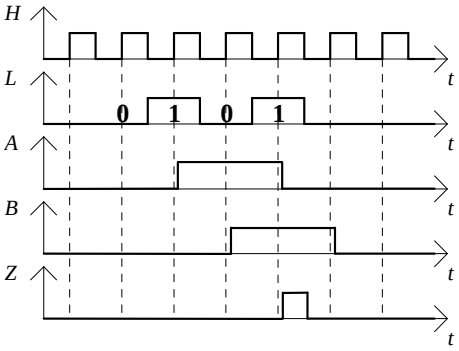
1. *Solution à logique câblée (1)*

1.1. Les bascules ont un état initial bas : $A = 0$, $B = 0$

1.2. Séquence à détecter :

Séquence à détecter *a b c d*

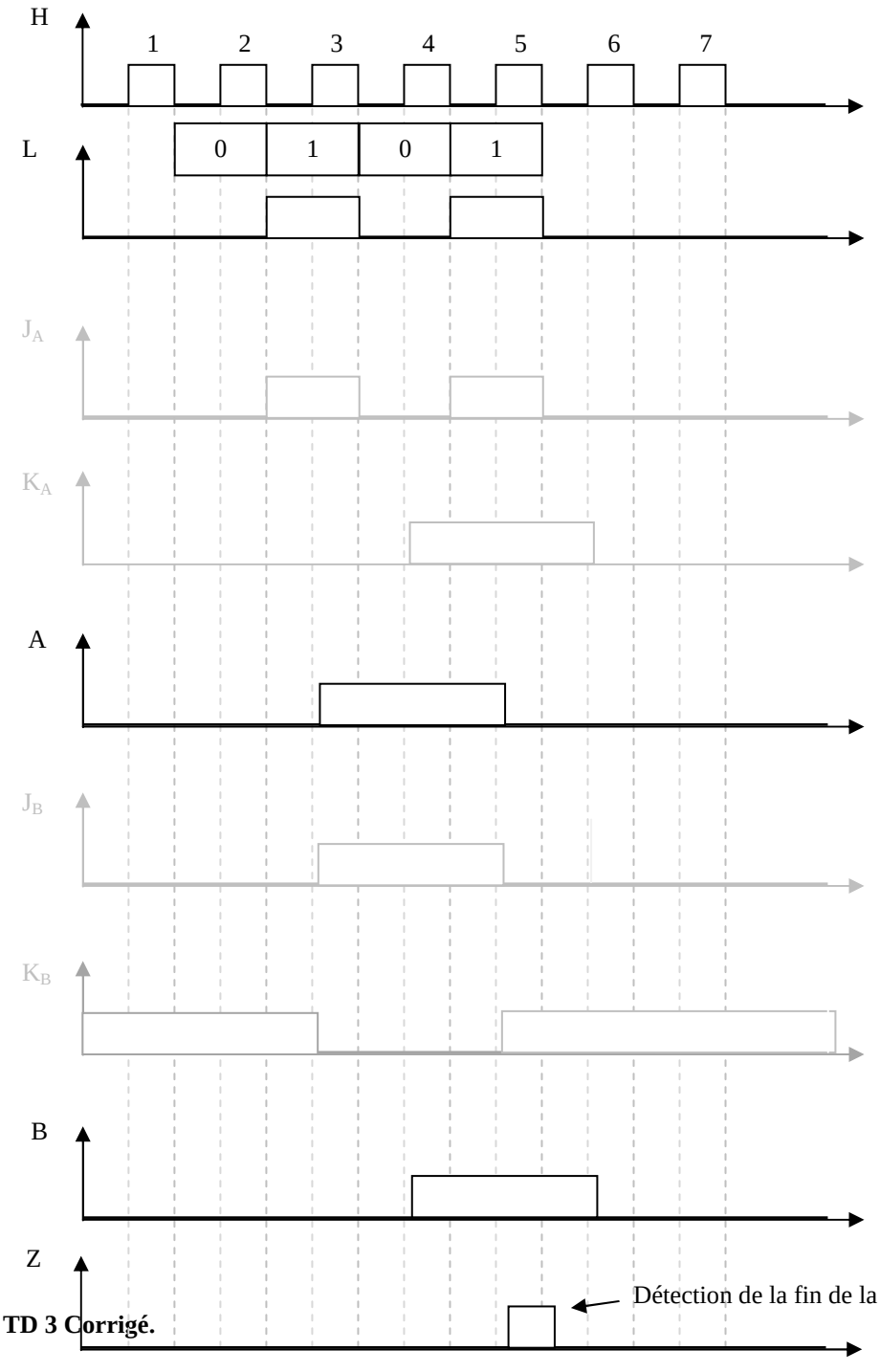
0	1	0	1



$J_A = L$
$K_A = B$

$J_B = A$
$K_B = \overline{A}$

$Z = B L \overline{A} H$

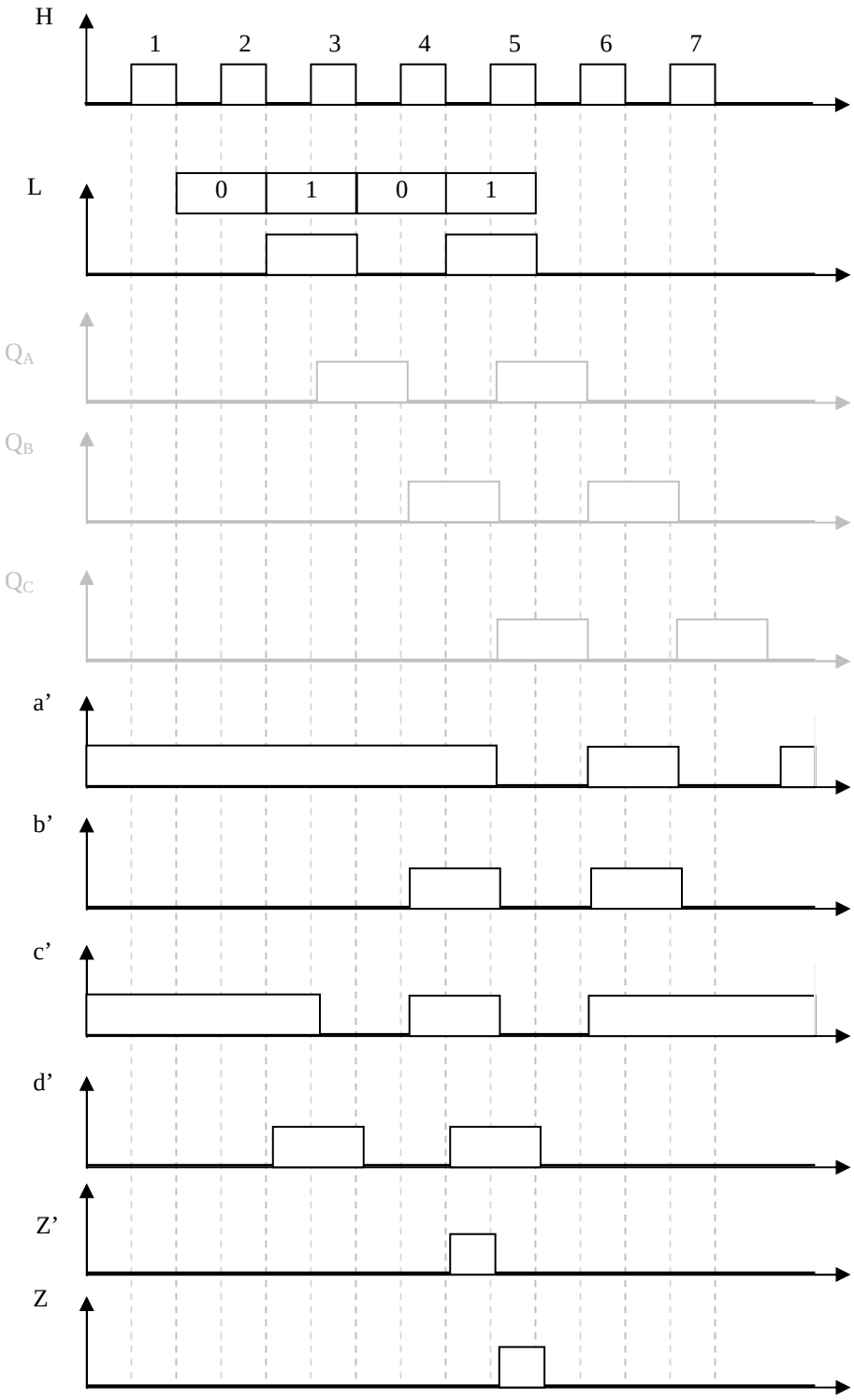


$J_A = L$
$K_A = B$

$J_B = A$
$K_B = A'$

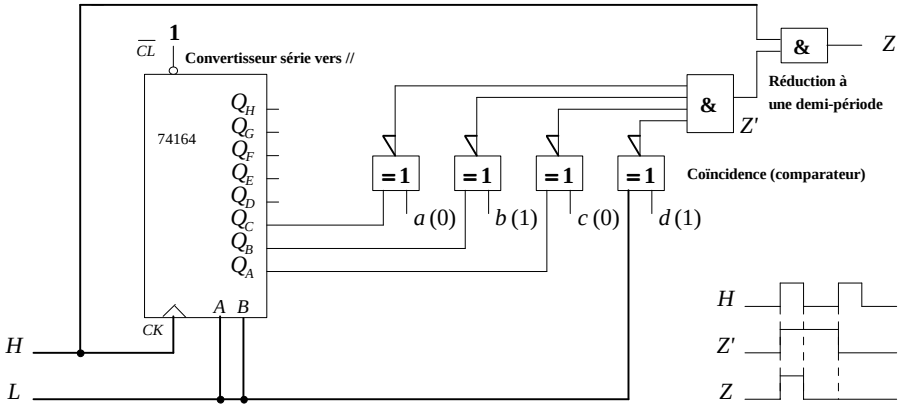
$Z = B \cdot A' \cdot H \cdot L$

abcd = 0101


H
1
2
3
4
5
6
7
L
0
1
0
1
Q_A
Q_B
Q_C
a’
b’
c’
d’
Z’
Z

2. *Solution à logique câblée (2)*

2.1.

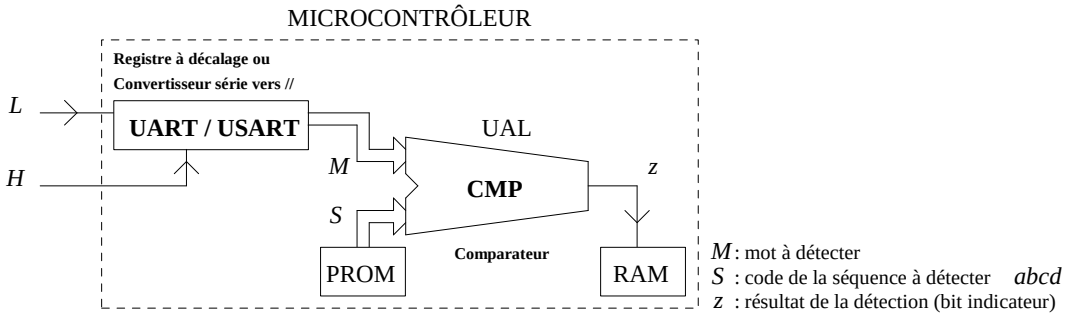


2.2.

Avantage 1	*Avantage 2*
• La séquence à détecter est programmable, par exemple à l'aide de commutateurs (switches) mécaniques, et peut être modifiée à volonté.	• Le circuit peut être étendu à un nombre quelconque de bits de la séquence à décoder, en allongeant le registre à décalage et le circuit de décodage.

3. *Solution à logique programmée*

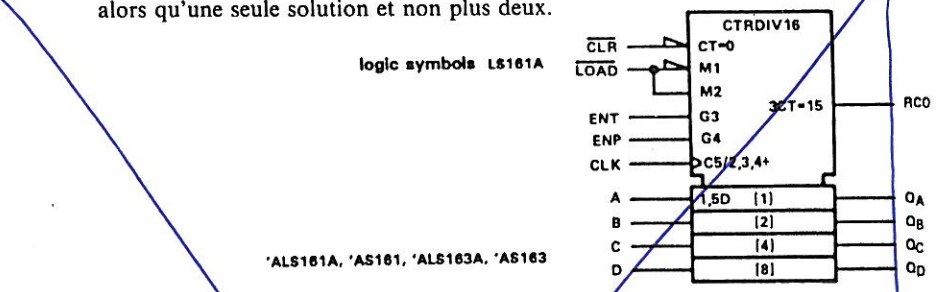
3.1.



3.2.

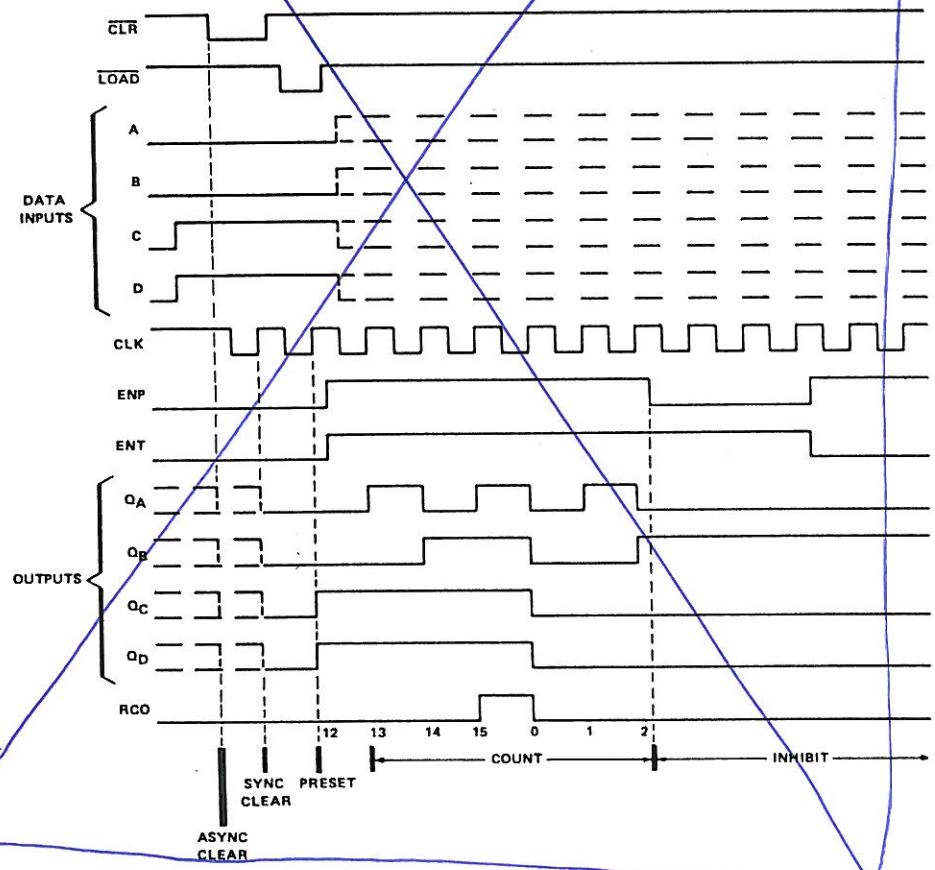
Avantage
• Le programme logé dans le microcontrôleur peut être modifié pour réaliser d'autres opérations que la simple détection d'une séquence.

35. Mêmes exercies avec le compteur 74161 (clear asynchrone). On ne recherche alors qu'une seule solution et non plus deux.



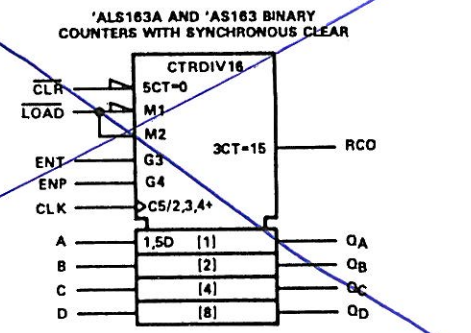
'ALS161A, 'AS161, 'ALS163A, 'AS163

Illustrated below is the following sequence:
1. Clear outputs to zero ('ALS161A and 'AS161 are asynchronous; 'ALS163A and 'AS163 are synchronous)
2. Preset to binary twelve
3. Count to thirteen, fourteen, fifteen, zero, one, and two
4. Inhibit



Document Texas Instruments.

36. Mêmes exercices avec le compteur 74163 (clear synchrone).



- Système combin. : $S = f(\text{entrées})$ (sortie)
- Système séquentiel : $S = f(\text{entrées, état, précédent de S})$ ≡ à une mém. combinatoire, la sortie est différente à 2 instants différents pour les mêmes entrées

Solutions des exercices

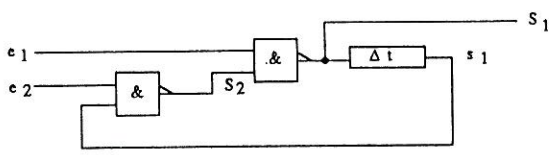
1.

1a. Séquentiel. En effet, à un même état des entrées (H = 1 E = 1) correspond deux états possibles pour S. (voir 2ème et 3ème niveau haut de H)

b. Combinatoire : H . (A ⊕ B). (déf. combin., non mise en défaut)

c. Séquentiel. Même remarque qu'en 1a. (voir 1er et dernier niveau haut de

2a. Il n'y a qu'une seule boucle :



$$S_1 = \overline{(s_1 \cdot e_2) \cdot e_1} = s_1 \cdot e_2 + \overline{e_1}$$

$$S_2 = \overline{e_2 s_1} = \overline{e_2} + \overline{s_1}$$

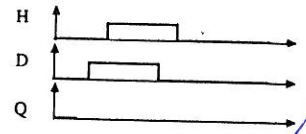
d'où la matrice d'excitation :

S_1 ; s_1 \ e_1e_2	00	01	11	10
0	1	1	⓪	⓪
1	①	①	①	0

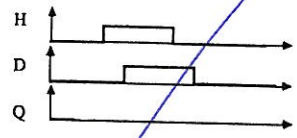
TD ELN n° 3
Logique séquentielle 1
Corrigé

9. Mêmes exercices avec une bascule D latch.

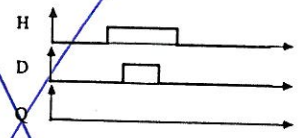
a.



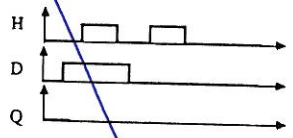
b.



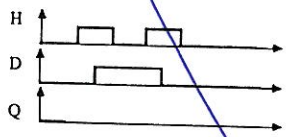
c.



d.

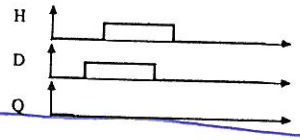


e.

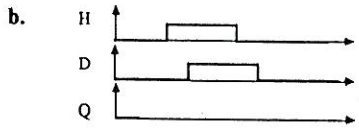


10. Mêmes exercices pour une bascule D synchronisée par un front positif (*positive edge triggered*).

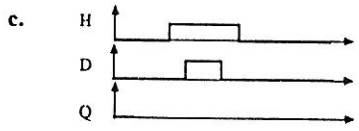
a.



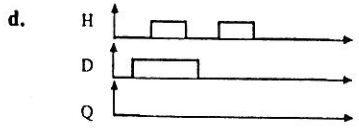
b.



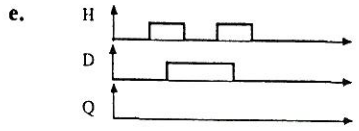
c.



d.



e.



Rappels :

D

C	D	Q_n
0	X	Q_{n-1}
1	0	0
1	1	1

JK

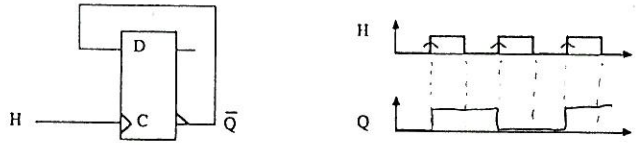
J	K	Q_n
0	0	Q_{n-1}
0	1	0
1	0	1
1	1	$\overline{Q}_{n-1}$

ou encore :

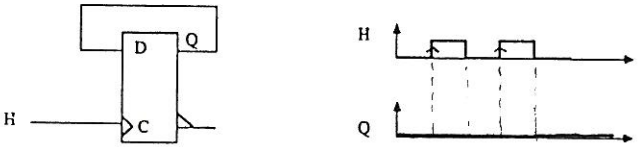
transition $Q_{n-1} \to Q_n$	J K
$0 \to 0$	0 X
$0 \to 1$	1 X
$1 \to 1$	X 0
$1 \to 0$	X 1

2. Compléter les chronogrammes pour chacun des schémas suivants :

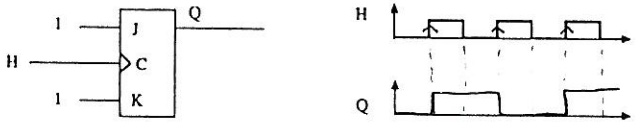
a.



b.



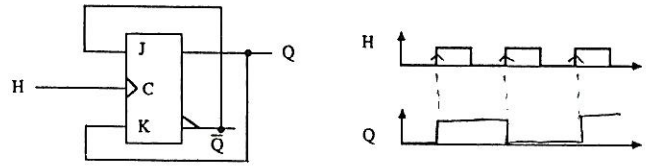
c.



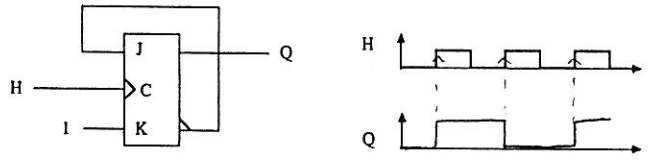
Diviseur par 2 de la fréq. de H.

2 suite

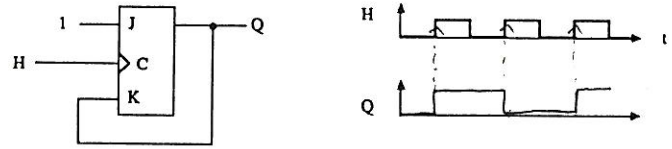
d.



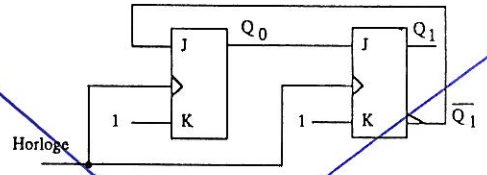
e.



f.

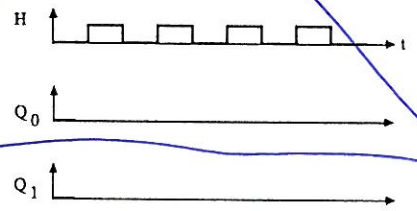


12. On donne le schéma suivant :

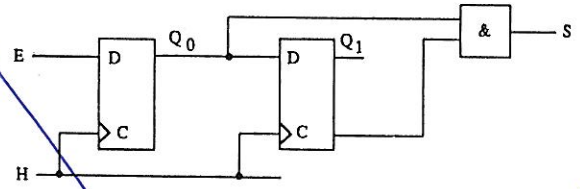


Initialement $Q_0 = Q_1 = 0$. Quelles seront les valeurs des sorties Q_0 et Q_1 ?

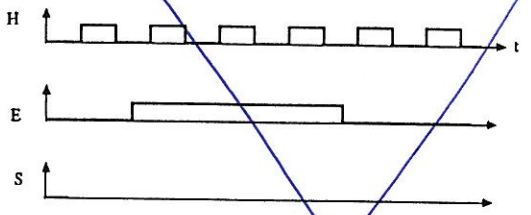
a. Après une impulsion d'horloge.
b. Après deux impulsions d'horloge.
c. Après trois impulsions d'horloge.
d. Compléter le chronogramme suivant.



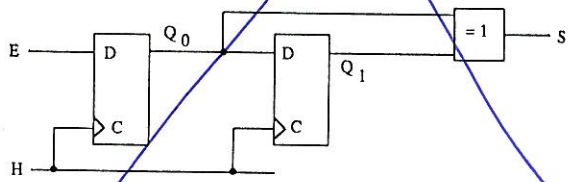
13. Analyser le fonctionnement du montage suivant :



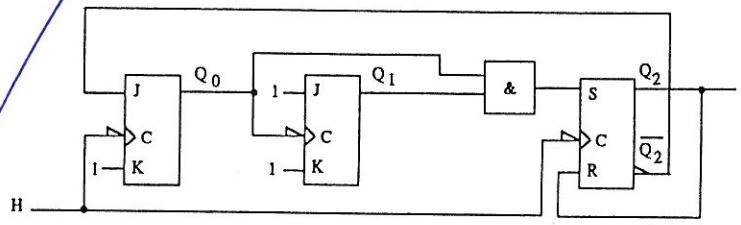
et compléter le chronogramme.



14. Même exercice avec le montage.



15. Donner la succession des états du compteur suivant, celui-ci étant supposé à $(000)_2$ au départ.

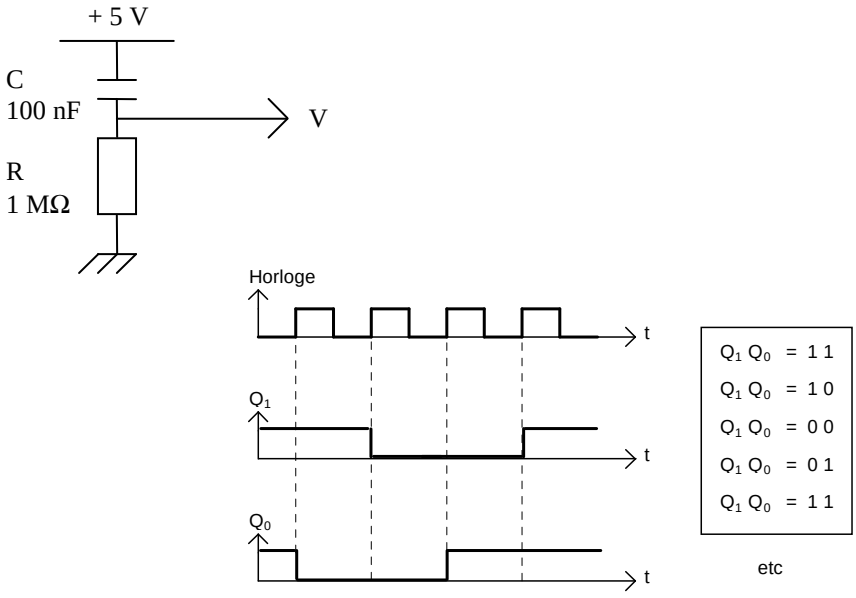


30

TP 3 CORRIGE. LOGIQUE SEQUENTIELLE 1

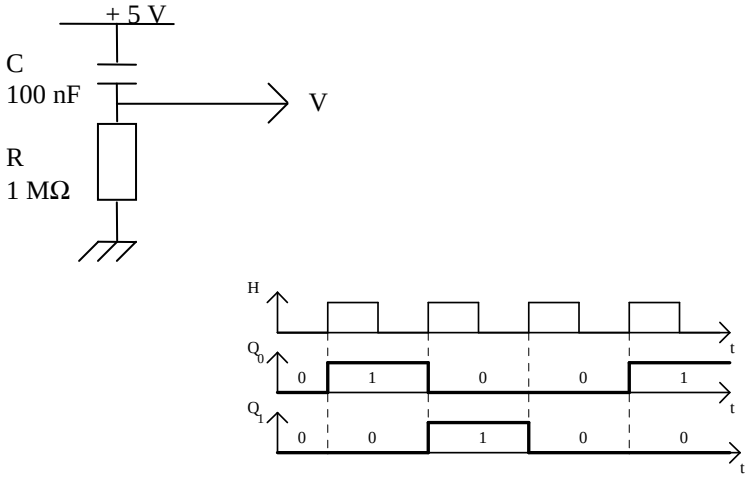
3. Bascules JK (1)

L'état initial $Q_1Q_0 = 11$ est obtenu (câblage uniquement) en envoyant *V* au *Set* des bascules, le *Reset* étant à 0 : (en simulation, ne pas câbler les entrées *R* et *S* revient à initialiser à $Q_1Q_0 = 00$).



4. Bascules JK (2)

L'état initial $Q_1Q_0 = 00$ est obtenu (câblage uniquement) en envoyant *V* au *Reset* des bascules, le *Set* étant à 0 : (en simulation, ne pas câbler les entrées *R* et *S* revient à initialiser à $Q_1Q_0 = 00$).



- On a un compteur par 3 : $Q_1Q_0 = 00 \rightarrow Q_1Q_0 = 01 \rightarrow Q_1Q_0 = 10 \rightarrow Q_1Q_0 = 00 \rightarrow$ ...

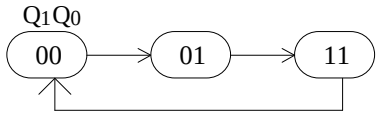
Avec l'initialisation $Q_1Q_0 = 11$, on a : $Q_1Q_0 = 11 \rightarrow Q_1Q_0 = 00 \rightarrow Q_1Q_0 = 01 \rightarrow Q_1Q_0 = 10 \rightarrow Q_1Q_0 = 00 \rightarrow$ ...

Le montage est autocorrecteur.

TD 4 CORRIGE. LOGIQUE SEQUENTIELLE 2

3. Compteur synchrone (Synthèse)

a) Cycle normal de comptage : $Q_1Q_0 = 00 \rightarrow 01 \rightarrow 11 \rightarrow 00 \rightarrow \ldots$



Rappel : Table de transitions d'une bascule JK :

Transition $Q_{n-1} \rightarrow Q_n$	***J***	***K***
$0 \rightarrow 0$	0	X
$0 \rightarrow 1$	1	X
$1 \rightarrow 1$	X	0
$1 \rightarrow 0$	X	1

. Table de Karnaugh établissant les entrées J_0K_0 de la bascule de sortie Q_0 :

J_0K_0 / Q_1 \ Q_0	0	1
0	1X	X0
1	XX	X1

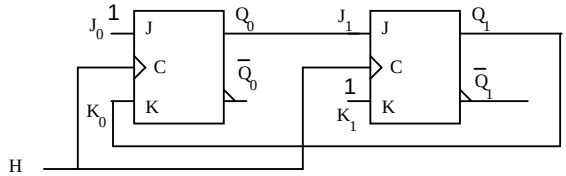
$$\begin{cases} J_0 = 1 \\ K_0 = Q_1 \end{cases}$$

. Table de Karnaugh établissant les entrées J_1K_1 de la bascule de sortie Q_1 :

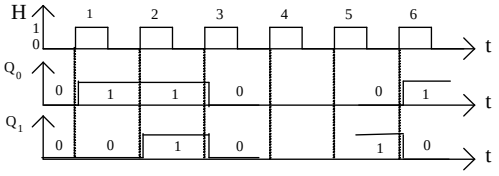
J_1K_1 / Q_1 \ Q_0	0	1
0	0X	1X
1	XX	X1

$$\begin{cases} J_1 = Q_0 \\ K_1 = 1 \end{cases}$$

. Schéma de câblage :

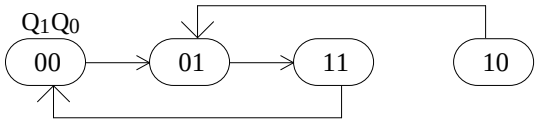


Chronogramme :



c) L'analyse (chronogramme) montre que l'état successeur de $Q_1Q_0 = 10$ est $Q_1Q_0 = 01$: $Q_1Q_0 = 10 \rightarrow 01$

Le compteur est donc **autocorrecteur** :

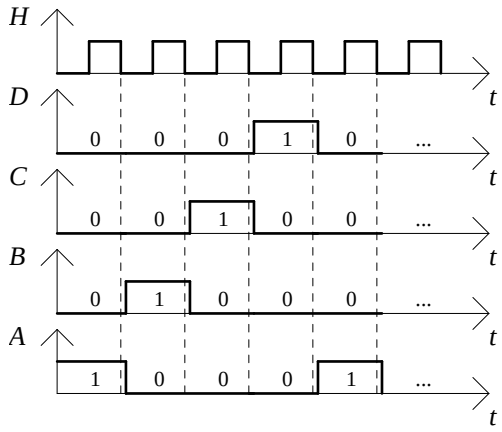
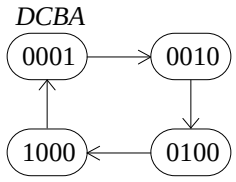
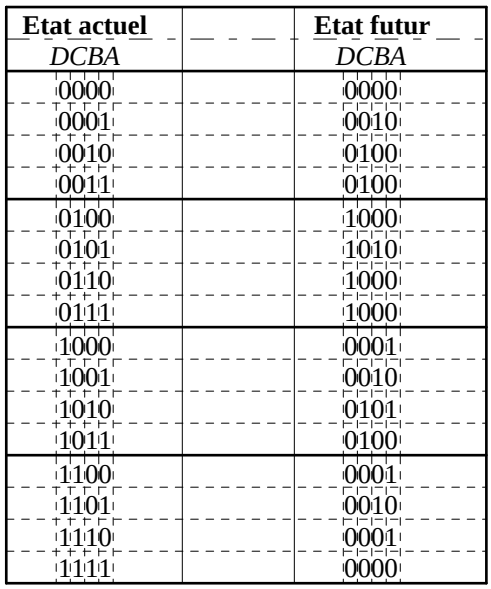
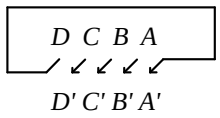
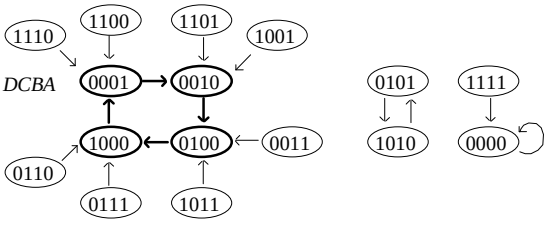


d) Si le compteur n'avait pas été autocorrecteur, le rendre autocorrecteur se ferait en reprenant la synthèse *a)* et en éliminant les choix *xx* effectués dans les tables de Karnaugh pour les forcer selon le cycle autocorrigé.

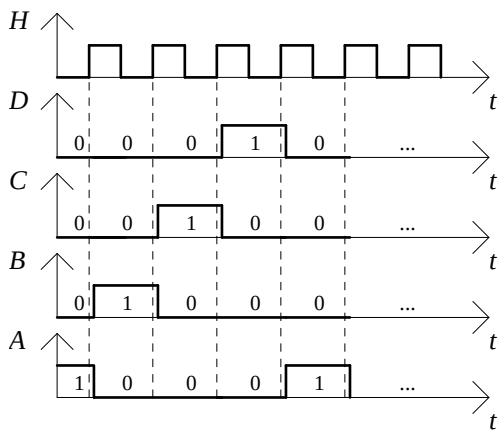
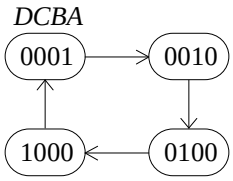
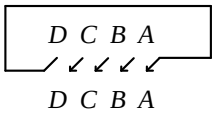
TD 4 ANNEXE CORRIGE. LOGIQUE SEQUENTIELLE 2

3. Compteurs synchrones en anneau et non bouclé (Analyse)

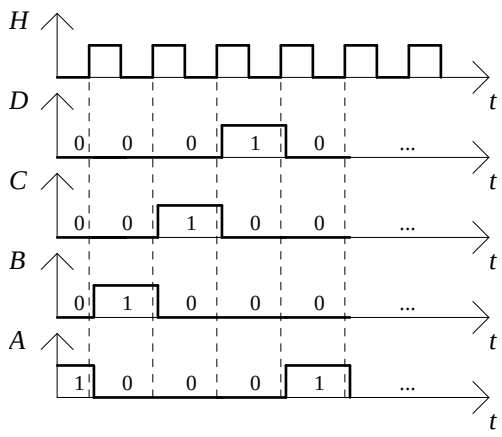
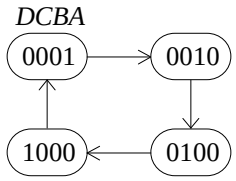
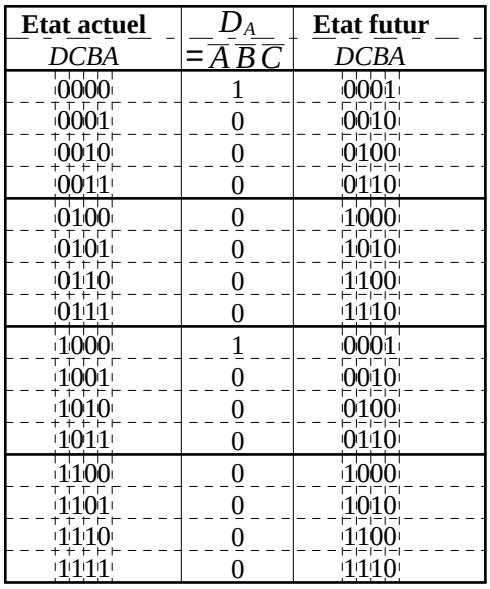
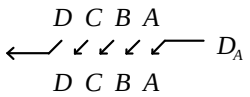
1. Compteur en anneau à bascules JK

	Résultat	*Commentaires*
1.		
2.		Compteur modulo 4 câblé en registre à décalage
3.		Le tableau est beaucoup plus vite rempli si on a remarqué que le compteur est un registre à décalage propageant un 0 si l'entrée est un 0, et une inversion de l'état si l'entrée est un 1 :  
4.	**NON**	L'état hors cycle $DCBA = 0000$ par exemple, ne peut être ramené dans le cycle normal : il boucle sur lui-même: $DCBA = 0000 \rightarrow 0000$ Autre contre-exemple : l'état hors cycle $DCBA = 0101$ ne peut être corrigé : il boucle sur lui-même : $DCBA = 0101 \rightarrow 1010 \rightarrow 0101$

2. Compteur en anneau à bascules D

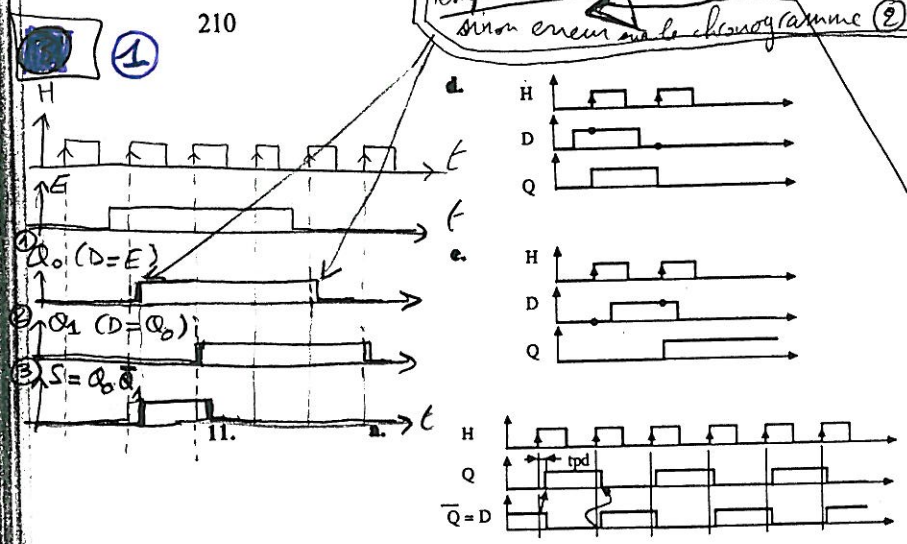
	Résultat	*Commentaires*
1.		
2.		Compteur modulo 4 câblé en registre à décalage
3.	**Etat actuel** *DCBA* / **Etat futur** *DCBA* 0000 → 0000 0001 → 0010 0010 → 0100 0011 → 0110 0100 → 1000 0101 → 1010 0110 → 1100 0111 → 1110 1000 → 0001 1001 → 0011 1010 → 0101 1011 → 0111 1100 → 1001 1101 → 1011 1110 → 1101 1111 → 1111	Le tableau est beaucoup plus vite rempli si on a remarqué que le compteur est un registre à décalage parfait : 
4.	**NON**	L'état hors cycle $DCBA = 0000$ par exemple, ne peut être ramené dans le cycle normal : il boucle sur lui-même: $DCBA = 0000 \rightarrow 0000$ Autre contre-exemple : l'état hors cycle $DCBA = 0011$ ne peut être corrigé : il boucle sur lui-même : $DCBA = 0011 \rightarrow 0110 \rightarrow 1100 \rightarrow 1001 \rightarrow 0011$

3. Compteur ouvert à bascules D

	Résultat	*Commentaires*
1.		
2.		Compteur modulo 4 câblé en registre à décalage
3.		Le tableau est beaucoup plus vite rempli si on a remarqué que le compteur est un registre à décalage parfait seulement sur les 3 variables C, B, A mais pas D: A devient $D_A = \overline{A}\,\overline{B}\,\overline{C}$ 
4.	**OUI**	Il n'y a pas de contre-exemple : tout état hors cycle est ramené dans le cycle normal du compteur.

Etat actuel *DCBA*	$D_A = \overline{A}\,\overline{B}\,\overline{C}$	**Etat futur** *DCBA*
0000	1	0001
0001	0	0010
0010	0	0100
0011	0	0110
0100	0	1000
0101	0	1010
0110	0	1100
0111	0	1110
1000	1	0001
1001	0	0010
1010	0	0100
1011	0	0110
1100	0	1000
1101	0	1010
1110	0	1100
1111	0	1110

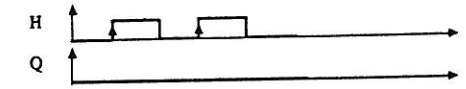
Temps de retard à faire figurer obligatoirement sinon erreur sur le chronogramme ②

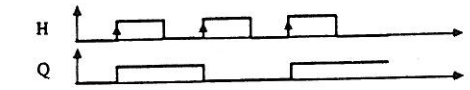


d.

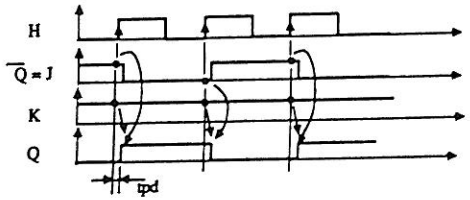
e.

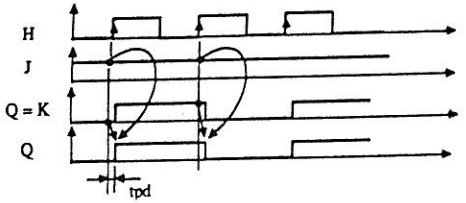
Pour mettre en évidence le phénomène décrit ci-dessus, il est indispensable de représenter la sortie Q avec le décalage temporel dû au principe de causalité (t_{pd} = temps de réaction de la bascule).

b.


c.


d. *Idem* c.

e.


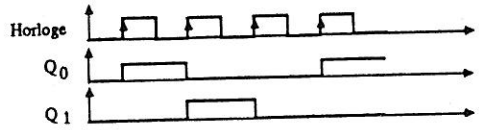
f.


12a. Initialement $Q_1 = Q_0 = 0$.
Donc $J_0 = 1$ et $K_0 = 1$ ⇒ La sortie changera après une impulsion d'horloge.
Et $J_1 = 0$ et $K_1 = 1$ ⇒ La sortie prendra la valeur 0 après une impulsion d'horloge.

b. Après une impulsion d'horloge $Q_0 = 1$ et $Q_1 = 0$.
Donc $J_0 = 1$ et $K_0 = 1$ ⇒ La sortie changera après une impulsion d'horloge.
Et $J_1 = 1$ et $K_1 = 1$ ⇒ De même la sortie changera après une impulsion d'horloge.

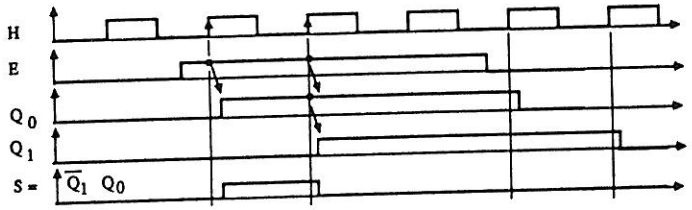
c. Après la seconde impulsion d'horloge $Q_0 = 0$ et $Q_1 = 1$.
Donc $J_0 = 0$ et $K_0 = 1$ ⇒ La sortie passera à 0 ainsi que Q_1, puisque $J_1 = 0$ et $K_1 = 1$.

d. D'où le chronogramme :



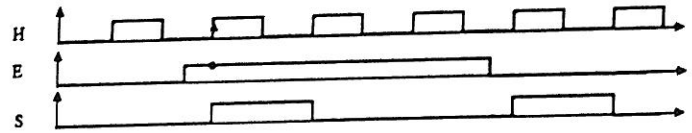
Les bascules sont synchronisées sur des fronts montants.

① On met en évidence sur les chronogrammes suivants les retards de chaque bascule, celles-ci étant synchronisées sur le front montant de l'horloge.



Ce dispositif permet de délivrer une impulsion d'une durée égale à la période d'horloge et synchronisée sur celle-ci à chaque front montant de l'entrée E.

14. Les sorties Q_0 et Q_1 évoluent comme dans l'exercice 13 d'où :



Cette fois une impulsion est délivrée à chaque front de l'information E.

I D4 - Logique séquentielle 2

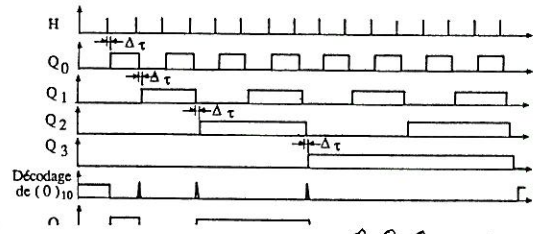
34

16. $(Q_2Q_1Q_0)_{t=0} = 0\,0\,0$
Après une impulsion d'horloge $Q_2Q_1Q_0 = 0\,0\,1$
Après la deuxième impulsion d'horloge $Q_2Q_1Q_0 = 0\,1\,0$
Après la troisième impulsion d'horloge $Q_2Q_1Q_0 = 0\,1\,1$
Après la quatrième impulsion d'horloge $Q_2Q_1Q_0 = 1\,0\,0$
Après la cinquième impulsion d'horloge $Q_2Q_1Q_0 = 0\,0\,0$ et le cycle recommence

→ Compteur modulo 5

$(Q_2Q_1Q_0)_{t=0} = 0\,0\,0$
Après la première impulsion d'horloge $Q_2Q_1Q_0 = 0\,0\,1$
Après la deuxième impulsion d'horloge $Q_2Q_1Q_0 = 0\,1\,0$
Après la troisième impulsion d'horloge $Q_2Q_1Q_0 = 1\,0\,0$
Après la quatrième impulsion d'horloge $Q_2Q_1Q_0 = 1\,0\,1$
Après la cinquième impulsion d'horloge $Q_2Q_1Q_0 = 1\,1\,0$
Après la sixième impulsion d'horloge $Q_2Q_1Q_0 = 0\,0\,0$ et le cycle recommence

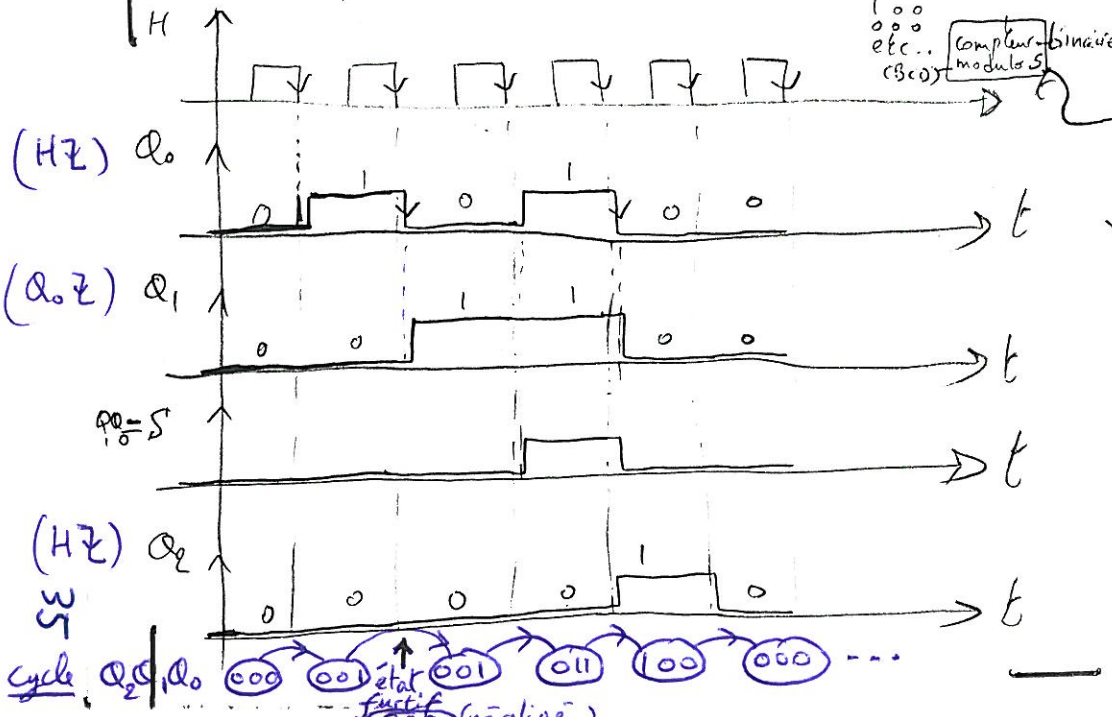
$Q_1Q_0 = S$



$Q_2Q_1Q_0 =$ 000 001 010 011 100 000 etc..

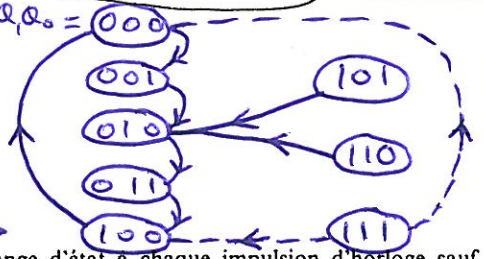
compteur binaire modulo 5

car au bout de 5 séquences des sorties $Q_2Q_1Q_0$, on retombe sur l'état initial de $Q_2Q_1Q_0$. → Automate des états:



19a. La suite des états est :

Q_2	Q_1	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
0	0	0
.	.	.

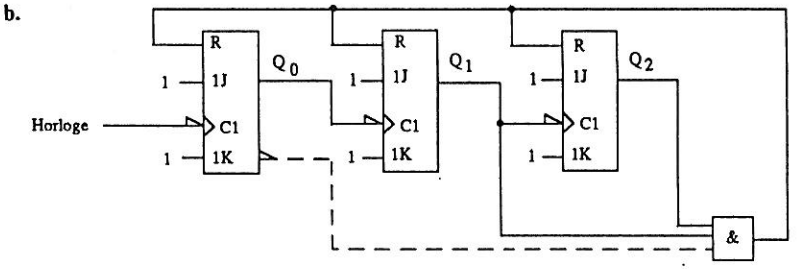


le compteur est autocorrecteur

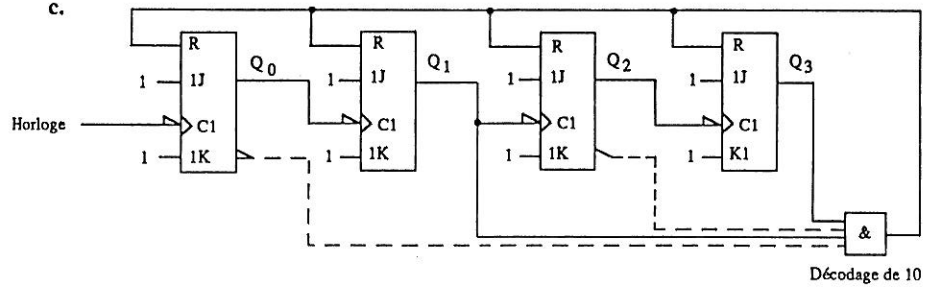
La première bascule change d'état à chaque impulsion d'horloge sauf si le contenu du compteur est 4. Ce qui impose le schéma :

Remarque : l'état initial 111 fait intervenir la combinaison interdite R = S = 1 → Q_n = 0 ou 1 selon la technologie de la bascule RS (à Nand ou à Nor)
→ 2 possibilités (en pointillé ----)

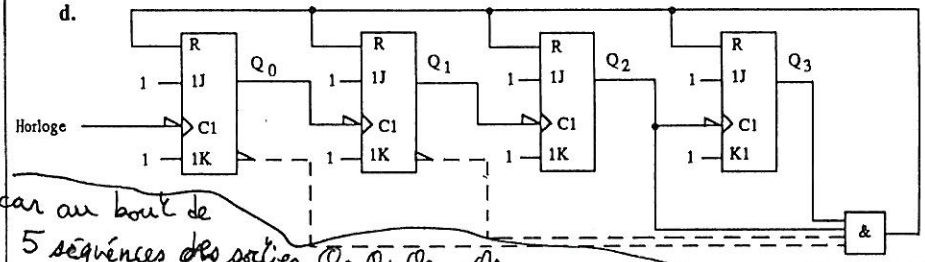
b.



c.



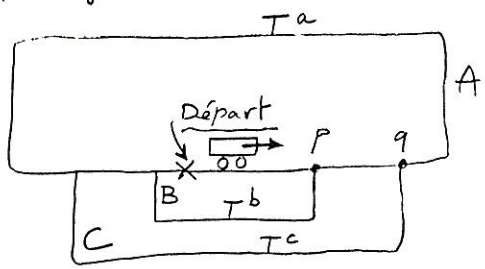
d.



3

6.2. Exemple

Soit un train électrique devant effectuer 3 boucles A, B, C sélectionnables par aiguillages p et q. Le passage dans une boucle est détecté par un contact (T) remontant après le passage du train.



aiguillages :

$\begin{vmatrix} p \\ q \end{vmatrix} = \begin{cases} 0 : \text{non dévié} \\ 1 : \text{dévié} \end{cases}$

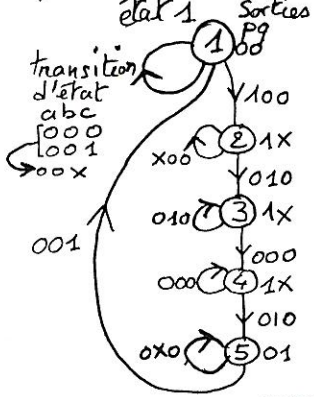
contacts :

$\begin{vmatrix} a \\ b \\ c \end{vmatrix} = \begin{cases} 0 : \text{repos} \\ 1 : \text{passage du train (retombée à 0 après passage)} \end{cases}$

Itinéraire désiré : [A, B, B, C]

état initial : Train au Départ, sorties p et q à 0

Automate (des états) / Graphe de fluence : (entrées du système : a, b, c ; sorties ——— : p, q)



la 2nde transition possible 001 pour l'état 1 a été rajoutée à la fin, d'après la transition entre l'état 5 et l'état 1

② table des états :

état résultant \ transition (c b a)	000	001	010	011	100	101	110	111	sorties p	sorties q
1	1	2					1		0	0
2	2	2	3						1	X
3	4		3						1	X
4	4		5						1	X
5	5		5				1		0	1

Les cases vides sont des X (cases indifférentes)

③ simplification de la table ; Recherche des états équivalents :

2 états sont équivalents s'ils ont : | – mêmes sorties
et | – mêmes transitions

états pouvant être équivalents : 2–3 si 2–4 le sont
2–4 si 3–5 le sont
3–4 si 3–5 le sont

or 3 et 5 ne sont pas équivalents
→ 2–4 ne le sont pas
→ 2–3 ne le sont pas

→ pas d'états équivalents → pas de simplification de la table
(la simplification aurait conduit à remplacer dans la table les états équivalents à un état donné par cet état donné et réunir les transitions correspondantes)

④ attribution des variables de sortie des bascules (Q_i) : table des adresses

On a d'autant plus besoin de variables Q_i qu'il y a d'états à adresser dans la proportion : | n états à adresser → $n = 2^m$ | m variables Q_i

ici : n = 5 → au moins 3 variables sont nécessaires : $Q_2 Q_1 Q_0$ pour le codage

$Q_2 Q_1 Q_0$ \ (c b a)	000	001	010	011	100	101	110	111	p	q
000 ≡ 1	001	001		011				000	0	0
001 ≡ 2	010			011					1	X
011 ≡ 3	010			110					1	X
010 ≡ 4	110			110				000	1	X
110 ≡ 5									0	1

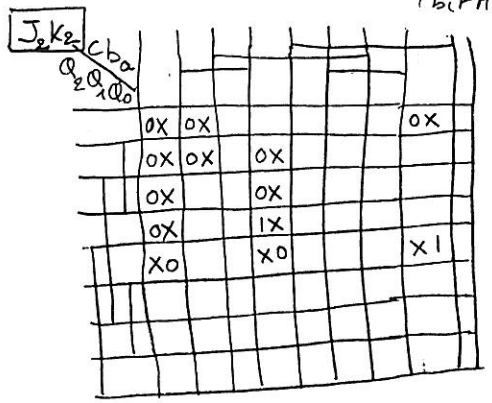
Exemple : utilisation de bascules JK

Table de Karnaugh des bascules $J_i K_i$: 1 bascule JK par variable Q

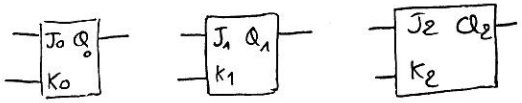
3 variables Q → 3 bascules JK :

ex : table de Karnaugh de $J_2 K_2$ (bit MSB)

(rappel : transition | JK : 0→0 : 0X ; 0→1 : 1X ; 1→1 : X0 ; 1→0 : X1)



→ $J_2 = \dots$
$K_2 = \dots$



entrées : a → … ; b → … ; c → …

sorties : … → p ; … → q

⑥ Equation des sorties :

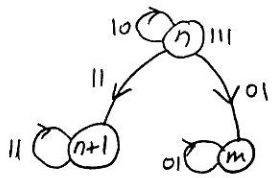
P9 \ Q1Q0 / Q2	00	1X	1X	1X
	XX	XX	XX	01

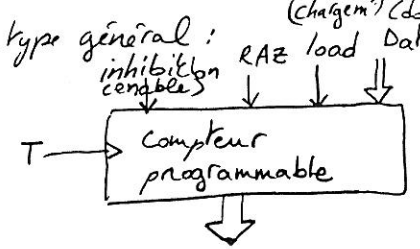
→ $p = Q_0 + Q_1 \bar{Q}_2$
$q = Q_1$

Séquenceur programmable

13.6.3 Utilisation d'un compteur programmable

Toute transition d'état est du type général :





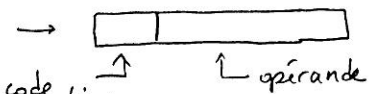
→ Commandes appliquées : - on reste dans l'état n : → enable
- on passe à n+1 : → rien (fonctionnement normal du compteur)
- on passe à m : load, data
- initialisation du compteur : RAZ

Plus de table de Karnaugh à calculer, ni entrées JK, mais il faut programmer les bonnes instructions sur les entrées | Enable, RAZ, Load, Data

① Codage et décodage des actions

- actions sur les sorties : mise à 0 / mise à 1) → il suffit d'1 bit

mais il faut préciser sur quelle sortie porte l'action :

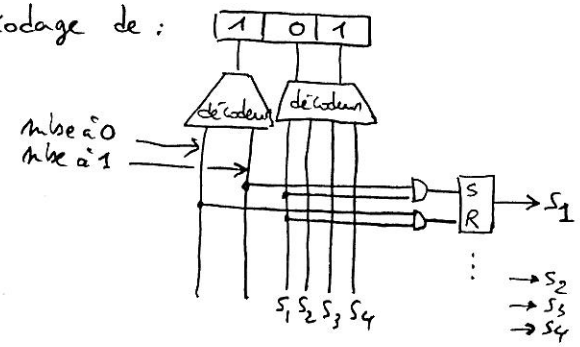


code opération : 1 : mise à 1
0 : mise à 0

opérande : 00 : sortie s_1
01 : sortie s_2
10 : sortie s_3
11 : sortie s_4
(exemple à 4 sorties)

ex: mise à 1 de S_2 :

→ décodage de : 1 0 1



- opérations
  - inhibition conditionnelle
  - rupture de séquence (load) | conditionnelle ou inconditionnelle
  - mise à 0 | conditionnelle ou inconditionnelle

→ 5 opérations : → 3 bits pour le code opération

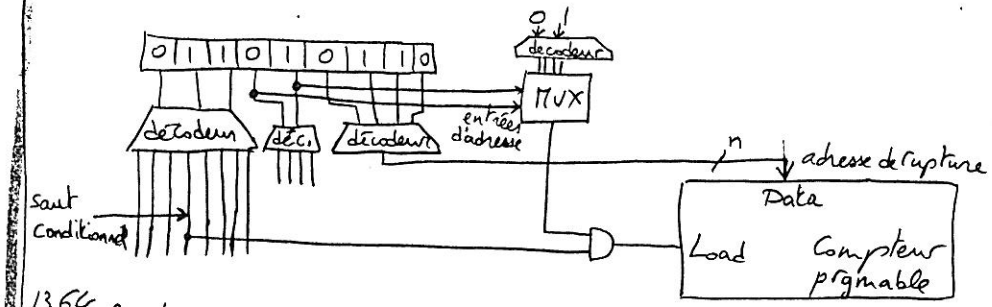
opération :		
	000	mise à 0 d'une sortie
	001	mise à 1 d'une sortie
	010	inhibition conditionnelle
	011	rupture conditionnelle
	100	rupture inconditionnelle
	101	RAZ conditionnelle
	110	RAZ inconditionnelle

opérande : sortie
condition
adresse de rupture

ex : rupture de séquence à l'adresse 0110 (conditionnel)

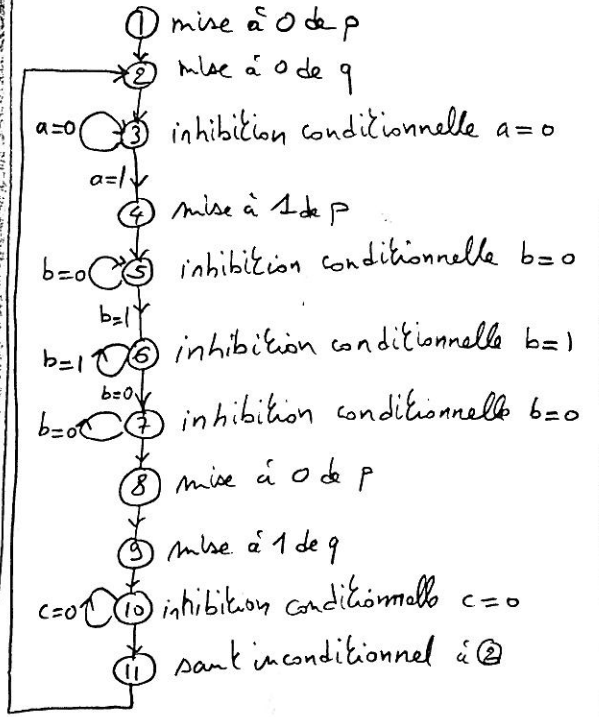
→ décodage de : 011 01 0110

condition codée sur 2 bits



13.64.

application sur l'exemple du train électrique :

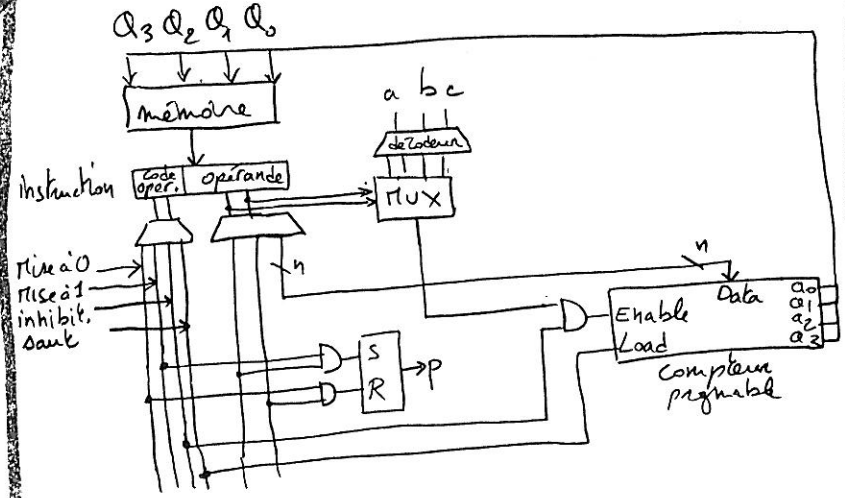


Codage :

actions :	mnémonique :	code binaire
Mise à Zero de sorties	MAZ	00
Mise à Un de sorties	MAU	01
Inhibition Conditionnelle	INH	10
Saut Inconditionnel	JMP	11
opérandes :		
sortie p	P	00
sortie q	Q	01
Conditions :		
a=0	A0	00
b=0	B0	01
c=0	C0	10
b=1	B1	11
adresse de saut 2	2	00

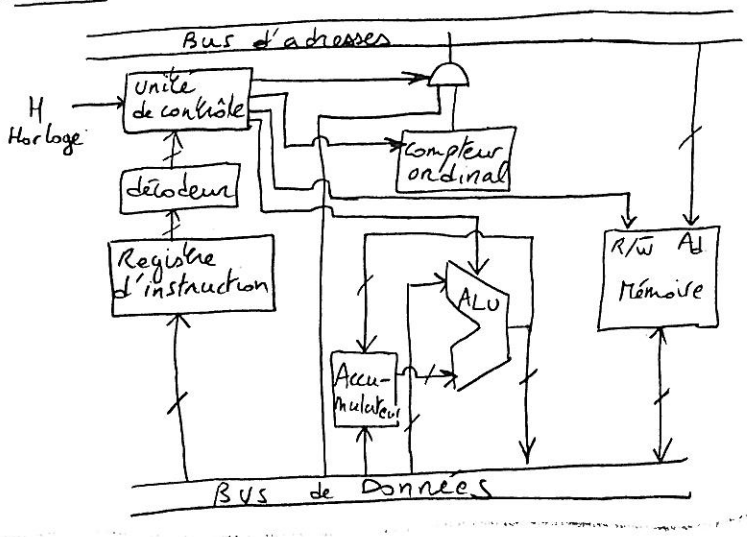
Programme :	état	mnémonique	code binaire
	0000	MAZ P	0000
	0001	MAZ Q	0001
	0010	INH A0	1000
	0011	MAU P	0100
	0100	INH B0	1001
	0101	INH B1	1011
	0110	INH B0	1001
	0111	MAZ P	0000
	1000	MAU Q	0101
	1001	INH C0	1010
	1010	JMP 2	1100

Matérialisation :

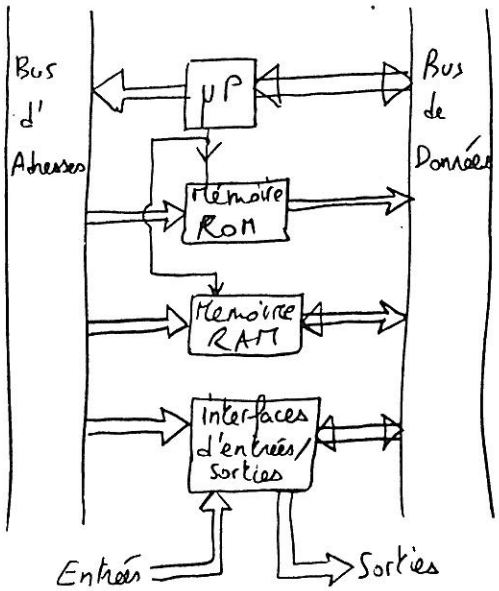


13.6.5 Utilisation d'un microprocesseur

Structure



Circuits associés :



(µP ≡ micro-processeur)

13.6.6 Utilisation d'un micro-contrôleur

Un microcontrôleur est l'ensemble microprocesseur et ses interfaces d'entrées/sorties intégrées sur dans une même structure. (interfaces = ports)

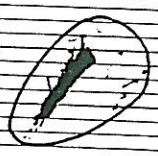
13.7. Conclusion

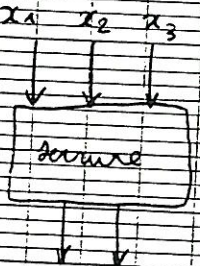
Logique câblée ≡ utilisation de portes combinatoires et séquentielles
- Avantage : rapidité du système
- inconvenient : structure figée → manque de souplesse pour une modification - adaptation du système

Logique programmée ≡ utilisation de séquenceur (compteur) programmable ou d'un microprocesseur ou microcontrôleur
- inconvenient = plus lent que la logique câblée car décodage instruction
- avantage = plus souple à modifier et concevoir

4

ex: * cahier des charges imposé :





état initial $y_1 y_2 = 00$
$x_1 x_2 x_3 = 000$

x_i ⊓_ 1 fois
1 seule x_i à 1 à la fois

séquence à reconnaître : x_1 puis x_2 puis x_3 à 1
$y_1 = 1$ à la fin de la séquence ⊓_ x_3
⊓_ y_1

1ère autre séquence : $y_2 = 1$
en particulier si x_1 a été mis à 1, on peut pas le remettre à 1 une 2ème fois.

* On construit le graphe de fluence correspondant.
Le cahier des charges nous aide en imposant l'état initial à 000.

Sorties y_1 y_2
Entrées x_1 x_2 x_3

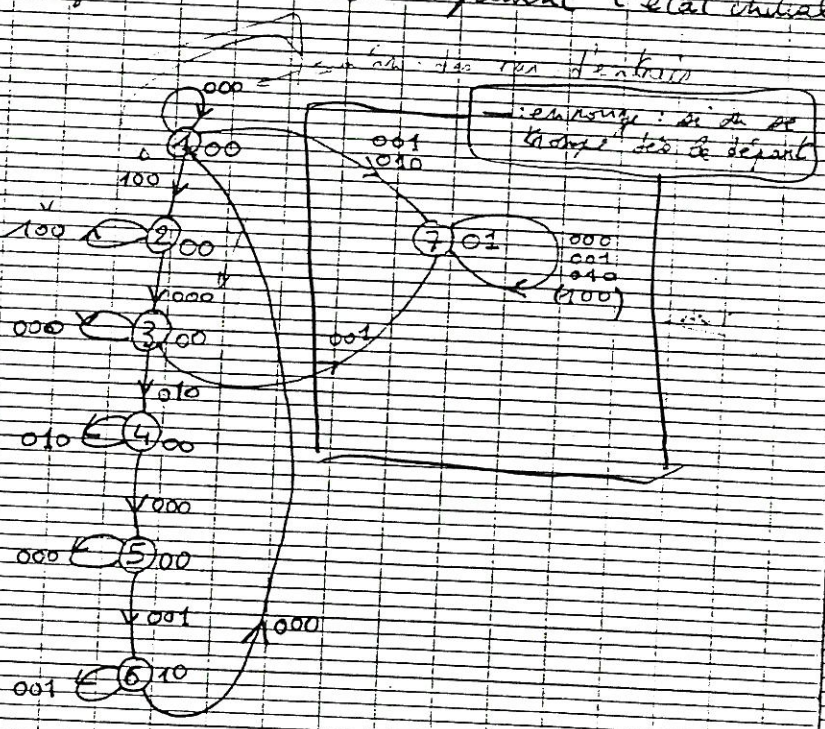


Table des états :

$x_3x_2x_1$									y_1	y_2
1	1	2		7				7	0	0
2	3	2							0	0
3	3			4				7	0	0
4	5			4					0	0
5	5							6	0	0
6	1							6	1	0
7	7	7		7				7	0	1

les cases vides sont des ϕ. (cases indifférentes elles valent n'importe quoi (ce qui nous arrange))

on est sûr 6 et 7 sont éq. à aucun autre
les autres peuvent être équiv. 2 à 2.
pour le voir, on construit un tableau :

paire d'états →

	~~1.2~~	~~1.3~~	~~1.4~~	~~1.5~~	2.3	~~2.4~~	~~2.5~~	~~3.4~~	~~3.5~~	4.5
~~1.2~~		X								
~~1.3~~										
~~1.4~~										
~~1.5~~										
2.3										
~~2.4~~									X	
~~2.5~~									X	
~~3.4~~									X	
~~3.5~~									X	
4.5										

1.2 eq. si 1.3 équiv (équiv. conditionnelle) symbolisé par une croix X
1.3 non eq car 4 et 7 ne sont pas eq. (7 eq. à aucun autre)
⇒ 1.2 non eq. symbol. par croix X
1.5 non eq. car 6 et 7 non eq.
2.3 : rien ne s'oppose à ce qu'il soit éq. on décrète qu'ils le sont car ça nous arrange. (on encadre ▭)
3.5 non eq. car ça contient 7. ⇒ on barre ~~3.5~~ ~~3.5~~ puis ~~2.4~~ ~~2.5~~ ~~3.4~~

états : 2-3 équivalents
4-5 équivalents

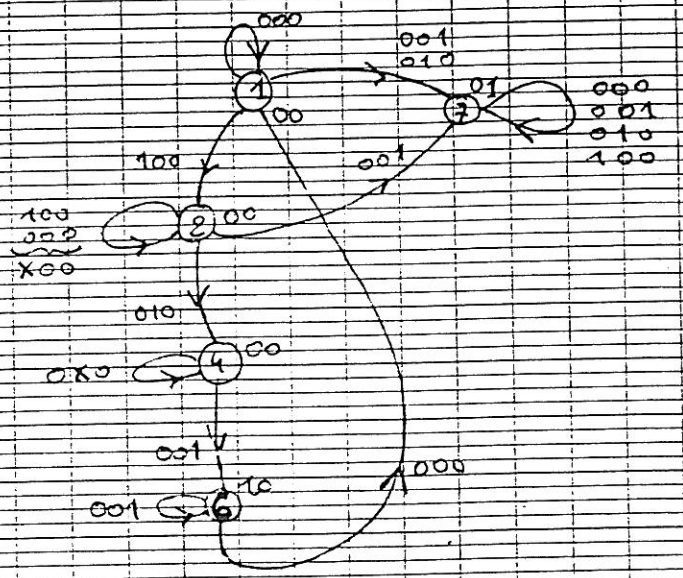
nouvelle table des états avec simplification dûe aux équiv.

Table simplifiée :

x_1x_2 / x_3					y_1y_2
1	1	2	7	7	0 0
2	2	2	4	7	0 0
4	4		4	6	0 0
6	1			6	1 0
7	7	7	7	7	0 1

ce qui peut nous troubler : on se trouve maintenant avec un nouvel état 2, il y a des transitions supplémentaires

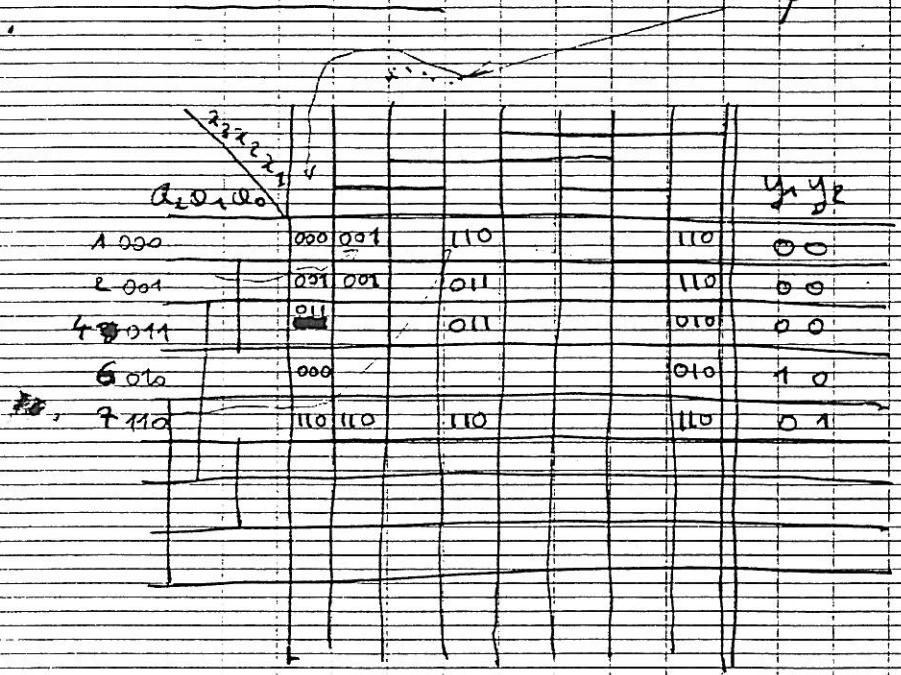
pour le voir, on fait un nouveau schéma de fluence :



d. attribution des variables secondaires → table des adres

on a besoin d'autant de var. second. qu'il y a d'état.

table des adresses : on note l'état futur des états.



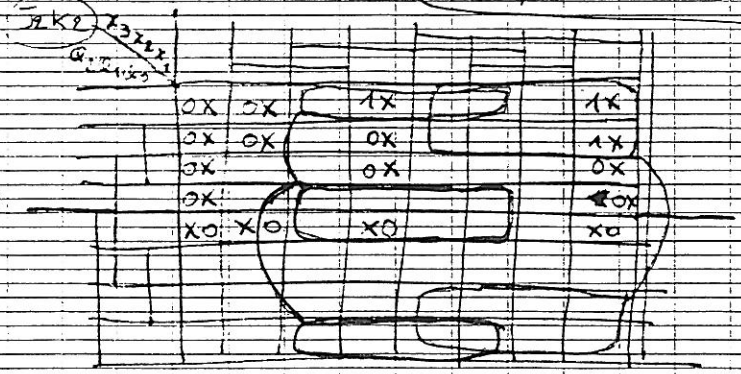
répétition de la table de la page précédente avec codage des états :

1 = 000
2 = 001
4 = 011
6 = 010
7 = 110

e. Bascules JK choisies pour la réalisation
table de Karnaugh des excitat. secondaires ($J_i K_i$)
3 basculeurs (J, K) 1 par état secondaire

$J_0 K_0$ $J_1 K_1$ $J_2 K_2$

table de Karnaugh de ($J_2 K_2$) par exemple



par ex.

transition	J K
0 → 0	0 X
0 → 1	1 X
1 → 1	X 0
1 → 0	X 1

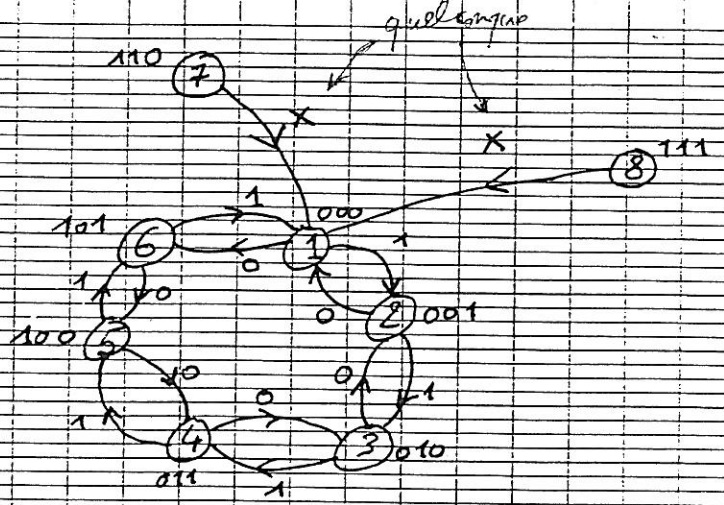
les cases vides sont indifférentes à nous de faire ces meilleurs regroupements

(la prochaine fois on verra le bloc 2.)

(compteur à cycle incomplet mais ici, on tient compte de l'initialisation ⑦ et ⑧)

le 15/1/85 entrée C

sorties $y_2 y_1 y_0$

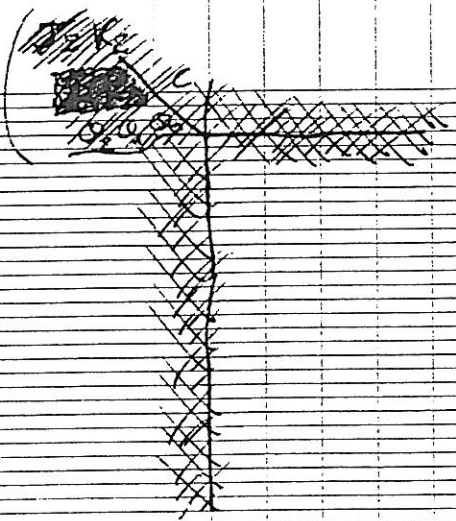


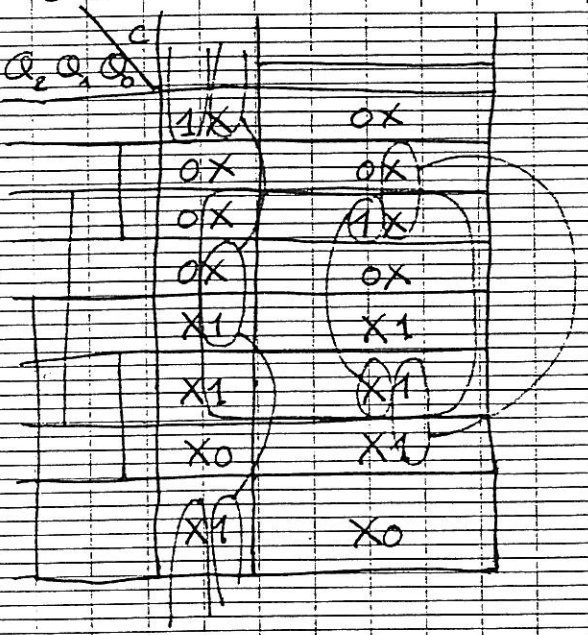
C			y_2 y_1 y_0
1	6	2	0 0 0
2	1	3	0 0 1
3	2	4	0 1 0
4	3	5	0 1 1
5	4	6	1 0 0
6	5	1	1 0 1
7	1	1	1 1 0
8	1	1	1 1 1

le choix des variables secondaires est libre – on prend ce qu'on veut – autant choisir ce qui nous simplifie (var. sec. = sorties) dans le cas présent. (mais pas obligatoire)

$Q_2 Q_1 Q_0$ \ C			y_2 y_1 y_0
1	101	001	0 0 0
2	000	010	0 0 1
4	010	100	0 1 1
3	001	011	0 1 0
7	000	000	1 1 0
8	000	000	1 1 1
6	100	000	1 0 1
5	011	101	1 0 0

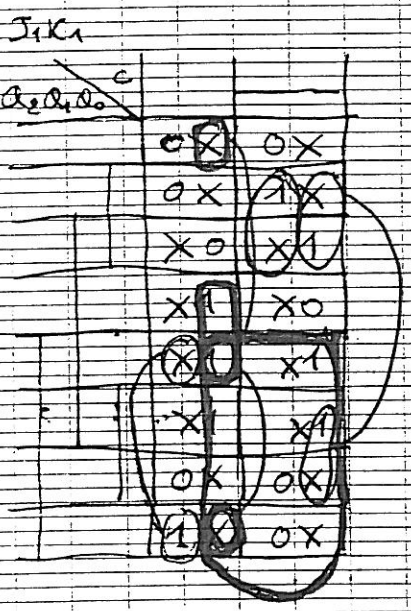
$\Rightarrow$ $y_2 = Q_2$, $y_1 = Q_1$, $y_0 = Q_0$

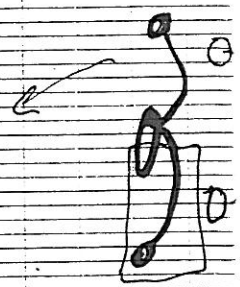




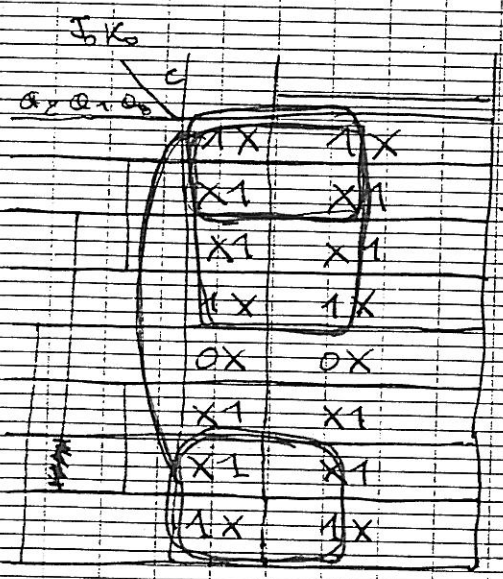
$$J_2 = \bar{c}\,\bar{Q}_1\bar{Q}_0 \vee c\,Q_1Q_0$$
$$K_2 = Q_1 \vee \bar{c}\,\bar{Q}_0 \vee cQ_0$$

transition	J K
0 → 0	0 X
0 → 1	1 X
1 → 1	X 0
1 → 0	X 1





$$J_1 = \bar{c}\,Q_2\bar{Q}_0 \vee c\bar{Q}_2Q_0$$
$$K_1 = \bar{c}\bar{Q}_0 \vee Q_2 \vee cQ_0$$



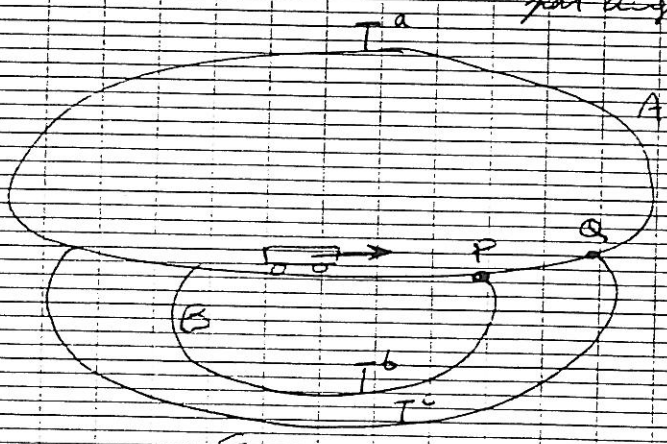
$$\begin{cases} J_2 = \bar{c}\,\bar{Q}_1\bar{Q}_0 \vee c\,Q_1 Q_0 \\ K_2 = Q_1 \vee c\,\bar{Q}_0 \vee c\,Q_0 \end{cases}$$

$$\begin{cases} J_0 = \bar{Q}_2 \vee \bar{Q}_1 \\ K_0 = 1 \end{cases}$$

$$y_2 = Q_2$$
$$y_1 = Q_1$$
$$y_0 = Q_0$$

ex: train électrique. 3 boucles: A, B, C qu'on peut relier par aiguillage par P et Q

cahier des charges:



T : contact le contact mémorise une fois que le train est passé dessus

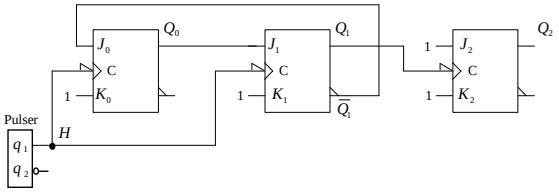
P et Q : var. logiques $\begin{cases} P \\ Q \end{cases}$ $\begin{cases} 0 : \text{non dévié} \\ 1 : \text{dévié} \end{cases}$

T: contact - a, b, c $\begin{cases} 0 & \text{au repos} \\ 1 & \text{passage du train} \end{cases}$

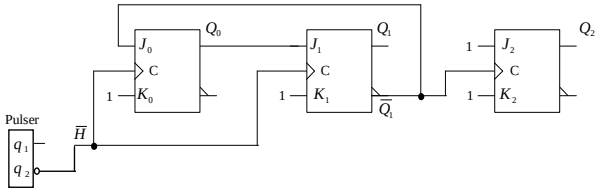
itinéraire → A, B, B, C

TP 4 CORRIGE. LOGIQUE SEQUENTIELLE 2

3. Compteurs



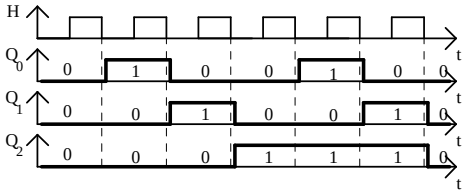
(1) ≡



(2)

Le schéma initial (1) peut être remplacé par le schéma (2) identique, mettant en jeu des bascules JK > 0 edge triggered (4027). Si le choix du schéma (1) est fait, l'utilisation de bascules JK > 0 edge triggered (4027) implique d'intercaler avant chaque entrée d'horloge des bascules un circuit inverseur 4049 (même famille CMOS que les bascules JK).

$$\begin{cases} J_0 = \overline{Q_1} \\ K_0 = 1 \\ H \downarrow \end{cases} \quad \begin{cases} J_1 = Q_0 \\ K_1 = 1 \\ H \downarrow \end{cases} \quad \begin{cases} J_2 = 1 \\ K_2 = 1 \\ Q_1 \downarrow \end{cases}$$



Compteur modulo 6

Etat initial : $Q_2Q_1Q_0$ = **000**
Après la 1ère impulsion de H : $Q_2Q_1Q_0$ = **001**
Après la 2nde impulsion de H : $Q_2Q_1Q_0$ = **010**
Après la 3ème impulsion de H : $Q_2Q_1Q_0$ = **100**
Après la 4ème impulsion de H : $Q_2Q_1Q_0$ = **101**
Après la 5ème impulsion de H : $Q_2Q_1Q_0$ = **110**
Après la 6ème impulsion de H : $Q_2Q_1Q_0$ = **000** **... et le cycle recommence**

4. Synthèse de Compteur 2 bits

Cycle voulu : Q_1Q_0 = 00 → 01 → 11 → 10 → 00 → …

Rappel : Table de transitions d'une bascule JK :

Transition $Q_{n-1} \rightarrow Q_n$	***J***	***K***
0 → 0	0	X
0 → 1	1	X
1 → 1	X	0
1 → 0	X	1

. Table de Karnaugh établissant les entrées J_0K_0 de la bascule de sortie Q_0 :

J_0K_0 / Q_1 \ Q_0	0	1
0	1X	X0
1	0X	X1

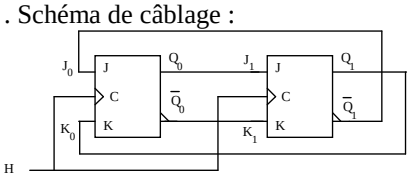
$$\begin{cases} J_0 = \overline{Q_1} \\ K_0 = Q_1 \end{cases}$$

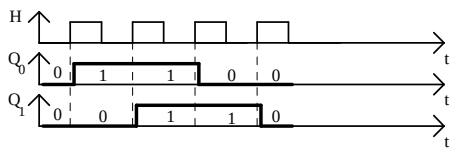
. Table de Karnaugh établissant les entrées J_1K_1 de la bascule de sortie Q_1 :

J_1K_1 / Q_1 \ Q_0	0	1
0	0X	1X
1	X1	X0

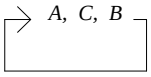
$$\begin{cases} J_1 = Q_0 \\ K_1 = \overline{Q_0} \end{cases}$$

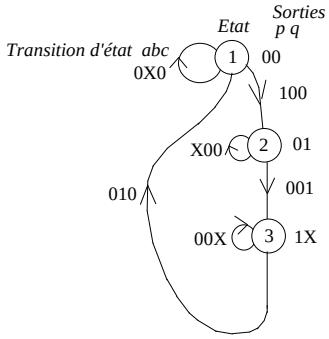
. Schéma de câblage :





5. Séquenceur Train électrique

a) Automate des états correspondant à la trajectoire 



b) Synthèse avec des bascules *JK* synchrones positive *edge triggered*, permettant de commander les aiguilleurs *p* et *q*

- *Table des états*

Etat résultant \ transition (c b a)	000	100	110	010	011	111	101	001	Sorties p	Sorties q
1	1	2		1					0	0
2	2	2						3	0	1
3	3			1				3	1	X

- *Pas de simplification de la table des états (pas d'états équivalents)*

- *Variables de sortie des bascules*

3 états → 2 variables Q_1Q_0

Etat résultant \ transition (c b a) / état Q_1Q_0	000	100	110	010	011	111	101	001	Sorties p	Sorties q
00	00	01		00					0	0
01	01	01						11	0	1
11	11			00				11	1	X
10										

- *Tables de Karnaugh des bascules* J_iK_i

J_0K_0 transition (c b a) / état Q_1Q_0	000 (a b c)	100	110	010	011	111	101	001	Sorties p	q
00	0X	1X		0X					0	0
01	X0	X0						X0	0	1
11	X0			X1				X0	1	X
10										

Non simplification (car trop de variables pour appliquer la méthode de Karnaugh) : $\begin{cases} J_0 = a\,\overline{b}\,\overline{c}\,\overline{Q}_1\,\overline{Q}_0 \\ K_0 = \overline{a}\,b\,\overline{c}\,Q_1\,Q_0 \end{cases}$

J_1K_1 transition (c b a) / état Q_1Q_0	000 (a b c)	100	110	010	011	111	101	001	Sorties p	q
00	0X	0X		0X					0	0
01	0X	0X						1X	0	1
11	X0			X1				X0	1	X
10										

Non simplification (car trop de variables pour appliquer la méthode de Karnaugh) : $\begin{cases} J_1 = \overline{a}\,\overline{b}\,c\,\overline{Q}_1\,Q_0 \\ K_1 = \overline{a}\,b\,\overline{c}\,Q_1\,Q_0 = K_0 \end{cases}$

- *Equation des sorties*

p q ; Q_1 \ Q_0	0	1
0	00	01
1	XX	1X

Après simplification : $\begin{cases} p = Q_1 \\ q = Q_0 \end{cases}$

TD 5 CORRIGE. VHDL

[Réf. Livre VHDL (Meaudre & all …)]

1. Programme VHDL des Opérateurs fondamentaux : NON, ET, OU

1. Descriptions comportementales

NON 1

```
-- inverseur  (ceci est un commentaire)
ENTITY non1 IS
      PORT (e : IN BIT;     -- entree
            s : OUT BIT);   -- sortie
END non1;
ARCHITECTURE pleonasme OF non1 IS
BEGIN
      s <= NOT e;
END pleonasme;
```

NON 2

```
-- inverseur
ENTITY non2 IS
      PORT (e : IN BIT;     -- entree
            s : OUT BIT);   -- sortie
END non2;
ARCHITECTURE logique OF non2 IS
BEGIN
      s <= '1' WHEN (e='0') ELSE '0';
END logique;
```

ET 1

```
-- operateur ET
ENTITY et1 IS
      PORT (e1, e2 : IN BIT;      -- entree
              s : OUT BIT);           -- sortie
END et1;
ARCHITECTURE pleonasme OF et1 IS
BEGIN
      s <= e1 AND e2;
END pleonasme;
```

ET 2

```
-- operateur ET
ENTITY et2 IS
      PORT (e1, e2 : IN BIT;     -- entree
            s : OUT BIT);        -- sortie
END et2;
ARCHITECTURE logique OF et2 IS
BEGIN
      s <= '0' WHEN (e1='0' OR e2='0') ELSE '1';
END logique;
```

ET 3

```
-- ET
ENTITY et3 IS
	PORT (e1, e2 : IN BIT;		-- entree
		s : OUT BIT);	-- sortie
END et3;

ARCHITECTURE abstrait OF et3 IS
BEGIN
	PROCESS (e1,e2)
	BEGIN
	  IF (e1= '0' OR e2='0') THEN
	    s <= '0';
	  ELSE
	    s <= '1';
	  END IF;
	END PROCESS;
END abstrait;
```

OU 1

```
-- operateur OU
ENTITY ou1 IS
		PORT (e1, e2 : IN BIT;	-- entree
			s : OUT BIT);			-- sortie
END ou1;
ARCHITECTURE pleonasme OF ou1 IS
BEGIN
		s <= e1 OR e2;
END pleonasme;
```

OU 2

```
-- operateur OU
ENTITY ou2 IS
	PORT (e1, e2 : IN BIT;	-- entree
	    s : OUT BIT);	-- sortie
END ou2;
ARCHITECTURE logique OF ou2 IS
BEGIN
	s <= '0' WHEN (e1='0' AND e2='0') ELSE '1';
END logique;
```

OU 3

```
-- OU
ENTITY ou3 IS
		PORT (e : IN BIT_VECTOR (0 TO 1);
		    s : OUT BIT);
END ou3;
ARCHITECTURE abstrait OF ou3 IS
BEGIN
		PROCESS (e)
		BEGIN
			CASE e IS
				WHEN "00" =>
					s <= '0';
				WHEN OTHERS =>
					s <= '1';
			END CASE;
		END PROCESS;
END abstrait;
```

2. Programme VHDL de l'Opérateur OU exclusif

1. Description structurelle

OUX 1

```
-- operateur OU EXCLUSIF
ENTITY oux1 IS
        PORT (e1, e2 : IN BIT;      -- entree
              s : OUT BIT);                 -- sortie
END oux1;

ARCHITECTURE pleonasme OF oux1 IS
BEGIN
        s <= e1 XOR e2;
END pleonasme;
```

OUX 2

```
-- operateur OU EXCLUSIF
ENTITY oux2 IS
        PORT (e1, e2 : IN BIT;      -- entree
              s : OUT BIT);                 -- sortie
END oux2;

ARCHITECTURE logique OF oux2 IS
BEGIN
        s <= '0' WHEN (e1 = e2) ELSE '1';
END logique;
```

OUX 3

```
-- operateur OU EXCLUSIF
ENTITY oux3 IS
        PORT (e : IN BIT_VECTOR (0 TO 1);
            s : OUT BIT);
END oux3;

ARCHITECTURE abstrait OF oux3 IS
BEGIN
        PROCESS (e)
        BEGIN
                CASE e IS
                        WHEN "00" | "11"  =>                    -- | signifie ou logique
                                s <= '0';
                        WHEN OTHERS =>
                                s <= '1';
                END CASE;
        END PROCESS;
END abstrait;
```

3. Programme VHDL d'un Multiplexeur 2 → 1

1. Description comportementale

```
-- mux 2 -> 1
entity mux2bv is
        port (e0,e1,sel : in bit;       -- entree + adresse
                s : out bit);   -- sortie
end mux2bv;

architecture comporte of mux2bv is
begin
process (e0,e1,sel)                     -- attention : process (sel) engendrerait une erreur car e1,e2 iterviennent
begin
        if sel = '0' then
                s <= e0;
        else
                s <= e1;
        end if;
end process;
end comporte;
```

2. Description flot de données

```
-- mux 2 -> 1
entity mux2df is
        port (e0,e1,sel : in bit;       -- entree + adresse
                s : out bit);   -- sortie
end mux2df;

architecture dataflow of mux2df is
        signal sele0, sele1 : bit;
begin
        s <= sele0 or sele1;                    -- mux comme somme de mintermes
        sele0 <= e0 and not sel;
        sele1 <= e1 and sel;
end dataflow;
```

3. Description structurelle

MUX 1

```
-- mux 2 -> 1
ENTITY mux21 IS
        PORT (e0,e1,sel : IN BIT;       -- entree + adresse
                s : OUT BIT);   -- sortie
END mux21;

ARCHITECTURE pleonasme OF mux21 IS
BEGIN
WITH sel SELECT
        s <= e0 WHEN '0',
                e1 WHEN '1';
END pleonasme;
```

MUX 2

```
-- mux 2 -> 1
entity mux22 is
        port (e0,e1,sel : in bit;       -- entree + adresse
                s : out bit);   -- sortie
end mux22;
architecture pleonasme of mux22 is
begin
        s <= e0 when (sel = '0') else e1;
end pleonasme;
```

3. MUX n > 1

```
-- mux n -> 1 (ici 8 >1)
entity muxn is
	port (e : in bit_vector (0 to 7);	-- entree
		sel : in integer range 0 to 7;	-- adresse
		s : out bit);			-- sortie
end muxn;

architecture vecteur of muxn is
begin
	s <= e(sel);
end vecteur;
```

4. Programme VHDL d'une Bascule D latch

D latch 0

```
-- d-latch
entity dlatch0 is
	port (D,clock : in bit;	-- entree + horloge
		Q : out bit);	-- sortie
end dlatch0;

architecture behav of dlatch0 is		-- description comportementale (if+process)
begin
	process (clock,D)
	begin
		if (clock = '1') then
			Q <= D;
		end if;				-- l'omission du cas ou clock='0' genere le mode memoire
	end process;
end behav;
```

D latch 1

```
-- d-latch bascule ( -> sequentiel)
entity dlatch1 is
	port (D,clock : in bit;	-- entree + horloge
		Q : out bit);	-- sortie
end dlatch1;

architecture dataflow of dlatch1 is		-- description flot de donnees
signal reac: bit;	-- le signal de reaction
begin
	Q <= reac;
	reac <= D when clock = '1'
		else reac;		-- explicite le mode memoire
end dataflow;
```

D latch 2

```
-- d-latch
entity dlatch2 is
	port (D,clock : in bit;	-- entree + horloge
		Q : out bit);	-- sortie
end dlatch2;

architecture behav of dlatch2 is		-- description comportementale (if+process)
signal reac : bit;	-- le signal de reaction
begin
	Q <= reac;
	process (clock,D)
	begin
		if (clock = '1') then
			reac <= D;
		end if;				-- l'omission du cas ou clock='0' genere le mode memoire
	end process;
end behav;
```

D latch init

```
-- d-latch
entity dlatch22 is
	port (D,clock,init : in bit;	-- entree + horloge
		Q : out bit);	-- sortie
end dlatch22;

architecture behav of dlatch22 is		-- description comportementale (if+process)
begin
	process (clock,D,init)
	begin
		if (init = '1') then
			Q <= '1';
		end if;
		if (clock = '1') then
			Q <= D;
		end if;				-- l'omission du cas ou clock='0' genere le mode memoire
	end process;
end behav;
```

D latch component

```
-- d-latch avec instanciation (appel) d'un composant (multiplexeur 2 > 1) (architecture structurelle)
entity mux21 is
	port (a0,a1,sel : in bit;	-- entree
		s : out bit);	-- sortie
end mux21;

architecture pleonasme of mux21 is
begin
	s <= a0 when (sel = '0') else a1;
end pleonasme;

entity dlatch3 is
	port (D,clock : in bit;	-- entree + horloge
		Q : out bit);	-- sortie
end dlatch3;

architecture struct of dlatch3 is		-- description structurelle (= appel de composants)
component mux21				-- declaration du composant utilise
	port (a0,a1,sel : in bit;
		s : out bit);
end component;
signal reac : bit;				-- declaration du signal de reaction (bouclage)
begin
	Q <= reac;
	s1 : mux21 port map (reac,D,clock,reac);	-- instanciation de composant : reac=a0, D=a1, clock=sel, reac=Q
end struct;					-- if sel=0 (clock inactive), Q(=reac)=a0(=reac) : memoire
						-- if sel=1 (clock active), Q(=reac)=a1(=D) : recopie de donnee
```

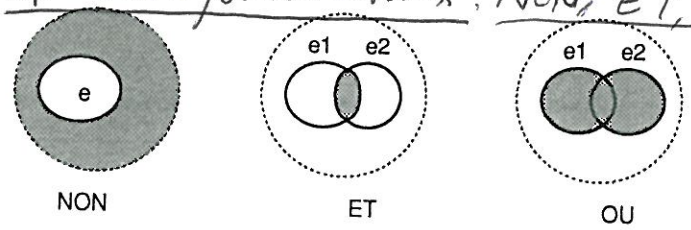


Figure III-5

Description en VHDL

Des tautologies

Les exemples de code source VHDL ci-dessous ne nous apprennent rien sur les propriétés des opérateurs concernés, ils nous montrent l'aspect d'un programme VHDL et nous rappellent que les opérations NON, ET et OU sont définies sur les objets de type BIT comme sur ceux de type BOOLEAN, avec une convention logique positive (1 ≡ TRUE, 0 ≡ FALSE).

```
-- inverseur (ceci est un commentaire)
ENTITY inverseur IS
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END inverseur;
ARCHITECTURE pleonasme OF inverseur IS
BEGIN
s <= NOT e;
END pleonasme;
```

de même :

```
-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
           s : OUT BIT );
END et;
ARCHITECTURE pleonasme OF et IS
BEGIN
s <= e1 AND e2;
END pleonasme;
```

ou encore :


```
-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
           s : OUT BIT );
END ou;
ARCHITECTURE pleonasme OF ou IS
BEGIN
s <= e1 OR e2;
END pleonasme;
```

On notera la structure générale d'un programme et le symbole « d'affectation » particulier aux objets de nature signal (s, e, e1, e2). La déclaration ENTITY correspond au prototype d'une fonction en C, elle décrit l'interaction entre l'opérateur et le monde environnant. La partie ARCHITECTURE du programme correspond à la description interne de l'opérateur, elle décrit donc son fonctionnement. Les mots clés du langage ont été mis en majuscule, c'est une habitude de certains, pas une obligation.

Des affectations conditionnelles

Dans les programmes qui suivent on voit apparaître la notion de « haut niveau » du langage. Des expressions purement booléennes sont utilisées pour décrire le fonctionnement d'un circuit. Ici elles traduisent strictement les tables de vérité, mais permettent évidemment des constructions beaucoup plus élaborées.

```
-- inverseur
ENTITY inverseur IS
PORT ( e : IN  BIT ;
       s : OUT BIT );
END inverseur;
ARCHITECTURE logique OF inverseur IS
BEGIN
s <= '1' WHEN (e = '0') ELSE  '0';
END logique;
```

de même :

```
-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
           s : OUT BIT );
END et;
ARCHITECTURE logique OF et IS
BEGIN
s <= '0' WHEN (e1 = '0' OR e2 = '0') ELSE '1';
END logique;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
             s : OUT BIT );
END ou;
ARCHITECTURE logique OF ou IS
BEGIN
s <= '0' WHEN (e1 = '0' AND e2 = '0') ELSE '1';
END logique;
```

Des exemples de modèles comportementaux

Terminons cette première découverte de VHDL par deux descriptions purement comportementales des opérateurs ET et OU :

```
ENTITY et IS -- operateur ET
PORT ( e1, e2 : IN  BIT ;
             s : OUT BIT );
END et;
ARCHITECTURE abstrait OF et IS
BEGIN
PROCESS ( e1,e2 )
BEGIN
      IF (e1 = '0' OR e2 = '0') THEN
            s <= '0';
      ELSE
            s <= '1';
      END IF;
END PROCESS;
END abstrait;
```

ou encore :

```
-- operateur OU
ENTITY ou IS
PORT ( e : IN  BIT_VECTOR(0 TO 1) ; -- ATTENTION!!!
             s : OUT BIT );
END ou;
ARCHITECTURE abstrait OF ou IS
BEGIN
PROCESS ( e )
BEGIN
      CASE e IS
            WHEN "00" =>
                  s <= '0';
            WHEN OTHERS =>
                  s <= '1';
      END CASE;
```



```
END PROCESS;
END abstrait;
```

III.2.2 Un peu d'algèbre

Nous rappelons rapidement ici quelques propriétés élémentaires des opérateurs fondamentaux de la logique combinatoire. Le lecteur désireux de parfaire sa culture sur ce sujet pourra consulter un ouvrage de mathématiques, au chapitre qui traite de l'algèbre de Boole ou de l'algèbre des parties d'un ensemble[5]. Parmi ces propriétés, les plus importantes, et de loin, dans les applications, sont les lois de De Morgan : ces deux lois permettent de passer d'une convention logique à une autre, sans calcul, ou presque.

Les démonstrations concernant l'algèbre de Boole peuvent toujours se faire, en dernier recours, par un examen des tables de vérité. Cette méthode, un peu lourde, doit être envisagée si des méthodes plus astucieuses ne sont pas trouvées ; en tout état de cause, il n'est pas pensable de rester dans le doute en ce qui concerne un résultat de logique combinatoire. L'intuition permet de gagner du temps dans l'obtention d'un résultat, son absence ne justifie pas le doute.

Propriétés des opérateurs ET et OU

Associativité, commutativité

Associativité :

$$a * (b * c) = (a * b) * c \text{ , de même : } a + (b + c) = (a + b) + c .$$

Commutativité :

$$a * b = b * a \text{ , et : } a + b = b + a .$$

Un opérateur, agissant sur deux opérandes, qui est associatif et commutatif peut être généralisé à un nombre quelconque d'opérandes, sans qu'il soit nécessaire de parenthéser les expressions, par exemple :

$a + b + c + d + e$ est défini de façon univoque quel que soit l'ordre dans lequel on effectue les « calculs ».

Pratiquement cela signifie qu'il est possible de concevoir des opérateurs ET et OU à nombre arbitraire d'entrées (figure III-6) :

[5] Par exemple : J.C. BELLOC et P. SCHILLER : *Mathématiques pour l'électronique*, Masson 1994

sortie d'un opérateur au moyen d'un « fusible » de polarité. L'opérateur OU EXLUSIF permet de créer cette fonctionnalité, l'une de ses entrées est alors considérée comme une entrée de donnée, l'autre comme une commande de polarité, conformément au schéma de principe de la figure III-13.

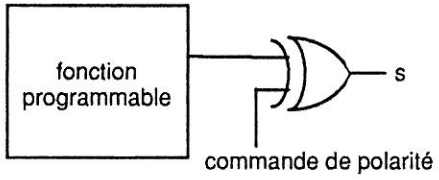


Figure III-13

Descriptions en VHDL

VHDL connaît l'opérateur XOR comme primitive ; les exemples qui suivent sont destinés à explorer, outre les propriétés de cet opérateur, des fonctionnalités du langage que nous n'avions pas abordées jusqu'ici.

Description structurelle

Ayant défini les opérateurs élémentaires ET, OU et NON comme précédemment, il est possible de les utiliser dans une construction plus complexe, comme le OU EXCLUSIF. L'exemple qui suit est, bien sûr, complètement académique, il est difficile d'imaginer plus compliqué pour réaliser un opérateur aussi simple !

```
ENTITY ouex IS  -- operateur OU exlusif
PORT ( a, b : IN  BIT ;
           s : OUT BIT );
END ouex;

use work.portelem.all ; -- rend visible le contenu de
                -- portelem
ARCHITECTURE struct OF ouex IS
signal abar,bbar,abbar,abarb : bit;

BEGIN  -- les differents composants sont instancies ici
i1 : inverseur port map (a,abar);
i2 : inverseur port map (b,bbar);
et1 : et port map (a,bbar,abbar);
et2 : et port map (b,abar,abarb);
ou1 : ou port map (abbar,abarb,s);
```



```
END struct;
```

Pour que le programme précédent soit compris correctement, il a fallu, au préalable, créer et compiler le paquetage portelem et la description des opérateurs élémentaires qui y sont décrits comme suit :

```
package portelem is

component inverseur
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END component;

component et
PORT ( e1, e2 : IN  BIT ;
          s : OUT BIT );
END component;

component ou
PORT ( e1, e2 : IN  BIT ;
          s : OUT BIT );
END component;

end portelem;
-- ce qui suit est la copie de programmes déjà vus
ENTITY inverseur IS
PORT ( e : IN  BIT ; -- les entrees
       s : OUT BIT ); -- les sorties
END inverseur;
ARCHITECTURE pleonasme OF inverseur IS
BEGIN
s <= NOT e;
END pleonasme;

-- operateur ET
ENTITY et IS
PORT ( e1, e2 : IN  BIT ;
          s : OUT BIT );
END et;
ARCHITECTURE pleonasme OF et IS
BEGIN
s <= e1 AND e2;
END pleonasme;

-- operateur OU
ENTITY ou IS
PORT ( e1, e2 : IN  BIT ;
          s : OUT BIT );
     END ou;
ARCHITECTURE pleonasme OF ou IS
```

```
BEGIN
s <= e1 OR e2;
END pleonasme;
```

L'addition élémentaire

L'opérateur OU EXCLUSIF n'est autre que l'opérateur d'addition en base deux, le programme suivant en est la conséquence directe :

OUEX

```
-- operateur OU exlusif

ENTITY ouex IS
PORT ( a, b : IN  INTEGER RANGE 0 TO 1 ;
             s : OUT INTEGER RANGE 0 TO 1 );
END ouex;
ARCHITECTURE arith of ouex is
BEGIN
s <= a + b;
END arith;
```

La comparaison

Si deux opérandes binaires sont différents le résultat de l'opérateur OU EXCLUSIF est '1' :

OUEX

```
-- operateur OU exlusif

ENTITY ouex IS
PORT ( a, b : IN  BIT ;
             s : OUT BIT );
END ouex;
ARCHITECTURE compare of ouex is
BEGIN
s <= '0' WHEN a = b ELSE '1';
END compare;
```

Indicateur de parité impaire

Nous terminerons cette découverte du OU EXCLUSIF par sa généralisation comme contrôleur de parité d'un mot d'entrée :

OUEX

```
-- operateur OU exlusif generalise

ENTITY ouex IS
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
             s : OUT BIT );
END ouex;
```



```
ARCHITECTURE parite of ouex is
BEGIN
process(a)
variable parite : bit := '0';
begin
FOR i in 0 to 3 LOOP
      if a(i) = '1' then
            parite := not parite;
      end if;
END LOOP;
s <= parite;
end process;
END parite;
```

Rien ne s'oppose, semble-t-il, à généraliser ce programme à un mot d'entrée de, mettons, 16 bits. Là se pose un petit problème : l'optimiseur du compilateur va tenter de « réduire » les équations logiques sous-tendues par la boucle « for » pour exprimer la fonction obtenue comme somme (logique) de produits (logiques). Mais il y a 32 768 produits logiques dans un contrôleur de parité sur 16 bits (2^{15}), d'où les dangers des descriptions abstraites....

Une solution plus raisonnable, mais, il est vrai, non optimale du point de vue vitesse de calcul est[8] :

OUEX

```
ENTITY ouex4 IS  -- le même que précédemment
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
             s : OUT BIT );
END ouex4;
ARCHITECTURE parite of ouex4 is
BEGIN
process(a)
variable parite : bit := '0';
begin
FOR i in 0 to 3 LOOP
      if a(i) = '1' then
            parite := not parite;
      end if;
END LOOP;
s <= parite;
end process;
END parite;

ENTITY ouex16 IS
PORT (e : IN BIT_VECTOR(0 TO 15);
      s : OUT BIT_VECTOR (0 TO 3);
-- force la conservation des signaux intermédiaires
```

[8] Le lecteur est instamment convié à dessiner un schéma logi

```
      s16 : OUT BIT);  -- le résultat complet
END ouex16;

ARCHITECTURE struct OF ouex16 IS
SIGNAL inter : BIT_VECTOR(0 TO 3);
COMPONENT ouex4
PORT ( a : IN  BIT_VECTOR(0 TO 3) ;
           s : OUT BIT );
END COMPONENT;
BEGIN
par16 : for i in 0 to 3 generate
g1 : ouex4 port map (e(4*i to 4*i + 3),inter(i));
end generate;
g2 : ouex4 port map (inter,s16);
s <= inter;
END struct;
```

On notera l'intérêt des boucles « generate » pour créer des motifs répétitifs.

III.2.5 Le sélecteur, ou multiplexeur à deux entrées

Le lecteur attentif n'aura pas manqué de remarquer que beaucoup de choses, en logique combinatoire, peuvent s'exprimer par des alternatives SI... ..ALORS... ...AUTREMENT. Mais quel est donc l'opérateur élémentaire qui, en logique câblée, permet de matérialiser directement ce type de propositions ? Le *sélecteur*, ou *multiplexeur*. Nous donnons ci-dessous la description de sa version la plus simple, quand il n'y a que deux choix possibles dans l'alternative, mais il est bien sûr possible de le généraliser pour représenter des choix multiples (IF... ...THEN... ...ELSIF... ...ELSIF... ...END IF, ou, CASE... ...IS WHEN... ...WHEN... ...END CASE).

Description

Principe général

Le sélecteur est construit comme un opérateur où l'on sépare les variables d'entrée en deux groupes :

- Les entrées de *données*, qui sont en général issues d'autres fonctions ;
- L'entrée de sélection, qui est une *commande*.

Prenons un exemple. Pour faire l'addition de deux chiffres décimaux, codés en BCD, il faut commencer par faire l'addition de ces deux chiffres, sans se poser de question, comme s'il s'agissait de nombres écrits en base 2. Deux éventualités peuvent alors se produire :

1. La somme est inférieure à 10, l'opération est alors terminée

2. La somme est supérieure ou égale à 10, ce résultat n'est alors pas correct en BCD. Il faut lui rajouter l'écart entre un nombre binaire sur 4 bits (0 à 15) et un chiffre décimal (0 à 9), soit 6.

Résumons ce qui précède sous forme d'un algorithme :

a et b sont les deux chiffres à additionner, s est le résultat.
s = a + b
si s < 10 terminé
autrement s = s + 6.

Une structure de la réalisation câblée de ce qui précède pourrait être celle de la figure III-14 :

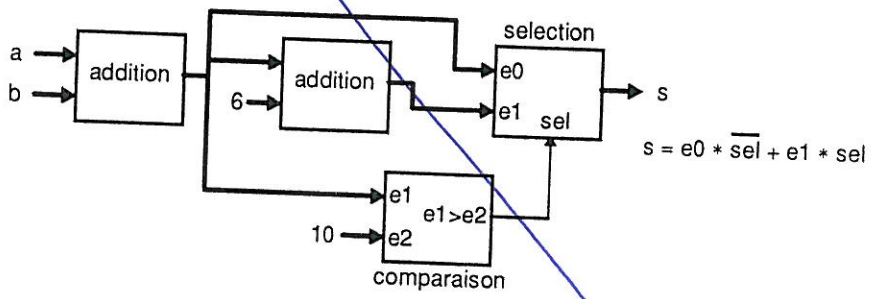


Figure III-14

Symbole et logigramme

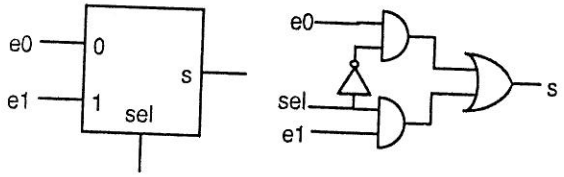


Figure III-15

Le multiplexeur élémentaire est souvent représenté par un symbole qui indique les valeurs de l'entrée de sélection à côté des entrées de données correspondantes (figure III-15).

③ Multiplexeur 2→1 = Description a) comportementale, b) flot de données et c) structurelle

Code source VHDL

Le multiplexeur à deux entrées est l'élément de base des descriptions dans des langages comme VHDL, nous n'en donnerons que quelques exemples :

Quelques tautologies

Nous retrouvons ici la définition même d'un multiplexeur.

③ MUX 2→1

```
entity sel is
port ( e0,e1,sel  : in bit;
                s  : out bit);
end sel;

architecture pleonasme of sel is
begin
with sel select
      s <= e0 when '0',
           e1 when '1';
end pleonasme;
```

ou :

```
entity selecteur is
port ( e0,e1,sel  : in bit;
                s  : out bit);
end selecteur;

architecture procif of selecteur is
begin
process (sel)
begin
if(sel = '0') then
      s <= e0 ;
else
      s <= e1;
end if;
end process;
end procif;
```

ou encore :

```
entity selecteur is
port ( e0,e1,sel  : in bit;
                s  : out bit);
end selecteur;

architecture pleonasme of selecteur is
```

99


```
begin
      s <= e0 when (sel = '0') else e1;
end pleonasme;
```

Une autre façon de voir : les tableaux

VHDL connait les types structurés, la recherche d'un élément d'un tableau se traduit, en logique câblée, par un multiplexeur :

```
entity selecteur is
port ( e     : in bit_vector(0 to 1);
       sel   : in integer range 0 to 1;
       s     : out bit);
end selecteur;

architecture vecteur of selecteur is
begin
      s <= e(sel);
end vecteur;
```

ou, en généralisant :

```
entity selecteur is
port ( e     : in bit_vector(0 to 7);
       sel   : in integer range 0 to 7;
       s     : out bit);
end selecteur;

architecture vecteur of selecteur is
begin
      s <= e(sel);
end vecteur;
```

III.3. Opérateurs séquentiels

Nous avons déjà évoqué l'importance de la notion de *mémoire*, ce qui différencie un opérateur séquentiel d'un opérateur combinatoire réside dans la capacité du premier à « se souvenir » des événements antérieurs : une même combinaison des entrées, à un certain instant, pourra avoir des effets différents suivant les valeurs des combinaisons précédentes de ces mêmes entrées. Pour traduire cet effet de mémoire on introduit la notion d'*état interne* de l'opérateur, l'action des entrées est alors de provoquer d'éventuels changements d'état, la situation qui suit le changement de l'une d'elles dépend des valeurs des entrées et de l'état initial de l'opérateur ; si le nouvel état est différent du précédent on dit qu'il y a eu une *transition*.

Son contenu : une architecture

Le fonctionnement interne d'un module, son *corps*, est précisé par une architecture associée à l'entité qui décrit l'aspect extérieur de ce module. Une architecture porte un nom, ce qui autorise la création de plusieurs architectures différentes pour la même déclaration d'entité. Une unité de conception est la réunion d'une entité et d'une architecture.

Syntaxe

L'architecture est divisée en deux parties : une zone déclarative et une zone d'instructions. Ces instructions sont concurrentes, elles s'exécutent en parallèle. Cela signifie que les instructions d'une architecture peuvent être écrites dans un ordre quelconque, le fonctionnement ne dépend pas de cet ordre d'écriture. Ce point est simple à comprendre si on « pense circuits », les différentes parties d'un circuit coexistent et agissent simultanément. Il peut être un peu surprenant pour les programmeurs habitués des langages procéduraux comme C ou PASCAL, qui continuent à « penser » algorithmes séquentiels.

```
architecture_body ::=
architecture architecture_name of entity_name is
architecture_declarative_part
begin
architecture_statement_part
end [ architecture ] [ architecture_name ] ;
```

La zone déclarative permet de définir des types, des objets (signaux, constantes) locaux au bloc considéré. Elle permet également de déclarer des noms d'objets externes (composants d'une librairie, par exemple) utilisés dans le corps de l'architecture.

Exemples

Les trois exemples qui suivent correspondent aux trois exemples de déclarations d'entités donnés précédemment.

Un multiplexeur de quatre voies vers une voie :

```
architecture essai of mux4_1 is
   constant zero  : bit_vector(1 downto 0) := "00" ;
   constant un    : bit_vector(1 downto 0) := "01" ;
   constant deux  : bit_vector(1 downto 0) := "10" ;
   constant trois : bit_vector(1 downto 0) := "11" ;
begin
process (e0,e1,e2,e3,sel)
begin
      case sel is
         when zero  => sort <= e0 ;
         when un    => sort <= e1 ;
         when deux  => sort <= e2 ;
         when trois => sort <= e3 ;
      end case ;
```



```
end process ;
end essai ;
```

Notons en passant que <= représente l'opérateur d'affectation d'un signal.

Un multiplexeur de dimension arbitraire :

```
architecture essai of muxN_1 is
begin
sort <= entree(sel) ;
end essai ;
```

Un registre bidirectionnel de dimension arbitraire :

```
architecture essai of registre_N is
   signal temp : std_logic_vector(dimension - 1 downto 0) ;
begin
donnee <= temp when direction = '0' else (others => 'Z') ;[7]
process
begin
      wait until hor = '1' ;
      if direction = '1' then
         temp <= donnee ;
      end if ;
end process ;
end essai ;
```

On distingue classiquement en VHDL trois styles de descriptions, qui peuvent être utilisés simultanément.

a) *Description comportementale*

Une description comportementale (*behavioral*) se présente comme des blocs d'algorithmes séquentiels exécutés par des processus indépendants. Elle peut traduire un fonctionnement séquentiel ou combinatoire du circuit modélisé. Nous expliciterons ce point en détail dans la suite.

Un simple multiplexeur deux voies vers une voie peut être décrit par un algorithme qui reproduit son comportement :

```
entity mux2_1 is
   port(e0,e1 : in bit ;
        sel : in bit ;
        sort : out bit ) ;
end mux2_1 ;

architecture comporte of mux2_1 is
begin
process (e0,e1,sel)[8]
```

7. L'expression (others => 'Z') est un aggregat. La même valeur 'Z' est affectée à tous les éléments du tableau.
8. Les éléments mis entre parenthèse indiquent les signaux dont les changements doivent « réveiller » le processus, et donc provoquer l'évaluation du signal de sortie.

```
begin
   if sel = '0' then
      sort <= e0 ;
   else
      sort <= e1 ;
   end if ;
end process ;
end comporte ;
```

b) Description flot de données

Une description flot de données (*data flow*) correspond grosso modo à un *register transfert language*, les signaux passent à travers des couches d'opérateurs logiques qui décrivent les étapes successives qui font passer des entrées d'un module à sa sortie. Le même multiplexeur élémentaire s'écrit, dans ce style :

```
architecture flot of mux2_1 is
      signal sele0, sele1 : bit ;
begin
      sort  <= sele0 or sele1 ;
      sele0 <= e0 and not sel ;
      sele1 <= e1 and sel ;
end flot ;
```

c) Description structurelle

Une description structurelle (*structural*) utilise des composants supposés exister dans une librairie de travail, sous forme d'unités de conception. Le programme se contente alors d'*instancier* les composants nécessaires et de décrire leurs interconnexions. Le même multiplexeur utilise deux portes ET, un NON et un OU[9] :

```
architecture struct of mux2_1 is
component et
      port (a, b : in bit ; s : out bit) ;
end component ;
component ou
      port (a, b : in bit ; s : out bit) ;
end component ;
component non
      port (a : in bit ; s : out bit) ;
end component ;
signal nonsel, sele0, sele1 : bit ;
begin
result  : ou  port map(sele0, sele1, sort) ;
```

9. Le lecteur est vivement convié à dessiner le schéma classique d'un tel multiplexeur.


```
complem : non port map(sel, nonsel) ;
choix0  : et  port map(nonsel, e0, sele0) ;
choix1  : et  port map(sel, e1, sele1) ;
end struct ;
```

Ce dernier programme suppose que les composants portent les mêmes noms que les unités de conception auxquelles ils se réfèrent, ce n'est en rien une obligation. Des déclarations de configuration permettent de créer des liens, entre les composants instanciés et les couples entité architecture, autres que par homonymie.

Les exemples qui précèdent sont d'une naïveté qui n'utilise pas la puissance du langage, il ne resterait rien de ces programmes si on cherchait à rendre le code source plus compact, c'est une évidence.

Le plus souvent, on utilisera une description structurelle au niveau supérieur d'un projet, et des descriptions des deux autres types, suivant les fonctions décrites et les goûts du programmeur, pour les modules instanciés. Les programmes VHDL générés par les outils de placement routage, à des fins de vérification temporelle du bon fonctionnement d'une application, sont bien sûr essentiellement structurels : ils reproduisent le câblage réellement effectué dans le circuit ou sur une carte. Les fondeurs fournissent des modèles comportementaux des opérateurs élémentaires de leurs circuits qui prennent en compte leurs caractéristiques dynamiques, temps de propagation, entre autres.

II.2.3. Types et classes

VHDL est un langage fortement typé, tout objet[10] manipulé doit avoir un type défini avant la création de l'objet. Indépendamment de son type, un objet appartient à une classe[11]. Schématiquement, on peut dire que le type définit le format des données et l'ensemble des opérations légales sur ces données, alors que la classe définit un comportement dynamique, précise la façon dont évolue (ou n'évolue pas dans le cas d'une constante !) une donnée au cours du temps.

Le langage distingue quatre catégories de types :

- Les types scalaires, c'est-à-dire les types numériques et énumérés, qui n'ont pas de structure interne.
- Les types composés (tableaux et enregistrement) qui possèdent des sous-éléments.
- Les types *access*, qui sont des pointeurs.
- Le type *file*, qui permet de gérer des fichiers séquentiels.

10. Dans le langage un objet est défini comme une grandeur nommée qui contient une valeur d'un type défini. Une *entity*, par exemple, n'est donc pas un objet. Le fait qu'un objet doive être nommé souffre quelques exceptions.

11. En réalité cette indépendance entre classe et type n'est pas vérifiée pour les fichiers : la classe *file* est associée à des types qui lui sont spécifiques. La manipulation des fichiers sera abordée plus loin dans ce chapitre, à propos des outils de modélisation.

④. Bascule D latch

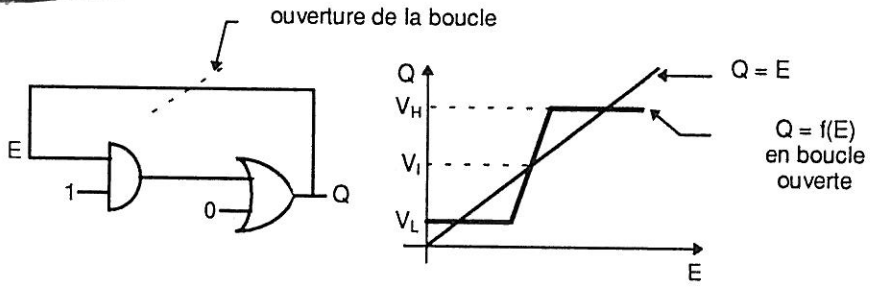


Figure III-17

Le système d'équations associé à cette construction graphique a trois solutions :

- $Q = V_L$ et $Q = V_H$, qui correspondent à deux états logiques possibles, sont des solutions stables. Si l' entrée D de la bascule place celle ci dans l'un de ces deux états, quand L = '1', le circuit conservera son état dans le mode mémoire (L = '0'), quelle que soit la valeur de D.
- $Q = V_I$ est une solution instable qui correspond à un état analogique intermédiaire. Si la bascule se trouve accidentellement dans cet état, elle évoluera vers l'un ou l'autre des états stables[10].

Quelques descriptions en VHDL

VHDL ne connait pas la fonction mémoire comme élément primitif. Une bascule asynchrone est générée soit par une description structurelle, soit par une description comportementale exhaustive, c'est à dire qui comprend la description explicite du mode mémoire, soit, *et cela constitue, pour les débutants, un piège du langage*, par une description incomplète des alternatives d'une instruction « IF ».

```
entity selecteur is  -- déjà vu précédemment
port (      a0,a1,sel : in bit;
      s     :   out bit);
end selecteur;

architecture pleonasme of selecteur is
begin
      s <= a0 when (sel = '0') else a1;
end pleonasme;

entity d_latch is
port ( D,L : in bit;
```

69

[10] Voir à ce sujet au paragraphe II.3 la présentation des états métastables dans les circuits synchrones, l'existence de ces états est due à cette troisième solution $Q = V_I$ dans les bascules asynchrones qui servent à réaliser une bascule synchrone.


```
      Q : out bit);
end d_latch;

architecture struct of d_latch is
 -- description structurelle
component selecteur
port ( a0,a1,sel : in bit;
       s : out bit);
end component;
signal reac : bit;
begin
Q <= reac;
s1 : selecteur port map(reac,D,L,reac);
end struct;
```

Le code qui précède n'est que la traduction naïve du premier schéma de la figure III-16. Les architectures qui suivent, qui décrivent la même entité, sont plus synthétiques :

```
architecture d_flow of d_latch is
signal reac : bit;  --  le signal de réaction
begin
Q <= reac;
reac <= D when L = '1'
        else reac; -- explicite le mode mémoire.
end d_flow;
```

Donnons enfin une forme de description qui génère le mode mémoire par omission d'une combinaison dans une alternative « IF ». La possibilité de ce type de construction présente le danger qu'elle est parfois le résultat d'un réel oubli du programmeur, et non d'une volonté de sa part :

```
architecture behav of d_latch is
signal reac : bit;
begin
Q <= reac;
process(L,D)
begin
if(L = '1') then
     reac <= D ;
end if; -- L'omission du cas où L = '0'
        -- génère le mode mémoire.
end process;
end behav;
```

5. Bascule RS

La bascule RS met bien en évidence la difficulté majeure des systèmes asynchrones : si les deux commandes passent *simultanément* de l'état actif à l'état inactif (mémoire), le *résultat est imprévisible*, c'est ce qu'on appelle classiquement un *aléa*. Un phénomène analogue existe évidemment dans les bascules D-Latch, mais choque moins en raison de la dissymétrie fonctionnelle des deux entrées D et L.

Quelques descriptions en VHDL

```
entity rs is port (
R,S : in bit;
Q   : out bit);
end rs;

architecture df of rs is
signal etat : bit;
begin
q <= etat;
with R&S select
-- l'opérateur & concatène les signaux R et S
etat <=     '1' when "01",
            '0' when "10",
            etat when "00", -- mode mémoire explicite
            '0' when "11";
end df;
```

Ayant utilisé, pour décrire une bascule D-Latch, une instruction « IF » incomplète, nous donnerons ci-dessous l'exemple d'une instruction « CASE » pour générer le mode mémoire :

```
architecture bv of rs is
begin
process (R,S)
begin
   case R&S is
      when "01"         => Q <= '1';
      when "10" | "11" => Q <= '0'; -- '|' est un
                                    -- ou logique
      when others null ; -- mode mémoire : pas
                         -- d'action sur Q.
   end case;
end process;
end bv;
```

Noter que, si toutes les combinaisons de l'expression testée ne sont pas mentionnées explicitement, l'alternative « others » est obligatoire ; l'instruction « CASE » ne peut pas être incomplète. L'instruction « null » traduit l'absence

d'action, par défaut la valeur de Q est conservée, ce qui correspond bien à une cellule mémoire.

Les applications

La première application des bascules R-S réside dans les commandes de « forçage » à un ou à zéro des systèmes séquentiels, des plus simples aux plus complexes, à la mise sous tension notamment. Beaucoup de circuits offrent, à cet effet, un mode de fonctionnement de type « RS » en plus du fonctionnement normal, synchrone dans la plupart des cas. La réinitialisation matérielle (hard reset) d'un ordinateur, par exemple, correspond à une réinitialisation asynchrone (i.e. indépendante de l'horloge) de certains registres critiques du processeur, charge au programmeur d'avoir prévu la suite des événements.

Une autre application classique des bascules RS est l'élimination des rebonds des interrupteurs : Quand un interrupteur passe d'un état à un autre (ouvert ou fermé), il se produit généralement un régime transitoire oscillatoire où ces deux états se succèdent avant que l'un d'eux soit stable. Une solution classique consiste à remplacer les interrupteurs par des commutateurs, objets à deux états mécaniquement stables, F1 et F2, et un état mécaniquement instable où les deux contacts sont ouverts, deux résistances et une bascule RS, conformément au schéma de la figure III-20, dont nous laisserons l'étude détaillée à la sagacité du lecteur[12].

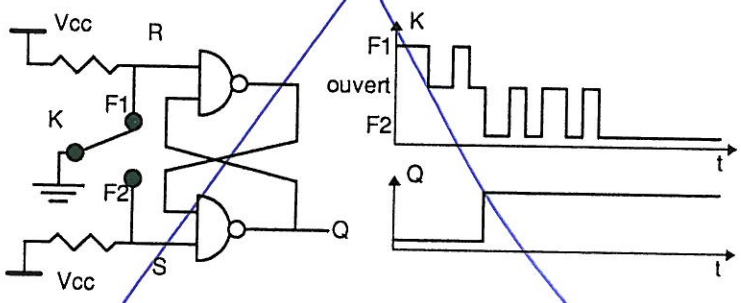


Figure III-20

[12] Guide de raisonnement : quand un commutateur passe de F1 à F2 il oscille entre F1 ou F2 et l'état intermédiaire où les deux contacts sont ouverts.

6. Bascule D

Sur ces chronogrammes on a souligné par une zone grise les moments où commandes et état d'une bascule ne sont pas forcément bien déterminés, mais il s'agit là d'une simple illustration. En aucun cas le contenu de ces zones n'est nécessaire à la compréhension du principe de fonctionnement.

Diagrammes de transition

Pour représenter de façon visuelle le fonctionnement des bascules synchrones, tout en mettant en évidence les notions centrales de la logique séquentielle que sont les états et les transitions, on utilise souvent des *diagrammes de transitions entre états* (*state transition diagram*), ou, pour abréger, diagrammes de transitions ou diagrammes d'états (figure III-23).

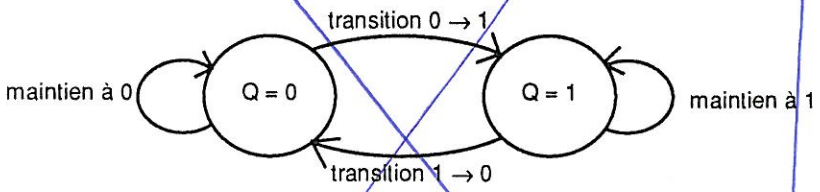


Figure III-23

Dans un tel diagramme un cercle représente un état (une bascule en a deux), une flèche une transition entre deux états (qui peut être un maintien dans l'état initial). Pour qu'une transition soit effectuée *trois* conditions doivent être vérifiées :

1. La bascule doit être dans l'état de départ,
2. il doit y avoir un front actif du signal d'horloge,
3. les entrées de commande autre que l'horloge doivent autoriser la transition.

En général le signal d'horloge est implicite, mais il ne faut bien sûr pas oublier cette condition sine qua non. La description (analyse) ou la création (synthèse) d'une bascule revient donc à préciser les équations logiques (quatre au maximum pour une bascule) qui définissent les transitions en fonction des commandes.

Une précision qui concerne VHDL

« VHDL ne contient pas le concept de signal d'horloge. Le moyen d'introduire un signal d'horloge dans vos ouvrages (designs) est d'utiliser une instruction "WAIT" dans un processus, ou d'utiliser une description structurelle »[16] !

Cela a le mérite d'être clair, il vaut mieux être prévenu.

[16] Wait VHDL

Pour illustrer ce qui précède on donne ci-dessous la structure d'un programme qui décrit une bascule :

```
entity basc_synchrone is port (
clock : in bit;
commande : in bit_vector( ... );
q: out bit);
end basc_synchrone;

architecture fsm of basc_synchrone is
signal etat : bit;
begin
q <= etat;
process
begin
wait until (clock = '1') -- tout est là
case etat is
      when '0' =>
  -- conditions de la transition '0'->'1'
      when '1' =>
  -- conditions de la transition '1'->'0'
end case;
end process;
end fsm;
```

D'autres constructions équivalentes existent, qui utilisent une liste de « sensibilité » dans la description du processus, nous aurons l'occasion de les examiner dans la suite. Le point important à noter est que *seuls les processus* permettent de générer à partir d'une description comportementale la synthèse d'une fonction qui utilise des bascules synchrones.

L'élément fondateur : la bascule D

Le principe

La bascule D synchrone, plus laconiquement D-edge, est la cellule mémoire fondamentale. Munie d'une entrée de donnée (en général notée « D »), et, naturellement, d'une entrée d'horloge, elle prend, à chaque transition active de l'horloge, l'état dont la valeur est celle de l'entrée de donnée. L'équation générale des bascules synchrones devient, dans ce cas, extrêmement simple :

$$Q(t) = D(t-1) \quad \text{où t est la période d'horloge considérée.}$$

En notation abrégée, mais trompeuse car les deux membres de l'équation *ne sont pas* pris au même instant, on écrit parfois cette équation :

$$Q = D$$

Le symbole, le diagramme de transition et un exemple de chronogramme qui illustre le fonctionnement sont indiqués sur la figure III-24 :

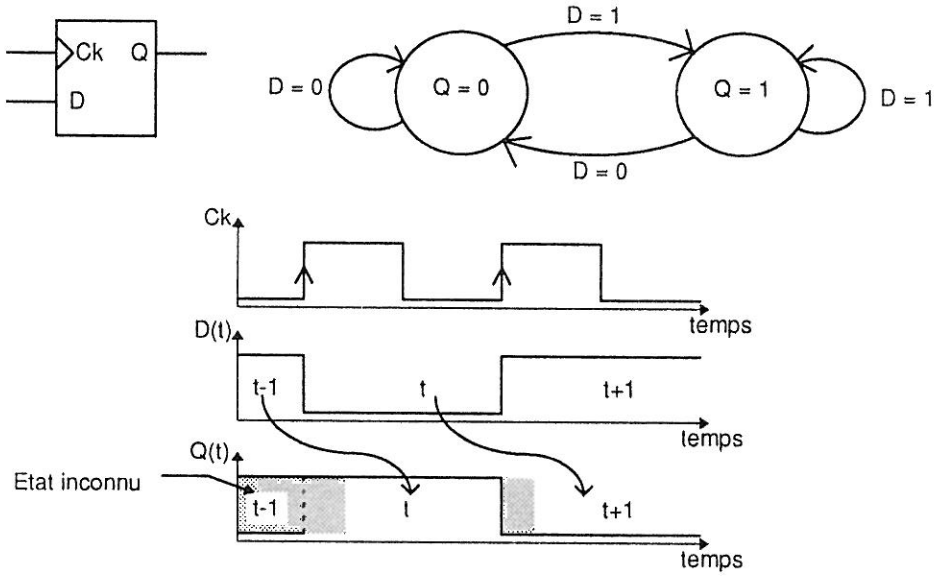


Figure III-24

On notera, dans le diagramme de transitions, le lien qui est indiqué entre les changements (ou le maintien) d'état et l'entrée de commande D. Dans la figure précédente on a pris la précaution de noter qu'à priori l'état initial de la bascule est inconnu. Ce point est important à garder en mémoire quand on se pose un problème de synthèse de système séquentiel.

Un exemple de réalisation

La réalisation interne d'une bascule D-edge *n'est en général pas* le souci du concepteur d'un ensemble logique, la bascule en question est un opérateur primitif, au même titre qu'une porte ET. Le schéma ci dessous, figure III-25, est donné à titre d'information.

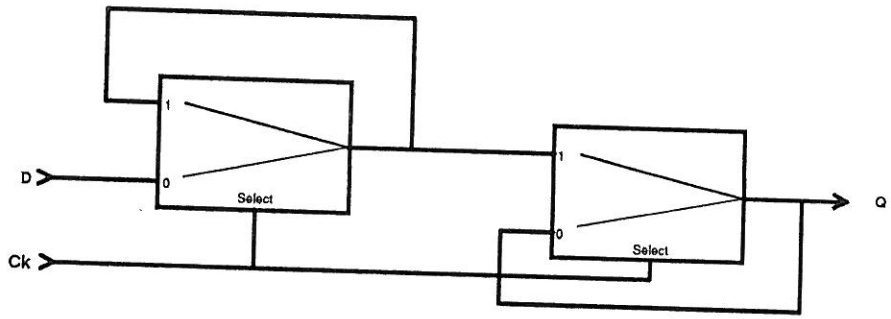


Figure III-25

Les deux multiplexeurs sont connectés en bascules D-Latch, avec des niveaux actifs inversés pour le mode mémoire. Quand l'horloge est au niveau bas la cellule de sortie est en mode mémoire, donc insensible aux variations éventuelles de son entrée, la cellule d'entrée en mode transparent. Quand l'horloge passe au niveau haut la cellule d'entrée mémorise la donnée présente, et la transfère dans la cellule de sortie. Le bon fonctionnement de l'ensemble est en fait assuré par l'existence de temps de commutation non nuls des multiplexeurs.

Cette technique de réalisation d'une bascule D, différente de celle utilisée pour la 74xx74 des familles TTL, est employée, par exemple, dans les circuits programmables (FPGAs) TPC12 (Texas Instrument).

Description en VHDL

Les deux exemples qui suivent, quoique des plus simples, sont à méditer attentivement. Ils représentent *les deux seules façons sûres* d'obtenir d'un compilateur VHDL la génération d'une bascule D synchrone générique (i.e. qui ne soit pas reconstruite au moyen de portes, ou, pire, qui ne soit pas une bascule D-Latch).

```
entity d_edge is
port ( d,hor : in bit;
       s : out bit);
end d_edge;

architecture d_primitive of d_edge is
begin
process
begin
wait until hor = '1';
s <= d ;
end process;
end d_primitive;
```

Et une variante qui remplace l'instruction « WAIT » par une liste de sensibilité du processus et un test sur *l'existence d'une transition* du signal d'horloge *et le niveau qui suit* cette transition.

```
architecture d_primitive1 of d_edge is
begin
process(hor) -- Le process ne « réagit » qu'au signal hor.
begin
if(hor'event and hor = '1') then
-- attention ! deux conditions
      s <= d ;
end if;
end process;
end d_primitive1;
```

L'omission du facteur « hor'event » dans le test conduit certains compilateurs à générer une bascule D-Latch.

La deuxième des deux formes présentées ci-dessus est un peu plus compliquée que la première, mais plus souple. Elle permet en effet d'inclure une commande d'initialisation asynchrone, reset dans l'exemple qui suit, à une bascule D synchrone. La méthode utilisée dans cet exemple présente cependant un certain danger, les compilateurs sont toujours accompagnés d'optimiseurs qui modifient éventuellement les polarités des signaux internes, en utilisant les lois de De Morgan. L'utilisateur peut alors avoir la désagréable surprise de découvrir qu'une remise à zéro asynchrone se traduit parfois par une mise à un de la sortie attachée à la bascule visée !

```
entity d_edge is
port ( d,hor,reset : in bit;
       s : out bit);
end d_edge;

architecture d_primitive of d_edge is
begin
process(hor,reset)
begin
if(reset = '1') then
      s <= '0';
elsif(hor'event and hor = '1') then
      s <= d ;
end if;
end process;
end d_primitive;
```


Terminons ce tour d'horizon des descriptions en VHDL d'une bascule D par la traduction dans ce langage du schéma construit au moyen de multiplexeurs :

```
entity d_edge is
port ( d,hor : in bit;
       s : out bit);
end d_edge;

architecture d_flow of d_edge is
signal sort,entre : bit;
begin
s <= sort;
entre <=  d when hor = '0' else entre;
sort  <=  entre when hor = '1' else sort;
end d_flow;
```


A n'utiliser qu'en dernier recours, quand on a épuisé toutes les « vraies » bascules D disponibles dans un circuit.

Les applications

Les bascules D sont la clé de voûte de toutes les applications séquentielles. Des quelques bascules (4 ou 8) couramment rencontrées dans les circuits standard de la famille TTL, on passe à plus de mille dans les « gros » circuits programmables.

Focalisée sur les transitions : la bascule T

Le principe

L'une des difficultés d'emploi des bascules D dans certaines applications réside dans le fait que la condition de maintien à '1' de son diagramme de transition ne doit pas être omise dans les équations obtenues pour la commande D, ce qui complique parfois notablement ces équations.

La bascule T (T pour Toggle, c'est à dire bascule) est un élément qui interprète son unique entrée de commande (en plus de l'horloge, évidemment), T, non comme une donnée à mémoriser, mais comme un ordre de changement d'état :

⇒ Si T = "actif" changer d'état à la prochaine transition de l'horloge,

⇒ si non conserver l'état initial.

D'où l'équation qui décrit son fonctionnement :

$$Q(t) = T * \overline{Q(t-1)} + \overline{T} * Q(t-1)$$

On peut éclairer l'équation précédente par le diagramme de transitions de la figure III-26, ci-dessous :

7. Bascule T

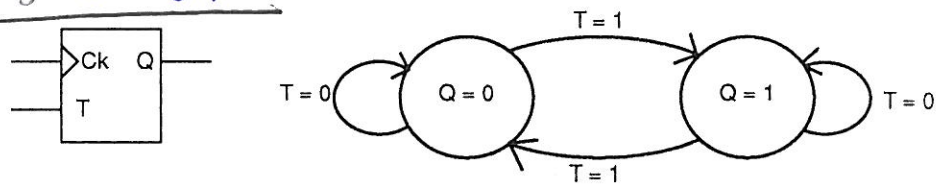


Figure III-26

Un exemple de réalisation

L'examen du diagramme de transitions de la figure III-26 nous montre que Q(t) = 1 si

Q(t-1) = '1' et T = '0',

ou

Q(t-1) = '0' et T = '1'

Ce qui nous fournit l'équation de l'entrée D d'une bascule D :

$$D = T \oplus Q$$

D'où le logigramme :

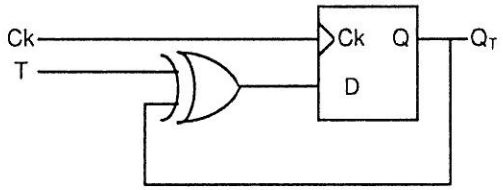


Figure III-27

Description en VHDL

La description d'une bascule T se déduit simplement du diagramme de transition :


```
entity T_edge is
port ( T,hor: in bit;
       s : out bit);
end T_edge;

architecture d_primitive of T_edge is
signal etat : bit;
begin
s <= etat ;
process
begin
wait until hor = '1' ;
if(T = '1') then
      etat <= not etat;
end if;
end process;
end d_primitive;
```

On rappelle que de tels exemples sont fournis à titre d'illustration du fonctionnement de l'opérateur considéré, et pour familiariser le lecteur avec le langage VHDL. On n'a jamais besoin, en pratique, de décrire chaque bascule utilisée dans ce langage !

Les applications

L'un des intérêts principaux des bascules de type T est qu'elles permettent de générer de façon extrêmement simple des compteurs binaires synchrones. Un compteur binaire est une fonction séquentielle synchrone dont l'état interne est un nombre entier naturel codé en binaire dont chaque chiffre binaire est matérialisé par une bascule. A chaque transition active de l'horloge ce nombre est incrémenté de 1, quand le nombre maximum est atteint, toutes les bascules sont à 1, la séquence recommence à partir de 0. Si le nombre de bits utilisés est n, on parlera d'un compteur modulo 2^n. La simple observation d'une table des entiers naturels écrits en base 2 nous fournit la clé du problème : la bascule de rang i doit changer d'état quand toutes les bascules de rang inférieur sont à 1.

D'où un exemple de réalisation d'un compteur synchrone modulo 16 dont on peut interrompre le comptage (en = '0') :

```
ENTITY cnt16 IS
PORT (ck, en : IN BIT;
      s : OUT BIT_VECTOR (0 TO 3)
      );
END cnt16;

ARCHITECTURE structurelle OF cnt16 IS
SIGNAL etat : BIT_VECTOR(0 TO 3);
SIGNAL inter: BIT_VECTOR(1 TO 3);
COMPONENT T_edge
```

```
-- la même que dans l'exemple précédent
port ( T,hor: in bit;
       s : out bit);
END COMPONENT;

BEGIN
-- Etablir le logigramme tout en lisant le texte
s <= etat ;
inter(1) <= etat(0) and en ;
inter(2) <= etat(1) and inter(1) ;
inter(3) <= etat(2) and inter(2) ;
g0 : T_edge port map (en,ck, etat(0));
g1 : for i in 1 to 3 generate
       g2 : T_edge port map (inter(i),ck,etat(i));
end generate;
END structurelle;
```

Là encore mettons en garde le lecteur, quand on a réellement besoin d'un compteur on écrit

```
etat <= etat + 1 ;
```

c'est nettement plus simple, le compilateur générera de lui même les interconnexions nécessaires entre les bascules.

L'ancêtre vénérable : la bascules J-K

Le principe

Quelque peu tombée en désuétude, la bascule J-K a régné en maître dans le monde de la logique séquentielle des décennies 60 et 70. Elle est l' héritière directe de la bascule R-S, que l'on a débarassé progressivement de ses difficultés asynchrones. Les premières bascules J-K n'étaient, en fait, pas de réels opérateurs synchrones, elles comportaient tout un mécanisme de mémorisation interne des commandes dans des bascules R-S (maître-esclave). Les versions actuelles sont construites au moyen d'une bascule D-edge et de logique combinatoire.

Une bascule J-K dispose de deux commandes, J et K, outre l'horloge. Son fonctionnement se décrit bien au moyen d'une table qui décrit la fonction réalisée en fonction des valeurs de la commande :

8 Bascule JK

J(t-1)	K(t-1)	Fonction	Equation
0	0	Mémoire	Q(t) = Q(t-1)
0	1	Mise à zéro synchrone	Q(t) = '0'
1	0	Mise à un synchrone	Q(t) = '1'
1	1	Changement d'état	$Q(t) = \overline{Q(t-1)}$

Comme pour tout opérateur synchrone, l'état de la bascule ne change pas entre deux transitions actives du signal d'horloge. La fonction « mémoire », dans la table ci-dessus, signifie que la bascule conserve son état précédent même lors d'une transition d'horloge. Dans la construction du diagramme de transitions de la figure suivante, III-28, on a tenu compte de ce que chaque transition peut être obtenue par deux combinaisons différentes des commandes J et K, la transition '0' → '1', par exemple, peut être obtenue par mise à 1 explicite (JK = "10") ou par changement d'état (JK = "11") :

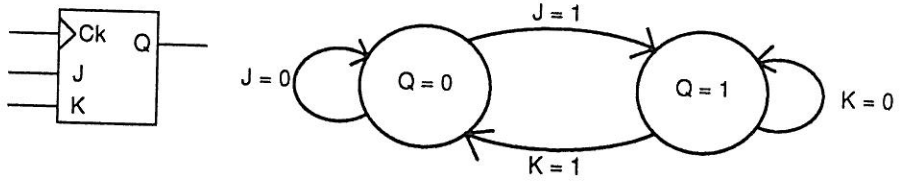


Figure III-28

On déduit aisément l'équation de la bascule J-K de son diagramme de transition :

$$Q(t) = J * \overline{Q(t-1)} + \overline{K} * Q(t-1)$$

Au lieu de raisonner sur l'équation de l'état futur, on peut décrire la bascule J-K par son équation de transition, si on introduit une variable binaire auxilliaire T_{JK}, égale à '1' si une transition doit avoir lieu, '0' autrement, on obtient :

$$T_{JK} = J * \overline{Q} + K * Q$$

Description en VHDL

La première description que nous donnerons est la simple traduction naïve de la table de vérité :

```
entity jk is port (
```

```
j,k,clock : in bit;
q: out bit);
end jk;

architecture fsm of jk is
signal etat : bit;
begin
process begin
wait until clock = '1';
      if (j = '1' and k = '1') then
            etat <= not etat;
      elsif (j = '1' and k = '0') then
            etat <= '1';
      elsif (j = '0' and k = '1') then
            etat <= '0';
      end if;
end process;
q <= etat;
end fsm;
```

Dans la version suivante, construite de la même façon, on a tenu compte des simplifications qui apparaissent dans le diagramme de transitions. Il est bien évident que le compilateur aurait, de toute façon trouvé tout seul ces simplifications.

```
architecture fsm1 of jk is
signal etat : bit;

begin
process begin
wait until clock = '1';
      IF (j = '1' and etat = '0') then
            etat <= '1';
      elsif (k = '1' and etat = '1') then
            etat <= '0';
      end if;
end process;
q <= etat;
end fsm1;
```

Les exemples précédents étaient construits à partir de la commande, ceux qui suivent le sont à partir de l'état interne de la bascule :

```
architecture fsm2 of jk is
signal etat : bit;
begin
q <= etat;
process begin
wait until clock = '1';
```



```
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end process;
end fsm2;
```

Ou, dans une variante déjà rencontrée :

```
architecture fsm3 of jk is
signal etat : bit;
begin
q <= etat;
process(clock)
begin
if(clock = '1'and clock'event) then
case etat is
      when '0' =>
            IF (j = '1' ) then
                  etat <= '1';
            end if;
      when '1' =>
            if (k = '1' ) then
                  etat <= '0';
            end if;
end case;
end if;
end process;
end fsm3;
```

En conclusion de cette énumération précisons que les équations logiques générées par un compilateur seront les mêmes quelle que soit la forme du programme source, à savoir :

PLD Compiler Software DESIGN EQUATIONS

$$q.D = q.Q * /k + /q.Q * j$$

Ce qui est rassurant.

TP 5 CORRIGE. VHDL

[Réf. Livre VHDL (Meaudre & all …)]

1. Programme VHDL d'une Bascule RS (asynchrone)

RS 1

```
-- rs bascule
entity rs1 is
        port (R,S : in bit;       -- entree
              Q : out bit);   -- sortie
end rs1;

architecture behav of rs1 is                    -- comportemental
signal etat : bit;
begin
q <= etat;
process (R,S)
begin
        if R='0' and S='0' then
                etat <= etat;                   -- mode memoire explicite (memo implicite possible aussi en omettant ce cas)
        elsif R='0' and S='1' then
                etat <= '1';                    -- set
        elsif R='1' and S='0' then
                etat <= '0';                    -- reset
        elsif R='1' and S='1' then
                etat <= '0';                    -- indesirable
        end if;
end process;
end behav;
```

RS 2

```
-- rs bascule
entity rs2 is
        port (R,S : in bit;       -- entree
              Q : out bit);   -- sortie
end rs2;

architecture behav of rs2 is                    -- comportemental
signal etat : bit;
begin
q <= etat;
process (R,S)
begin                                                   -- mode memoire R='0' and S='0' implicite (memorisation)
        if R='0' and S='1' then
                etat <= '1';                    -- set
        elsif R='1' and S='0' then
                etat <= '0';                    -- reset
        elsif R='1' and S='1' then
                etat <= '0';                    -- indesirable
        end if;
end process;
end behav;
```

2. Programme VHDL d'une Bascule D (>0 edge triggered)

D edge 1

```
-- dedge                bascule D +edge (appelee communement bascule D)
entity dedge1 is
        port (D,clock : in bit;       -- entree + horloge
              Q : out bit);   -- sortie
end dedge1;

architecture primitive of dedge1 is
begin
        process
        begin
                wait until clock = '1';              -- front montant
                Q <= D;
        end process;
end primitive;
```

D edge 2

```
-- dedge                bascule D +edge (appelee communement bascule D)
entity dedge2 is
        port (D,clock : in bit;       -- entree + horloge
              Q : out bit);   -- sortie
end dedge2;

architecture primitive of dedge2 is
begin
        process(clock)          -- le process ne reagit qu'au signal d'horloge
        begin
                if (clock 'event and clock ='1') then                   -- front montant
                        Q <= D;
                end if;
        end process;
end primitive;
```

3. Programme VHDL d'une Bascule T (>0 edge triggered)

La bascule T considérée a, en plus de l'entree d'horloge (clock), une entrée de validation T.
La sortie Q de la bascule T passe de 0 à 1 pour T=1 et clock active.
La sortie Q de la bascule T reste à 0 pour T=0 et clock active.
La sortie Q de la bascule T passe de 1 à 0 pour T=1 et clock active.
La sortie Q de la bascule T reste à 1 pour T=0 et clock active.
Si clock est inactive, la sortie Q de la bascule T reste à l'état antérieur.

```
-- Bascule T edge  (+edge triggered (front montant))
ENTITY tedge IS
        PORT (T, clock : IN BIT;       -- entree
              s : OUT BIT);     -- sortie
END tedge;

ARCHITECTURE primitive OF tedge IS
SIGNAL etat : BIT;
BEGIN
        s <= etat;
        PROCESS
        BEGIN
                WAIT UNTIL clock = '1';                 -- front montant d'horloge
                        IF (T = '1') THEN
                                etat <= not etat;
                        END IF;
        END PROCESS;
END primitive;
```

4. Programme VHDL d'une Bascule JK (>0 edge triggered)

JK Table de vérité

```
-- Bascule JK d'apres la table de verite (+edge triggered (front montant))
entity jk1 is
        port (j,k,clock : in bit;       -- entree
              q : out bit);     -- sortie
end jk1;

architecture primitive of jk1 is
signal etat : bit;
begin
process
        begin
        wait until clock = '1';             -- front montant
                if (j='1' and k='1') then
                                    etat <= not etat;
                elsif (j='1' and k='0') then
                                    etat <= '1';
                elsif (j='0' and k='1') then
                                    etat <= '0';
                -- le cas j='0' et k='0' n'est pas renseigne pour sequentiel (memorisation)
                end if;
        end process;
q <= etat;
end primitive;
```

JK Automate

```
-- Bascule JK d'apres l'automate  (+edge triggered (front montant))
entity jk2 is
    port (j,k,clock : in bit;       -- entree
          q : out bit);     -- sortie
end jk2;

architecture primitive of jk2 is
signal etat : bit;
begin
process
    begin
    wait until clock = '1'; -- front montant
      if (j='1' and etat='0') then
            etat <= '1';
      elsif (k='1' and etat='1') then
            etat <= '0';
      -- les autres cas de figure d'entrees ne sont pas renseignes pour sequentiel (memorisation)
      end if;
    end process;
q <= etat;
end primitive;
```

JK Etat interne

```
-- Bascule JK a partir de l'etat interne de la bascule et non plus des commandes j et k  (+edge triggered)
entity jk3 is
      port (j,k,clock : in bit;        -- entree
                q : out bit);      -- sortie
end jk3;

architecture primitive of jk3 is
signal etat : bit;
begin
q <= etat;
process
      begin
      wait until clock = '1';            --  front montant
         case etat is
         when '0'    =>
            if (j='1') then
                  etat <= '1';
            end if;
         when '1'    =>
            if (k='1') then
                  etat <= '0';
            end if;
         end case;                -- les autres cas d'entrees ne sont pas renseignes pour sequentiel (memorisation)
      end process;
end primitive;
```

JK Etat interne (variante)

```
-- Bascule JK a partir de l'etat interne de la bascule et non plus des commandes j et k (variante)  (+edge)
entity jk4 is
      port (j,k,clock : in bit;        -- entree
                q : out bit);      -- sortie
end jk4;

architecture primitive of jk4 is
signal etat : bit;
begin
q <= etat;
process(clock)
      begin
      if (clock = '1' and clock 'event) then                    -- front montant
         case etat is
         when '0'    =>
            if (j='1') then
                  etat <= '1';
            end if;
         when '1'    =>
            if (k='1') then
                  etat <= '0';
            end if;
         end case;                -- les autres cas d'entrees ne sont pas renseignes pour sequentiel (memorisation)
      end if;
      end process;
end primitive;
```

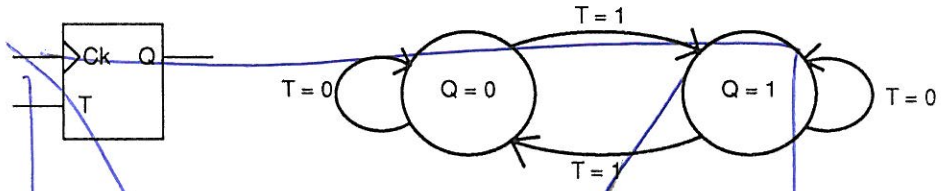


Figure III-26

Un exemple de réalisation

L'examen du diagramme de transitions de la figure III-26 nous montre que Q(t) = 1 si

Q(t-1) = '1' et T = '0',

ou

Q(t-1) = '0' et T = '1'

Ce qui nous fournit l'équation de l'entrée D d'une bascule D :

$$D = T \oplus Q$$

D'où le logigramme :

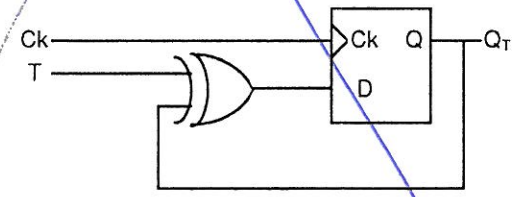


Figure III-27

Description en VHDL

La description d'une bascule T se déduit simplement du diagramme de transition :


```
entity T_edge is
port ( T,hor: in bit;
       s : out bit);
end T_edge;

architecture d_primitive of T_edge is
signal etat : bit;
begin
s <= etat ;
process
begin
wait until hor = '1' ;
if(T = '1') then
      etat <= not etat;
end if;
end process;
end d_primitive;
```

On rappelle que de tels exemples sont fournis à titre d'illustration du fonctionnement de l'opérateur considéré, et pour familiariser le lecteur avec le langage VHDL. On n'a jamais besoin, en pratique, de décrire chaque bascule utilisée dans ce langage !

Les applications

L'un des intérêts principaux des bascules de type T est qu'elles permettent de générer de façon extrêmement simple des compteurs binaires synchrones. Un compteur binaire est une fonction séquentielle synchrone dont l'état interne est un nombre entier naturel codé en binaire dont chaque chiffre binaire est matérialisé par une bascule. A chaque transition active de l'horloge ce nombre est incrémenté de 1, quand le nombre maximum est atteint, toutes les bascules sont à 1, la séquence recommence à partir de 0. Si le nombre de bits utilisés est n, on parlera d'un compteur modulo 2^n. La simple observation d'une table des entiers naturels écrits en base 2 nous fournit la clé du problème : la bascule de rang i doit changer d'état quand toutes les bascules de rang inférieur sont à 1.

D'où un exemple de réalisation d'un compteur synchrone modulo 16 dont on peut interrompre le comptage (en = '0') :

```
ENTITY cnt16 IS
PORT (ck, en : IN BIT;
      s : OUT BIT_VECTOR (0 TO 3)
      );
END cnt16;

ARCHITECTURE structurelle OF cnt16 IS
SIGNAL etat : BIT_VECTOR(0 TO 3);
SIGNAL inter: BIT_VECTOR(1 TO 3);
COMPONENT T_edge
```

```
-- la même que dans l'exemple précédent
port ( T,hor: in bit;
       s : out bit);
END COMPONENT;

BEGIN
-- Etablir le logigramme tout en lisant le texte
s <= etat ;
inter(1) <= etat(0) and en ;
inter(2) <= etat(1) and inter(1) ;
inter(3) <= etat(2) and inter(2) ;
g0 : T_edge port map (en,ck, etat(0));
g1 : for i in 1 to 3 generate
       g2 : T_edge port map (inter(i),ck,etat(i));
end generate;
END structurelle;
```

Là encore mettons en garde le lecteur, quand on a réellement besoin d'un compteur on écrit

```
etat <= etat + 1 ;
```

c'est nettement plus simple, le compilateur générera de lui même les interconnexions nécessaires entre les bascules.

L'ancêtre vénérable : la bascules J-K

Le principe

Quelque peu tombée en désuétude, la bascule J-K a régné en maître dans le monde de la logique séquentielle des décennies 60 et 70. Elle est l' héritière directe de la bascule R-S, que l'on a débarassé progressivement de ses difficultés asynchrones. Les premières bascules J-K n'étaient, en fait, pas de réels opérateurs synchrones, elles comportaient tout un mécanisme de mémorisation interne des commandes dans des bascules R-S (maître-esclave). Les versions actuelles sont construites au moyen d'une bascule D-edge et de logique combinatoire.

Une bascule J-K dispose de deux commandes, J et K, outre l'horloge. Son fonctionnement se décrit bien au moyen d'une table qui décrit la fonction réalisée en fonction des valeurs de la commande :

J(t-1)	K(t-1)	Fonction	Equation
0	0	Mémoire	Q(t) = Q(t-1)
0	1	Mise à zéro synchrone	Q(t) = '0'
1	0	Mise à un synchrone	Q(t) = '1'
1	1	Changement d'état	Q(t) = $\overline{Q(t-1)}$

Comme pour tout opérateur synchrone, l'état de la bascule ne change pas entre deux transitions actives du signal d'horloge. La fonction « mémoire », dans la table ci-dessus, signifie que la bascule conserve son état précédent même lors d'une transition d'horloge. Dans la construction du diagramme de transitions de la figure suivante, III-28, on a tenu compte de ce que chaque transition peut être obtenue par deux combinaisons différentes des commandes J et K, la transition '0' → '1', par exemple, peut être obtenue par mise à 1 explicite (JK = "10") ou par changement d'état (JK = "11") :

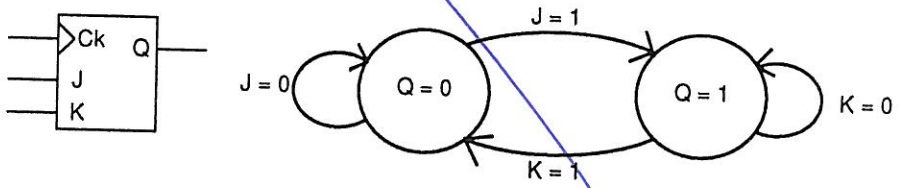


Figure III-28

On déduit aisément l'équation de la bascule J-K de son diagramme de transition :

$$Q(t) = J * \overline{Q(t-1)} + \overline{K} * Q(t-1)$$

Au lieu de raisonner sur l'équation de l'état futur, on peut décrire la bascule J-K par son équation de transition, si on introduit une variable binaire auxilliaire T_{JK}, égale à '1' si une transition doit avoir lieu, '0' autrement, on obtient :

$$T_{JK} = J * \overline{Q} + K * Q$$

Description en VHDL

La première description que nous donnerons est la simple traduction naïve de la table de vérité :

```
entity jk is port (
```

```
                moore_state <= O"7" ;
            end if ;
when O"2" => if man = '0' -- etat inutiles
             then
                moore_state <= O"6" ;
            else
                moore_state <= O"5" ;
            end if ;
when O"3" => if man = '0' then -- bis
                moore_state <= O"4" ;
            else
                moore_state <= O"7" ;
            end if ;
   end case ;
end process moore_mach ;
end comporte ;
```

Codage des états

Quand on utilise des circuits standard, des compteurs programmables, par exemple, pour réaliser une machine séquentielle, le codage des états du diagramme de transitions est, de fait, imposé par le circuit cible. Il en va tout autrement quand la dite machine doit être implantée dans un circuit programmable ou un ASIC. Libéré des contraintes liées à une quelconque fonction prédéfinie, le concepteur peut, à loisir, adapter le codage des états à l'application qu'il est en train de réaliser.

Le choix d'un code est particulièrement important quand on s'oriente vers la réalisation d'une machine de Moore. L'exemple du décodeur Manchester nous a appris que l'un des avantages de cette architecture réside dans la possibilité de générer les sorties directement à partir du registre d'état, donc dénuées de tout parasite lié à leur calcul. Mais, comme nous le verrons dans deux exemples, l'identification des sorties du système à celles des bascules du registre d'état ne suffit généralement pas pour définir le codage des états.

Ce choix du codage mérite une grande attention, il conditionne grandement la complexité de la réalisation, sa bonne adaptation au problème posé ; un choix judicieux conduira à un résultat simple et facilement testable, alors qu'aucun logiciel d'optimisation ne compensera des erreurs de décision à ce niveau.

Le nombre d'états nécessaires et le type de code adopté fixent, en premier lieu, la taille du registre d'état. Schématiquement, si n est la taille, en nombre de bits, du registre d'état, et N_e le nombre d'états nécessaires, ces deux nombres (entiers !) doivent vérifier la double inégalité :

$$n \leq N_e \leq 2^n$$

Si l'inégalité de gauche n'est pas vérifiée, certaines bascules sont probablement inutiles ; quand cette inégalité se transforme en égalité, on utilise un code très

« dilué », une bascule par état, qui présente l'avantage de la lisibilité, mais le danger de générer en grand nombre des états accessibles inutilisés (rappelons ici qu'il y a toujours 2^n états accessibles).

Si l'inégalité de droite n'est pas vérifiée, la tentative est sans espoir ; si elle se transforme en égalité, on utilise un encodage « fort », auquel il faudra très probablement adjoindre des fonctions combinatoires de calcul des sorties ; on ne réalise pas que des compteurs binaires ou des codeurs de position absolue (code de Gray).

Les situations intermédiaires correspondent en général à des codes adaptés aux sorties.

Encodage « fort » ou code « dilué » ? En caricaturant un peu, on peut dire que les tenants de la première solution préfèrent les fonctions combinatoires, et que les seconds sont des adeptes des bascules. Il n'est pas évident, à priori, de prévoir la complexité des équations engendrées par tel ou tel code. On gagne souvent à suivre le fonctionnement « naturel » de la machine[24], et, surtout, on gagne à se souvenir que les ordinateurs, et leurs compilateurs, ne sont pas posés sur un bureau à titre de décoration ; ils permettent de voir très vite quelle est la complexité sous jacente d'un choix, sans pour celà tomber dans le BAO[25].

Codes adaptés aux sorties

L'idée qui vient naturellement à l'esprit est de choisir le codage en fonction des sorties à générer. C'est souvent la méthode la plus souple, celle qui conduit aux équations les plus faciles à interpréter, et pas forcément plus compliquées que celles que l'on obtiendrait avec d'autres codes.

Une commande de feux tricolores.

Pour satisfaire à une tradition bien établie, nous prendrons comme premier exemple une commande de feux de circulation routière.

Un passage pour piétons traverse une avenue ; il est protégé par un feu tricolore qui fonctionne à la demande des piétons : En l'absence de toute demande, les feux sont à l'orange clignotant (un nombre Tor de secondes allumés, Tor secondes éteints). Quand un piéton souhaite traverser l'avenue, il est invité à appuyer sur un bouton, ce qui provoque le déclenchement d'une séquence (vue des voitures) :

- orange fixe pendant 2*Tor secondes,
- rouge pendant Tr secondes,
- vert pendant Tv secondes, pour laisser passer le flot de voitures pendant un minimum de temps,
- retour à la situation par défaut.

[24] Mais qu'est-ce que ce fonctionnement naturel ? Sa recherche est, sans doute, l'une des parties les plus intéressantes, et donc souvent difficile, du travail.

[25] Bricolage Assisté par Ordinateur

Profitons de cet exemple pour subdiviser la solution du problème en sous ensembles. Trois blocs fonctionnels peuvent être identifiés :

1. La commande des feux proprement dite, les sorties de trois bascules du registre d'état commandent directement l'allumage, ou l'extinction, des lampes rouge, verte et orange.
2. Une temporisation qui, suite à une commande d'initialisation, fournit les trois durées Tor, Tr et Tv.
3. Une mémorisation de l'appel des piétons, qui évite de se poser des questions concernant la durée pendant laquelle le demandeur appuie sur le bouton ; une simple pression suffit, l'appel est alors enregistré, quel que soit l'état d'avancement de la séquence de gestion des feux.

Outre les commandes des feux proprement dites, le bloc principal fournit un signal d'initialisation (cpt) à la temporisation, qui doit durer une période d'horloge[26], et un signal d'annulation (raz) de la requête, mémorisée, d'un piéton.

Les signaux d'entrée de ce bloc sont la requête (piet) et les trois indications de durée Tor Tr et Tv ; nous supposerons que ces dernières passent à '1', pendant une période d'horloge, quand les durées correspondantes se sont écoulées.

D'où le synoptique de la figure V-17 :

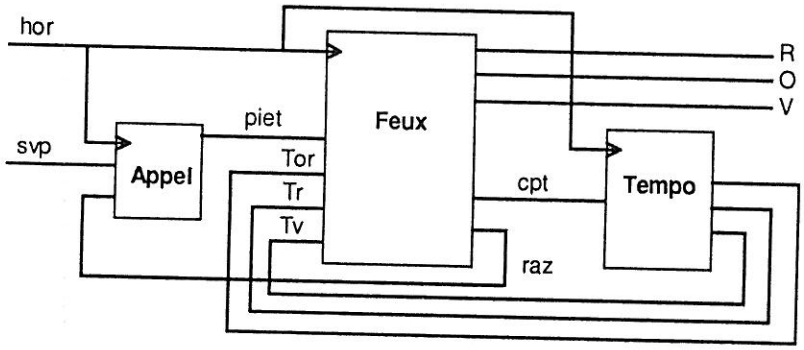


Figure V-17

[26] Nous sommes en train de définir trois processus qui se commandent et/ou s'attendent mutuellement. Le danger de ce type d'architecture, très fréquente, est de générer des interblocages : un processus en initialise un second et attend une réponse de ce dernier. Si le demandeur oublie de relâcher la commande d'initialisation, le système est bloqué. Ce type de situation porte, en informatique, le doux nom d'étreinte fatale (*deadly embrace*). La solution adoptée ici est d'envoyer des signaux fugaces (mais synchrones !), ce qui oblige le demandeur à attendre la réponse dans un état différent de celui où il a passé la commande d'initialisation

Nous nous contenterons d'étudier, ici, le bloc principal, feux, laissant la synthèse des deux autres blocs à titre d'exercice.

Première ébauche :

Le fonctionnement général peut être celui illustré par la figure V-18 :

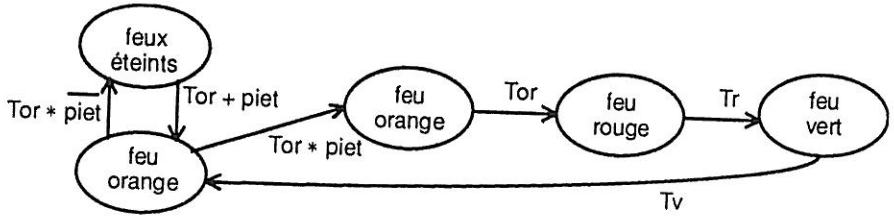


Figure V-18

Précisions :

A partir de l'ébauche précédente, il nous reste à préciser le mode de calcul des signaux gérés par le processus feux, et à en déduire le codage des états. Le signal cpt se prête bien à une réalisation sous forme de sortie de Mealy, les signaux de commande des feux à une réalisation sous forme de sorties de Moore. Les deux états où le feu orange est allumé doivent être distingués, une bascule supplémentaire, qui n'est attachée à aucune sortie, doit être rajoutée à cette fin. La sortie raz peut être identique à la sortie qui correspond au feu rouge ; il n'est pas utile de mémoriser une demande de piéton quand les voitures sont arrêtées au feu rouge. D'où une version plus élaborée du diagramme de transitions (figure V-19) :

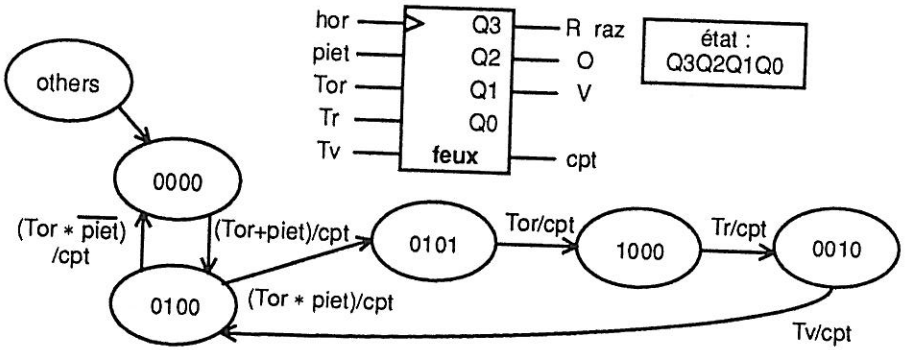


Figure V-19

Programme VHDL :

Un exemple de programme VHDL, qui correspond au module feux uniquement, est fourni ci-dessous ; il se déduit directement du diagramme de transitions précédent.

```
entity feux is
   port ( hor, piet, Tor, Tr, Tv : in bit ;
     R, O, V, cpt : out bit );
end feux ;

architecture comporte of feux is
signal etat : bit_vector(3 downto 0) ;
begin
R <= etat(3) ;
O <= etat(2) ;
V <= etat(1) ;

machine : process -- diagramme de transitions.
begin
   wait until hor = '1' ;
      case etat is
         when X"0" -- états en hexadécimal.
                 => if (Tor or piet) = '1' then
                       etat <= X"4" ;
                    end if ;
         when X"4" => if (Tor and piet) = '1' then
                       etat <= X"5" ;
            elsif (Tor and not piet) = '1' then
                       etat <= X"0" ;
                    end if ;
         when X"5" => if Tor = '1' then
                       etat <= X"8" ;
                    end if ;
         when X"8" => if Tr = '1' then
                       etat <= X"2" ;
                    end if ;
         when X"2" => if Tv = '1' then
                       etat <= X"4" ;
                    end if ;
         when others => etat <= X"0" ;
            -- pour les états inutilisés.
      end case ;
end process machine ;

mealy : process -- calcul de la sortie cpt.
begin
   wait on etat, piet, Tor, Tr, Tv ; -- liste de sensibilité
```



```
   cpt <= '0' ; -- assure un bloc combinatoire.
      case etat is
         when X"0" => if Tor = '1' or piet = '1' then
                         cpt <= '1' ;
                      end if ;
         when X"4" => if Tor = '1' then
                         cpt <= '1' ;
                      end if ;
         when X"5" => if Tor = '1' then
                         cpt <= '1' ;
                      end if ;
         when X"8" => if Tr = '1' then
                         cpt <= '1' ;
                      end if ;
         when X"2" => if Tv = '1' then
                         cpt <= '1' ;
                      end if ;
         when others => null ; -- case complet.
      end case ;
end process mealy ;
end comporte ;
```

Le décodeur Manchester réexaminé.

Comme deuxième exemple, reprenons, en la complétant un peu, l'étude du décodeur Manchester différentiel. Nous avions omis, dans la version précédente, un deuxième signal de sortie, rx, qui indique aux utilisateurs la cadence de transmission. Comme on peut le voir sur la figure V-20 , ce signal a une fréquence moitié de celle de l'horloge, mais il ne peut pas s'agir d'un simple diviseur par deux : un diviseur par deux est incapable de distinguer les transitions systématiques des transitions signifiantes du signal d'entrée man, il est incapable de se synchroniser.

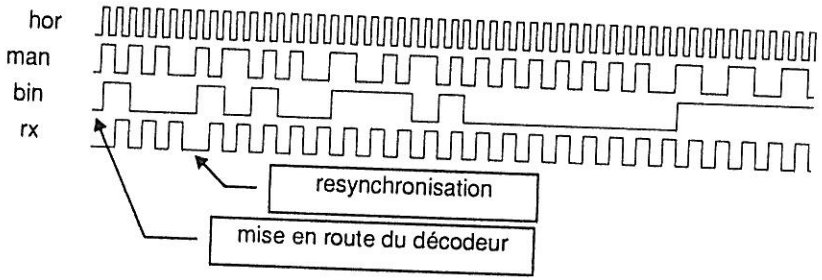


Figure V-20

ELECTRONIQUE

NUMERIQUE

CORRIGES

ANNEXE

TD 6R CORRIGE. REVISION LOGIQUE COMBINATOIRE & SEQUENTIELLE

1. Transcodeur Grey sur 3 bits *abc* → BCD sur 3 bits *xyz*

y : a \ bc	00	01	11	10
0	**0**	**0**	**1**	**1**
1	**1**	**1**	**0**	**0**

z : a \ bc	00	01	11	10
0	**0**	**1**	**0**	**1**
1	**1**	**0**	**1**	**0**

$x = a$ (immédiat)

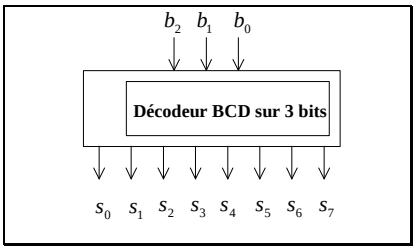
$y = a\bar{b} + \bar{a}b = a \oplus b$

$z = a \oplus b \oplus c$

2. Inhibition

$$Y = A \cdot \bar{I}$$

3. Démultiplexeur 1→8



4. Multiplexeur 2 → 1

a- $$Y = A \cdot \bar{S} + B \cdot S$$

b- $$Y = (S + A) \cdot (\bar{S} + B) \qquad \bar{Y} = S \cdot \bar{B} + \bar{S} \cdot \bar{A}$$

5. Multiplexeurs 4→1

Multiplexeur 1
e_0 = **0**
e_1 = **0**
e_2 = **0**
e_3 = **1**

Multiplexeur 2
e_0 = **1**
e_1 = **0**
e_2 = **0**
e_3 = **0**

Multiplexeur 3
e_0 = **1**
e_1 = **0**
e_2 = C
e_3 = **0**

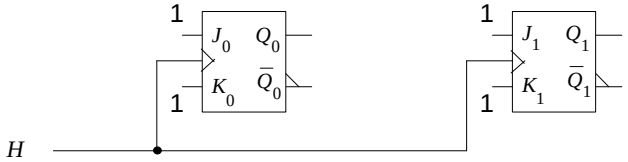
6. Synthèse de Compteur synchrone 2 bits à bascules JK

a-

J_0 K_0 : Q_1 \ Q_0	0	1
0	1 X	X X
1	X X	X 1

J_1 K_1 : Q_1 \ Q_0	0	1
0	1 X	X X
1	X X	X 1

$$\begin{cases} J_0 = 1 \\ K_0 = 1 \end{cases} \qquad \begin{cases} J_1 = 1 \\ K_1 = 1 \end{cases}$$



ARCHITECTURE

DES

ORDINATEURS

CORRIGES

TD 1 ADO Corrigé. Codage

On pourra s'aider de la calculatrice Windows – mode scientifique – pour vérification des conversions Décimal/Binaire/Hexadécimal

Exercice 1 : nombre relatifs sur machine

0.

Intervalle des entiers (en non signé) sur 16 bits (2^{16} éventualités de codage) : N = 0 à 2^{16}-1 (soit 0 à +65 535).

1.

Intervalle des entiers relatifs en Complément à 2 (C2) sur 16 bits (2^{16} éventualités de codage) : N = -2^{15} à 2^{15}-1 (soit -32 768 à +32 767).

2.

a) $2010_{10} = 7*16^2 + 13*16^1 + 10*16^0 = 07DA_H = 0000\ 0111\ 1101\ 1010_H$

b) $7DA_H = 7*16^2 + 13*16^1 + 10*16^0 = 2010_{10}$

3.

a) $0110\ 1100\ 0001\ 1011_{(C2)} = 2^{14}+2^{13}+2^{11}+2^{10}+2^4+2^3+2^1+2^0 = 16\ 384 + 8\ 192 + 2\ 048 + 1\ 024 + 16 + 8 + 2 + 1 = 27\ 675_{(10)}$

b) $1011\ 0110\ 1011\ 0011_{(C2)} = ?$

Ce nombre étant négatif (bit MSB = 1), on calcule sa valeur positive (valeur absolue) en faisant le calcul inverse que pour le passage vers le code C2 :

N → C2 = C1(N) + 1 C2 → N = C1(C2 – 1) = C1(C2 + 1111 1111 1111 1111)

C2 – 1 = 1011 0110 1011 0011 + 1111 1111 1111 1111 = 1011 0110 1011 0010

C1(C2 – 1) = C1(1011 0110 1011 0010) = 0100 1001 0100 1101

Or :

$0100\ 1001\ 0100\ 1101_{(C2)} = 2^{14}+2^{11}+2^8+2^6+2^3+2^2+2^0 = 16\ 384 + 2\ 048 + 256 + 64 + 8 + 4 + 1 = 18\ 765_{(10)}$

D'où : $1011\ 0110\ 1011\ 0011_{(C2)} = -18\ 765_{(10)}$

Autre méthode : on prend à nouveau le C2 du mot négatif pour avoir sa valeur positive : C2(C2) = N

C2 de 1011 0110 1011 0011 = 0100 1001 0100 1100 + 1 = 0100 1001 0100 1101

Or :

$0100\ 1001\ 0100\ 1101_{(C2)} = 2^{14}+2^{11}+2^8+2^6+2^3+2^2+2^0 = 16\ 384 + 2\ 048 + 256 + 64 + 8 + 4 + 1 = 18\ 765_{(10)}$

D'où : $1011\ 0110\ 1011\ 0011_{(C2)} = -18\ 765_{(10)}$

Remarque : aucune de ces méthodes ne permet de décoder 1000 0000 0000 0000 qui est égal à $-2^{15} = -32\ 768$.

4.

C2 sur 8 bits : -128 à 127 :

$122_{(10)} = 0111\ 1010_{(C2)}$

$-7_{(10)} = 1111\ 1001_{(C2)}$

$111_{(10)} = 0110\ 1111_{(C2)}$

$-111_{(10)} = 1001\ 0001_{(C2)}$

$17_{(10)} = 0001\ 0001_{(C2)}$

$-17_{(10)} = 1110\ 1111_{(C2)}$

Calculer en Complément à 2 sur 8 bits les additions suivantes (données en décimal) :

a) 122 + (-7) :

```
1 1111 000     (retenues)
  0111 1010    (122)
 +1111 1001    (-7)
  ---------
1 0111 0011    (115) => 0111 0011 représente bien 115(10)
               (2 dernières retenues à 1)
```

b) (-111) + (-17) :

```
1 1111 111      (retenues)
  1001 0001     (-111)
 +1110 1111     (-17)
  ---------
1 1000 0000     (-2^7 = -128) => 1000 0000 représente bien -128(10)
                 (2 dernières retenues identiques)
```

c) 111 + 17 :

```
01111 111       (retenues)
 0110 1111      (111)
+0001 0001      (17)
 ---------
 1000 0000      (-2^7 = -128) => 1000 0000 représente -2^7 = -128(10)
                (erreur ≠ 128   / 2 dernières retenues différentes)
```

Exercice 2 : nombres à virgule fixe

Rappels : **Conversion décimal binaire : Partie entière**

exemple : convertir 125

On divise le nombre 125 par 2 autant qu'il est possible, les restes successifs étant les poids binaires obtenus dans l'ordre des puissances croissantes.

125	: 2 =	**62**	reste	**1**
62	: 2 =	**31**	reste	**0**
31	: 2 =	**15**	reste	**1**
15	: 2 =	**7**	reste	**1**
7	: 2 =	**3**	reste	**1**
3	: 2 =	**1**	reste	**1**
1	: 2 =	**0**	reste	**1**

↓

$125_{(10)}$ = 0111 $1101_{(2)}$ sur 8 bits

←

Conversion : Partie fractionnaire

- *binaire vers décimal :* La partie fractionnaire représente les coefficients des puissances négatives de 2, soit :

$2^{-1} = ½ = 0,5$; $2^{-2} = 1/4 = 0,25$; $2^{-3} = 1/8 = 0,125$ … $2^{-p} = 1/(2^p)$

Ainsi $0,101_{(2)} = 1.2^{-1}+0.2^{-2}+1.2^{-3} = 0,5+0,125 = 0,625_{(10)}$

- *décimal vers binaire :* Pour convertir la partie fractionnaire d'un nombre en base 10 en un nombre en base 2, on multiplie la partie fractionnaire par 2, la partie entière obtenue représentant le poids binaire, la nouvelle partie fractionnaire étant à nouveau multipliée par 2 :

0,625 x 2 = **1**,25
0,25 x 2 = **0**,5
0,5 x 2 = **1**,0

↑

d'où $0{,}625_{(10)} = 1.2^{-1}+0.2^{-2}+1.2^{-3}$ s'écrit $0{,}101_{(2)}$ → sur 8 bits : **0111 1101, 1010 0000**

← ←

ainsi $125{,}625_{(10)} = 1111101{,}101_{(2)}$

1. Donner l'équivalent binaire des nombres décimaux suivants :
   a) $118{,}625_{(10)} = 118 + 0.625 = 1110110{,}101_{(2)}$
   b) $1/10_{(10)} = 0{,}\underline{00011}\ldots_{(2)}$
   c) $4/3_{(10)} = 1 + 1/3 = 1{,}\underline{01}\ldots_{(2)}$

2. Exprimez en base 10 le nombre binaire suivant :
   $101{,}101_{(2)} = ?$
   Partie entière : $101_{(2)} = 1x2^2 + 0x2^1 + 1x2^0 = 5_{(10)}$; Partie décimale : $101_{(2)} = 1x2^{-1} + 0x2^{-2} + 1x2^{-3} = 0{,}625_{(10)}$
   → $101{,}101_{(2)} = 5{,}625_{(10)}$

Exercice 3 : nombres à virgule flottante (IEEE754)

1. Donner l'équivalent binaire en simple précision des nombres décimaux suivants :

a) 1,5 = $2^k \cdot (1, \cdots) = 1 \times 1{,}5 = 2^0 \times (1 + 0{,}5) = 2^{127-127} \times (1 + 0{,}5)$ $(k \in \mathbf{Z})$
s (1) = **0**
e (8) = 127 = **0111 1111**
m (23) = 0,5 = 2^{-1} = **100 0000 0000 0000 0000 0000**
sem = **0011 1111 1100 0000 0000 0000 0000 0000 = 3F C0 00 00$_{(H)}$**

b) 0,5 = $2^k \cdot (1, \cdots) = 0{,}5 \times 1{,}0 = 2^{-1} \times (1 + 0{,}0) = 2^{126-127} \times (1 + 0{,}0)$
s (1) = **0**
e (8) = 126 = **0111 1110**
m (23) = 0 = **000 0000 0000 0000 0000 0000**
sem = **0011 1111 0000 0000 0000 0000 0000 0000 = 3F 00 00 00$_{(H)}$**

c) -142,625 = $2^k \cdot (1, \cdots) = -128 \times 1{,}114\ 257\ 8125 = -2^7 \times (1 + 0{,}114\ 257\ 8125) = -2^{134-127} \times (1 + 0{,}114\ 257\ 8125)$
s (1) = **1**
e (8) = 134 = **1000 0110**
m (23) = 0,114 257 8125 = **000 1110 1010 0000 0000 0000**
sem = **1100 0011 0000 1110 1010 0000 0000 0000 = C3 0E A0 00$_{(H)}$**

d) 10 = $2^k \cdot (1, \cdots)$ = 8 x 1,25 = 2^3 x (1 + 0,25) = $2^{130-127}$ x (1 + 0,25)

s (1) = **0**

e (8) = 130 = **1000 0010**

m (23) = 0,25 = 2^{-2} = **010 0000 0000 0000 0000 0000**

sem = **0100 0001 0010 0000 0000 0000 0000 0000 = 41 20 00 00$_{(H)}$**

e) 1/10 = $2^k \cdot (1, \cdots)$ = 0,0625 x 1,6 = 2^{-4} x (1 + 0,**6**) = $2^{123-127}$ x (1 + 0,**6**)

s (1) = **0**

e (8) = 123 = **0111 1011**

m (23) = 0,6 = **100 1100 1100 1100 1100 1100** (partie fractionnaire infinie)

sem = **0011 1101 1100 1100 1100 1100 1100 1100 = 3D CC CC CC$_{(H)}$**

2. Faire l'addition en IEEE754 simple précision de 1/10 et 1/10. Reconnaître le résultat.

Méthode pour une addition en IEEE 754

a. Ramener les deux nombres au même exposant
b. Restaurer le bit de poids fort
c. Effectuer l'addition ou la soustraction des valeurs absolues comme pour les entiers
d. Renormaliser le résultat (arrondi, bit de poids fort, exposant)

1/10 = $2^k \cdot (1, \cdots)$ = 0,0625 x 1,6 = 2^{-4} x (1 + 0,**6**) = $2^{123-127}$ x (1 + 0,**6**)

s (1) = **0**

e (8) = 123 = **0111 1011**

m (23) = 0,6 = **100 1100 1100 1100 1100 1100** (partie fractionnaire infinie)

sem = **0011 1101 1100 1100 1100 1100 1100 1100 = 3D CC CC CC$_{(H)}$**

1/10 + **1/10** =

⇨ $1,100\ 1100\ 1100\ 1100\ 1100\ 1100 * 2^{123-127}$

\+ $1,100\ 1100\ 1100\ 1100\ 1100\ 1100 * 2^{123-127}$

$11,001\ 1001\ 1001\ 1001\ 1001\ 1000 * 2^{123-127}$ **→ à normaliser**

$1,100\ 1100\ 1100\ 1100\ 1100\ 1100 * 2^{124-127}$ dernier bit (0) perdu - n'importe quel arrondi)

soit pour le résultat : **1/10** + **1/10** =

s (1) = **0**

e (8) = 124 = **0111 1100**

m (23) = **100 1100 1100 1100 1100 1100** (partie fractionnaire infinie)

sem = **0011 1110 0100 1100 1100 1100 1100 1100 = 3E 4C CC CC$_{(H)}$** (même mantisse que 1/10, exposant+1)

A comparer au résultat 2/10 :

2/10 = $2^k \cdot (1, \cdots)$ = 0,125 x 1,6 = 2^{-3} x (1 + 0,**6**) = $2^{124-127}$ x (1 + 0,**6**)

s (1) = **0**

e (8) = 124 = **0111 1100**

m (23) = 0,6 = **100 1100 1100 1100 1100 1100** (partie fractionnaire infinie)

sem = **0011 1110 0100 1100 1100 1100 1100 1100 = 3E 4C CC CC$_{(H)}$** (même résultat que 1/10 + 1/10) **OK**

3. Faire la multiplication en IEEE754 simple précision de -18 par 10.

Méthode pour une multiplication en IEEE 754

a. Calculer le signe puis la somme des exposants
b. Restaurer le bit de poids fort
c. Effectuer la multiplication des valeurs absolues comme pour les entiers
d. Eventuellement, arrondir, ajuster l'exposant et renormaliser.

X = -18 = $2^k \cdot (1, \cdots)$ = - 16 x 1,125 = - 2^4 x (1 + 0,125) = - $2^{131-127}$ x (1 + 0,125)

s (1) = **1**

e (8) = 4 % 127 = 131 = **1000 0011**

m (23) = 0,125 = 2^{-3} = **001 0000 0000 0000 0000 0000** = m_X

sem = **1100 0001 1001 0000 0000 0000 0000 0000 = C1 90 00 00$_{(H)}$**

Y = 10 = $2^k \cdot (1, \cdots)$ = 8 x 1,25 = 2^3 x (1 + 0,25) = $2^{130-127}$ x (1 + 0,25)

s (1) = **0**

e (8) = 3 % 127 = 130 = **1000 0010**

m (23) = 0,25 = 2^{-2} = **010 0000 0000 0000 0000 0000** = m_Y

sem = **0100 0001 0010 0000 0000 0000 0000 0000 = 41 20 00 00$_{(H)}$**

Z = X x **Y = (-18)** x **10** =

signe : **1** (xor entre les 2 signes des 2 opérandes)

exposant : **1000 0011 + 1000 0010 - (127)$_2$ = 1000 0011 + 1000 0010 – 0111 1111**
= 1000 0011 + 1000 0010 + (-127)$_{C2}$ = 1000 0011 + 1000 0010 + 1000 0001 = 0000 0101 + 1000 0001 = 1000 0110
(= 134$_{10}$) = 7 + 127 (**OK : 7 = 4 + 3)**

mantisse : Multiplication des mantisses m_X et m_Y :

remarque : $(1 + m_X)(1 + m_Y) = 1 + m_X.m_Y + m_X + m_Y$ → **$m_Z = m_X.m_Y + m_X + m_Y$**

```
      1,mX =          1,001 0…0
   x  1,mY =       x  1,01  0…0
                   -----------
                     1001
                    0000
                   1001
                   -----------
      1,mZ =       1,01101 0…0
```

(attention à la troncature: le produit de 2 mots de 24 bits a 48 bits)
(attention à la normalisation: les retenues peuvent faire qu'à gauche de la virgule on a plus que 1,)

→ **m_Z = 01101** 0…0 → **m_Z (23) = 011 0100 0000 0000 0000 0000**

Interprétation du résultat : **Z**

s (1) : **1**

e (8) : **1000 0110** → **e = 134** → **exposant = 134-127 = 7** % 127

m (23): **011 0100 0000 0000 0000 0000** → **mantisse =** $2^{-2} + 2^{-3} + 2^{-5}$ = 0,40 625

sem = 1100 0011 0011 0100 0000 0000 0000 0000 = C3 34 00 00$_{(H)}$

Z = - 1,40 625 x 2^7 = - 1,40 625 x 128 = **- 180**

Vérification du résultat : **Z**

Z (exact) = - 180 = $2^k \cdot (1, \cdots)$ = - 128 x 1,40 625 = - 2^7 x (1 + 0,40 625) = - $2^{134-127}$ x (1 + 0,40 625)

s (1) = **1**

e (8) = 134 = **1000 0110 = 7** % 127

m (23) = 0,40 625 = $2^{-2} + 2^{-3} + 2^{-5}$ = **011 0100 0000 0000 0000 0000**

sem = **1100 0011 0011 0100 0000 0000 0000 0000= C3 34 00 00$_{(H)}$**

TD 2 ADO Corrigé. Architecture de Von Neumann

Exercice 1 : Modes d'adressage (Complément de cours)

Les modes d'adressage

Un mode d'adressage est une méthode permettant d'interpréter, d'accéder à un opérande (aux données) lors de l'exécution d'une instruction. Par exemple l »assembleur MC68000 de Motorola présente 6 modes d'adressage :

1. Adressage Direct : l'opérande est un registre de données ou d'adresse.

2. Adressage Indirect : l'opérande est désigné :
- soit par le contenu d'un registre d'adresse,
- soit par l'addition du contenu d'un registre d'adresse et d'une constante (*offset*) et/ou du contenu d'un registre de donnée ou d'adresse (*index*).

3. Adressage Immédiat : la donnée est fournie dans le code instruction.

4. Adressage Absolu : l'adresse de la donnée est fournie dans le code instruction.

5. Adressage Relatif : l'adresse de la donnée est calculée par addition du contenu du Compteur Ordinal et d'un *offset* et/ou d'un *index*.

6. Adressage Implicite : les registres impliqués sont les registres de contrôle (Registre d'Etat, Compteur Ordinal, Pile).

Motorola donne le tableau 4.1 récapitulant les modes d'adressage avec les codes associés. An et Dn désignent respectivement les registres d'adresse et de donnée.

Mode	Code	Champ registre	Syntaxe
Direct	000	Num. reg.	`Dn`
Direct	000	Num. reg.	`An`
Indirect	010	Num. reg.	`(An)`
Indirect	011	Num. reg.	`(An)+`
Indirect	100	Num. reg.	`-(An)`
Indirect	101	Num. reg.	`d(An)`
Indirect	110	Num. reg.	`d(An,Rm)`
Absolu	111	000	`xxxx`
Absolu	111	001	`xxxxxxxx`
Relatif	111	010	Rel. CO
Relatif	111	011	Rel. CO+Rm
Immédiat	111	100	`#xxxx`

Table 4.1: Modes d'adressage du M68000.

Exemple 1 (Instructions de transfert)

`move,w d3,-(a4)` : les 16 bits de poids faible (extension `,w`) du registre `d3` sont transférés à l'adresse donnée par le contenu du registre `a4` décrémenté de 1 avant transfert. Le rangement en mémoire se fait octet de poids faible d'abord

Exemple 2 (Instructions de comparaison)

`cmp,l d4,d2`: compare d4 à d2 en effectuant la soustraction d2−d4. Les drapeaux (sauf X) sont modifiés en conséquence.

Exemple 3 (Instructions de branchement inconditionnel)

`bra plus_loin`: l'argument du branchement est un déplacement sur 8 ou 16 bits qui est ajouté au contenu du compteur ordinal (celui-ci contient alors l'adresse de l'instruction qui suit).

`jmp (a3)` : branchement à l'adresse donnée par le contenu de a3.

L'instruction ASSEMBLEUR INTEL 8086 :

AND AX, 06 *(AX est un registre accumulateur 16 bits)*

a pour code Machine :

25 06 00$_H$ *(convention INTEL Little Endian)*

Elle est implantée à l'adresse 01 00$_H$.

a) Indiquer le contenu des registres IR et IP juste avant exécution de l'instruction.

b) Indiquer le contenu du registre IP juste après exécution de l'instruction.

Corrigé :

a) Juste avant :

IR : 25 06 00 IR contient le code Machine de la prochaine instruction à exécuter

IP : 01 00 01 00$_H$ = adresse d'implantation du programme = contenu de IP, noté (IP) = adresse de la prochaine instruction à exécuter

b) Juste après :

IP : 01 03 01 00$_H$ + 3 octets = 01 03$_H$ (le code instruction occupe 3 octets)

Exercice 2 : Programme ASSEMBLEUR (ASM)

Soit l'extrait de programme ASSEMBLEUR INTEL 8086 suivant, stocké à l'adresse 01 00$_H$ (via le code ASM **ORG 100h**) avec les valeurs initiales : **AX = 00 00$_H$** , **BX = 00 00$_H$** et l'état de pile (STACK) suivant : **STACK** : **FF FE$_H$: 00 00$_H$**
FF FC$_H$: 00 00$_H$
FF FA$_H$: 00 00$_H$
…

IP		*Code ASM*	*Code Machine*	*Commentaire* *(AX) signifie : contenu de AX*
01 00		**MOV AX, 0100h**	B8 00 01 **(3)**	Ecrit 01 00$_H$ dans le registre AX ; **(AX) = 01 00** (convention INTEL Little Endian)
01 03		**MOV BX, 0304h**	BB 04 03 **(3)**	Ecrit 03 04$_H$ dans le registre AX ; **(AX) = 03 04** (convention INTEL Little Endian)
01 06	**Boucle:**	**ADD AL, 1**	04 01 **(2)**	Ajoute 1 à l'octet de poids faible de AX noté AL : **(AL) = (AL) + 1**
01 08		**CMP AL, 2**	3C 02 **(2)**	Compare (AL) à 2 ; place le bit de Flag **Z** à **1** en cas d'**égalité** de la comparaison
01 0A		**JNE Boucle**	75 FA **(2)**	Saut à l'étiquette ***Boucle*** si le bit **Z = 0** (≡ s'il n'y a pas égalité) (*Jump Not Equal*)
01 0C		**PUSH AX**	50 **(1)**	Empile le contenu de AX dans la pile (STACK) : **(AX) → STACK**
01 0D		**PUSH BX**	53 **(1)**	Empile le contenu de AX dans la pile (STACK) : **(BX) → STACK**

Attention : ne pas introduire de caractère « espace » dans l'étiquette « Boucle: »

a) Compléter ce tableau lors de l'exécution complète et pas à pas du programme, en indiquant le contenu des registres spécifiés :

Instruction ASM	**IP**	**AX**	**BX**	**Flag Z**	**SP**	**STACK** (FFFF,**FFFE**,FFFD,**FFFC**,FFFB,**FFFA**)
Etat initial	**01 00**	**00 00**	**00 00**	**0**	**FF FE**	00 **00** 00 **00** 00 **00**
MOV AX, 0100h	**01 03**	**01 00**	00 00	0	FF FE	00 00 00 00 00 00
MOV BX, 0304h	**01 06**	01 00	**03 04**	0	FF FE	00 00 00 00 00 00
ADD AL, 1	**01 08**	**01 01**	03 04	0	FF FE	00 00 00 00 00 00
CMP AL, 2	**01 0A**	01 01	03 04	0	FF FE	00 00 00 00 00 00
JNE Boucle	**01 06**	01 01	03 04	0	FF FE	00 00 00 00 00 00
ADD AL, 1	**01 08**	**01 02**	03 04	0	FF FE	00 00 00 00 00 00
CMP AL, 2	**01 0A**	01 02	03 04	**1**	FF FE	00 00 00 00 00 00
JNE Boucle	**01 0C**	01 02	03 04	1	FF FE	00 00 00 00 00 00
PUSH AX	**01 0D**	01 02	03 04	1	**FF FC**	00 **00** 01 **02** 00 **00**
PUSH BX	**01 0E**	01 02	03 04	1	**FF FA**	00 **00** 01 **02** 03 **04**

Note : Les 2 1ers octets de la pile (sommet FFFF et FFFE) sont réservés. La mémorisation commence après.

TD-TP 3 ADO Corrigé. Assembleur 80x86

0. Calcul de N ! (N<4) avec Entrées/Sorties (E/S)

```
; factoriel N : N!  (0 < N < 4)

org 100h

    mov ah, 1  ; Lecture de N au clavier
    int 21h
    sub al, 30h ; N dans al + conversion ascii

    mov bl, al  ; (al) -> bl
    cmp bl, 1  ; test si N=1
    je fin
boucle: dec bl    ; N-1 -> bl
    mul bl    ;
    cmp bl, 1
fin:  jne boucle  ; Resultat dans al
    push ax    ; sauvegarde du resultat dans la pile

    mov ah, 0Eh ; Affichage de N! a l'ecran
    add al, 30h ; Conversion ascii
    int 10h

ret
```

1a.som_N_1ers_nbres_iteration_sans_E-S.asm

```
; Somme des N 1ers nombres par iteration sans E/S :   S = somme de i (de i = 1 a N)
; 0 < N < 100

org 100h

        mov al, 99          ; N dans al
        mov bh, 0
        mov bl, al          ; (ax) -> bx
boucle: dec al
        add bx, ax
        cmp al, 1
        jnz boucle          ; Resultat S dans bx

ret
```

1b.som_N_1ers_nbres_formule_sans_E-S.asm

```
; Somme des N 1ers nombres par formule de Gauss sans E/S :     S = N(N+1)/2
; 0 < N < 100

org 100h

        mov ax, 99          ; N dans ax
        mov bx, ax          ; (ax) -> bx
        inc bx
        mul bx              ; (ax)*(bx) -> ax
        shr ax, 1           ; Division de (ax) par 2 : Resultat S dans ax

ret
```

2a.som_N_1ers_nbres_iteration_avec_E-S.asm

```
; Somme des N 1ers nombres par iteration avec E/S :   S = somme de i (de i = 1 a N)
; 0 < N < 4

org 100h

        mov ah, 1           ; Lecture de N au clavier
        int 21h
        sub al, 30h         ; N dans al + conversion ascii

        mov bl, al          ; (al) -> bl
boucle: dec al
        add bl, al
        cmp al, 1
        jnz boucle          ; Resultat dans bl

        mov ah, 0Eh         ; Affichage de S a l'ecran
        mov al, bl
        add al, 30h         ; Conversion ascii
        int 10h

ret
```

2b.som_N_1ers_nbres_formule_avec_E-S.asm

```
; Somme des N 1ers nombres par formule de Gauss avec E/S :      S = N(N+1)/2
; 0 < N < 4

org 100h

        mov ah, 1           ; Lecture de N au clavier
        int 21h
        sub al, 30h         ; N dans al + conversion ascii

        mov bl, al          ; (al) -> bl
        inc bl              ; (bl) = (bl)+1
        mul bl              ; (al)*(bl) -> al
        shr al, 1           ; Division de (al) par 2 : Resultat dans al

        mov ah, 0Eh         ; Affichage de S a l'ecran
        add al, 30h         ; Conversion ascii
        int 10h

ret
```

3a.som_N_1ers_nbres_iteration_avec_E-S.asm

```
; Somme des N 1ers nombres par iteration avec E/S :   S = somme de i (de i = 1 a N)
; 0 < N < 10

org 100h

        mov ah, 1           ; Lecture de N au clavier
        int 21h
        sub al, 30h         ; N dans al + conversion ascii

        mov bl, al          ; (al) -> bl
boucle: dec al
        add bl, al
        cmp al, 1
        jnz boucle          ; Resultat S dans bl

        mov al, bl
        mov ah, 0
        mov dl, 10
        div dl              ; (1er digit de S - poids fort) dans al
        mov cl, ah          ; Sauvegarde de S (2nd digit) dans cl

        mov ah, 0Eh         ; Affichage de S (1er digit) a l'ecran
        add al, 30h         ; Conversion ascii
        int 10h

        mov al, cl          ; Restitution de S (2nd digit) dans al

        mov ah, 0Eh         ; Affichage de S (2nd digit) a l'ecran
        add al, 30h         ; Conversion ascii
        int 10h

ret
```

4a.som_N_1ers_carres_iteration_sans_E-S.asm

```
; Somme des N 1ers nombres par iteration sans E/S :   S2 = somme de (i*i) (de i = 1 a N)
; 0 < N < 10

org 100h

        mov ax, 9           ; N dans al
        mov bx, 0           ; resultat dans bx
boucle: push ax             ; sauvegarde de i
        mul ax              ; i*i -> ax
        add bx, ax          ; (ax)+(bx) -> bx
        pop ax              ; restitution de i
        dec ax
        cmp ax, 0
        jnz boucle          ; resultat S2 dans bx

ret
```

4b.som_N_1ers_carres_formule_sans_E-S.asm

```
; Somme des N 1ers nombres par formule sans E/S :     S2 = N(N+1)(2N+1)/6
; 0 < N < 9

org 100h

        mov ax, 8           ; N dans ax
        mov bx, ax          ; (ax) -> bx
        inc bx              ; N+1 -> bx

        push ax             ; sauvegarde de ax
        mov dl, 2
        mul dl              ; 2N -> ax
        inc ax              ; 2N+1 -> ax
        mov cx, ax          ; 2N+1 -> cx

        pop ax              ; N -> ax
        mul bx              ; N(N+1) -> ax
        mul cx              ; N(N+1)(2N+1) -> ax

        mov dl, 6
        div dl              ; division de (ax) par 6 : Resultat S2 dans ax

ret
```

TD 4 CORRIGE. Les Processeurs actuels. Pipelining

1. Pipelining

L'accès à l'opérande d'un LOAD/STORE - à cause des accès cache - coûte 2 cycles, le résultat du LOAD/STORE est ensuite disponible via le mécanisme de bypass, sans attendre l'écriture registre du résultat.

1.
2 bulles sont insérées car l'instruction LOAD nécessite les 2 cycles suivants. Explication : appelons X l'instruction après le LOAD. L'instruction X reste bloquée à l'étage 3 tant que l'accès cache du LOAD n'est pas terminé, c'est-à-dire tant que le LOAD n'a pas dépassé l'étage 6.

Juste avant que le LOAD rentre dans l'étage 7, le résultat du LOAD est envoyé sur le réseau de *bypass* et l'instruction X va pouvoir rentrer dans l'étage 4. A cet instant précis, les étages 4 et 5 ne contiennent aucune instruction. 2 bulles ont donc été introduites.

2.
La boucle itère un grand nombre de fois parce que la valeur intiale de R4 est grande. Donc on peut raisonner comme si la boucle itérait un nombre infini de fois et négliger le temps de remplissage du pipeline. Le corps de la boucle comporte 5 instructions. La 2ème instruction dépend de la 1ère, qui est un LOAD. 2 bulles sont donc générées toutes les 5 instructions.

Le débit d'exécution vaut : $\frac{5}{5+2} = \frac{5}{7} \approx 0.71$ instructions / cycle.

Définition du débit : *débit* = nombre d'instructions / (nombre d'instructions + nombre de bulles).

3.
Dans le programme initial, l'instruction 2 qui suit immédiatement un LOAD, utilise le résultat du LOAD, ce qui introduit 2 bulles dans le pipeline, bulles qu'il faut supprimer pour ramener le débit d'exécution à 1 instruction / cycle.

Pour supprimer les 2 bulles, il faut placer 2 instructions indépendantes du LOAD juste après le LOAD, comme ceci :

```
1:      boucle:  R2 = LOAD R1+0
2:               R1 = R1 ADD 4
3:               R4 = R4 SUB 1
4:               R3 = R3 ADD R2
5:               BNZ R4, boucle
```

4.
a) Pour supprimer les 2 bulles introduites par un LOAD, il faut faire suivre ce LOAD de 2 instructions indépendantes du résultat du LOAD. Ici, le LOAD n'est suivi que d'1 seule instruction indépendante du résultat du LOAD, ce qui introduit 1 *bulle* dans le pipeline.

Le débit d'exécution du programme, comportant 4 instructions et coûtant 5 cycles, est donc de :

$\frac{4}{4+1} = \frac{4}{5}$ instructions / cycle.

b) Pour comparer 2 programmes différents effectuant le même travail et n'ayant pas le même nombre d'instructions, il ne faut pas regarder le débit en instructions par cycle mais le temps total pour effectuer le travail.

Ici, on peut se contenter de comparer le nombre de cycles par itération, puisque chaque itération traite un élément du tableau.

Avec le programme de la question **2**, chaque itération coûte 7 cycles par boucle.

Le nouveau programme ne comporte que 4 instructions au lieu de 5. 1 bulle est générée à chaque itération. Chaque itération coûte donc 5 cycles. Le nouveau programme est plus performant que celui de la question **2** et aussi performant que celui de la question **3** qui compte 5 cycles aussi mais sans *bulle*.

5.

Rappel :

étape	*pipeline P1*
1	**lit cache d'instructions**
2	**decode instruction**
3	**lit registres**
4	**execute / calcul adresse**
5	**acces cache de donnees**
6	**acces cache de donnees**
7	**ecrit registre**

étape	*pipeline P2*
1	**lit cache d'instructions**
2	**decode instruction**
3	**lit registres**
4	**calcul adresse**
5	**acces cache de donnees**
6	**execute / acces cache de donnees**
7	**ecrit registre**

. Pipeline initial (P1) : 2 *bulles* sont introduites si le résultat de l'exécution d'une instruction LOAD/STORE dépend de l'instruction précédente.

. Nouveau Pipeline (P2) : 2 *bulles* sont introduites avec le calcul d'adresse pour chaque opérande d'une instruction LOAD/STORE dépendant de l'instruction précédente.

Si les 2 instructions précédant un LOAD/STORE sont indépendantes du calcul d'adresse de ou des opérandes de l'instruction LOAD/STORE, aucune *bulle* n'est introduite.

Pipeline P1 : 2 bulles sont introduites dans le pipeline si l'instruction *immédiatement après* un LOAD/STORE *utilise* comme *opérande* le *résultat* du LOAD/STORE.

Pipeline P2 : 2 bulles sont insérées dans le pipeline si l'instruction *immédiatement avant* un LOAD/STORE *met à jour* un *opérande* du LOAD/STORE.

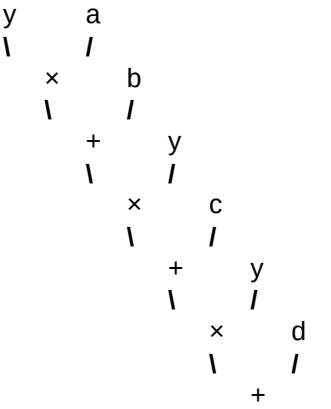
Dans le cas du programme de la question **4**, le calcul d'adresse du LOAD dépend de R1 et de R4. Seul R4 est mis à jour dans la boucle par l'instruction SUB. Comme il y a 2 instructions intercalées entre le SUB et le LOAD, aucune *bulle* n'est introduite ici. Chaque itération coûte 4 cycles sur le nouveau pipeline (P2), au lieu de 5 cycles sur l'ancien pipeline (P1).

TP 4 CORRIGE. Les Processeurs actuels. Pipelining

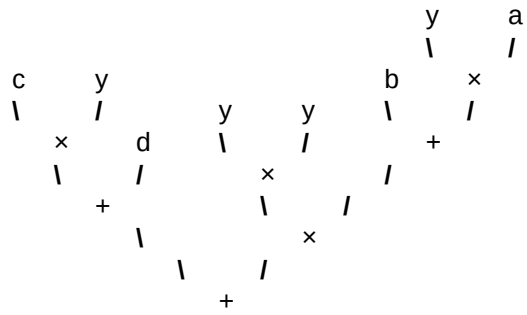
1. Pipelining (1)

Une remarque importante est que le compilateur doit effectuer le calcul exactement comme spécifié par le programme car l'addition et la multiplication en virgule flottante ne sont pas des opérations associatives.

On peut représenter le calcul par un arbre de dépendance, comme représenté ci-dessous :



Méthode 1 (de Horner) : d + y(c + y(b + ya))



Méthode 2 : (d + cy) + ((y × y) × (ay + b))

a) Pseudo-codes assembleur

Rappel : Initialisation :

F0 = y
F1 = a
F2 = b
F3 = c
F4 = d

Méthode 1 (de Horner) : d + y(c + y(b + ya))

Le programme suivant implémente la méthode 1 (de Horner) :

```
1 :     F5 = F1 FMUL F0          // F5 = ay
2 :     F5 = F5 FADD F2          // F5 = ay + b
3 :     F5 = F5 FMUL F0          // F5 = (ay + b)y
4 :     F5 = F5 FADD F3          // F5 = (ay + b)y + c
5 :     F5 = F5 FMUL F0          // F5 = [(ay + b)y + c]y
6 :     F5 = F5 FADD F4          // F5 = [(ay + b)y + c]y + d
```

Méthode 2 : (d + cy) + ((y × y) × (ay + b))

La méthode 2 peut être implémentée avec le programme suivant :

```
1 :     F5 = F0 FMUL F1          // F5 = ya
2 :     F6 = F0 FMUL F3          // F6 = yc
3 :     F7 = F0 FMUL F0          // F7 = y^2
4 :     F5 = F5 FADD F2          // F5 = ya + b
5 :     F6 = F6 FADD F4          // F6 = yc + d
6 :     F5 = F5 FMUL F7          // F5 = (ya + b)y^2
7 :     F5 = F5 FADD F6          // F5 = (ya + b)y^2 + yc + d
```

b) Performances

Méthode 1 (de Horner) :

Chacune des 5 instructions (les instructions 2 à 6) dépend de l'instruction précédente. Ce programme va occuper l'étage de décodage (**decode instruction**) pendant :

$6 + 5 \times 2 = 16$ cycles [= 6 instructions + (10=5x2) bulles = 16 cycles].

Méthode 2 :

Les instructions 1, 2, 3 sont indépendantes.

L'instruction 4 dépend de l'instruction 1 mais la pénalité de 2 cycles est couverte par les instructions 2 et 3.

De même, la dépendance entre les instructions 2 et 5 est couverte par les instructions 3 et 4.

La dépendance entre les instructions 4 et 6 n'est que partiellement couverte par l'instruction 5, donc on paye 1 cycle de pénalité.

Quant à la dépendance entre les instructions 6 et 7, elle n'est pas couverte, on paye donc 2 cycles de pénalité.

Au total, le programme occupe l'étage de décodage (**decode instruction**) pendant :

$7 + 2 + 1 = 10$ cycles [= 7 instructions + (3=2+1) bulles = 10 cycles].

La méthode 2 est donc plus performante que la méthode 1 (de Horner).

2. Pipelining (2)

1.

Dans un processeur non pipeliné, chaque instruction doit entiérement être exécutée avant que l'exécution de la suivante ne puisse commencer.

Dans un processeur pipeliné, l'exécution de l'instruction est décomposée en plusieurs étapes correspondant aux étages du pipeline.

Dès que le premier étage du pipeline est franchi par une instruction, l'instruction suivante peut entamer son exécution en y accédant a son tour pendant que l'instruction précédente continue son exécution. Ceci augmente la vitesse d'exécution.

Le pipelining devise le chemin de données d'un processeur en étages séparés par des latches (registres).

Dans un processeur non pipeliné, une instruction doit être capable de parcourir la totalité du chemin de données en un seul cycle d'horloge.

Dans un processeur pipeliné, l'instruction doit simplement pouvoir passer à l'étage suivant à chaque cycle, ce qui permet d'avoir un cycle d'horloge beaucoup plus court.

2.

Il existe 2 limites principales.

La 1ère tient au fait que, à mesure que le nombre d'étages du pipeline augmente, la fraction de temps que représente la latence (temps de traitement) du latch de chaque étage devient plus importante - en valeur relative.

La 2nde limite provient des dépendances des données et des délais de branchement.

Les instructions qui dépendent des résultats d'autres instructions doivent attendre que ces derniéres aient terminé leur exécution et provoquent dans ce cas des blocages dans le pipeline (*bulles*).

Par ailleurs, les instructions qui suivent les branchements doivent attendre que ces branchements se terminent pour pouvoir être insérées dans le pipeline. Ces différents retards conduisent le pipeline à exécuter moins d'1 instruction par cycle en moyenne, ce qui implique qu'une plus grande partie du temps processeur sera consacré à attendre les déblocages du pipeline.

3.1. L'étage le plus long posséde une latence de 7 ns. En ajoutant le délai de 1 ns pour le latch de pipeline (latch de chaque étage de pipeline), on obtient un temps de cycle de 8 ns (si on a des étages de différentes latences, on prend toujours la latence de l'étage le plus long).

3.2. Puisqu'il y a 5 étages, la latence totale du pipeline est de : 8 ns * 5 étages = 40 ns.

Note :

Le temps d'exécution *réel* (en nombre de cycles) d'une séquence d'instructions d'un pipeline sans délais de branchement ni dépendance de données est :

nombre d'étages + nombre d'instructions – 1.