ELECTRONIQUE

NUMERIQUE

CORRIGES

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

→ $S = \bar{P}_L + P_P$

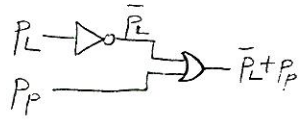
si on regroupe les 0 : (→ $\bar{S}$ obtenue)

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.

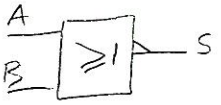
P_P \ P_L	0	1
0	①	
1		②

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

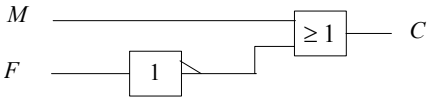
A, B, $\bar{A}$, $\bar{B}$, $\overline{\bar{A}\,\bar{B}} = A+B$, $\overline{A+B}$

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

C / F \ 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 : $s = \overline{a}bc + a\overline{b}c + ab\overline{c} + abc$

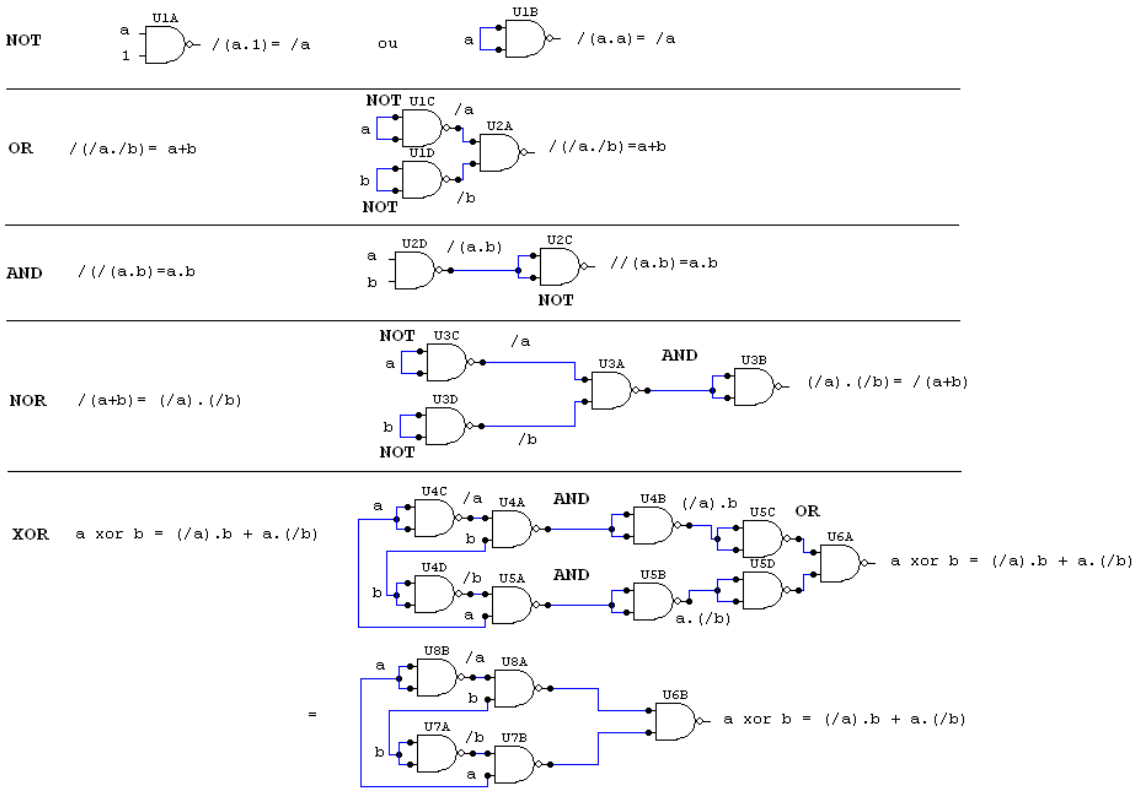
Simplification de s :

- algébrique : $s = c\left(\overline{a}b + a\overline{b}\right) + abc = c\left(a \oplus b\right) + abc$

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

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

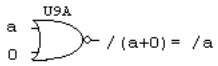
$\boxed{S = B \oplus C}$

TP 1 CORRIGE. LOGIQUE COMBINATOIRE 1

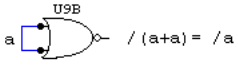
4. Etude Expérimentale

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

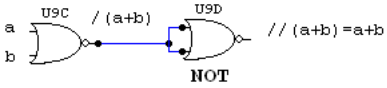
NOT



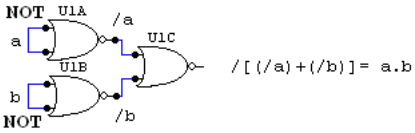
ou



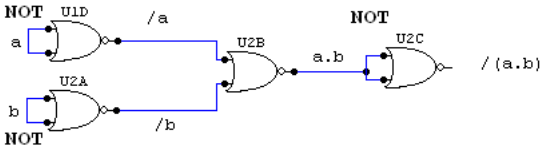
OR /(/(a+b) = a+b



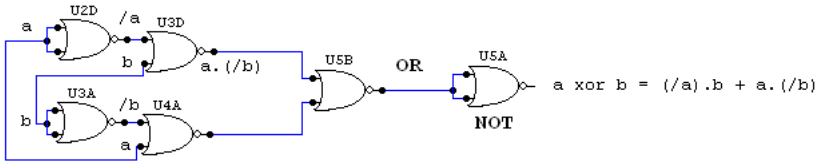
AND /[(/a)+(/b)] = a.b



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



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



Corrigé TD d'électronique n°2

Logique combinatoire (2)

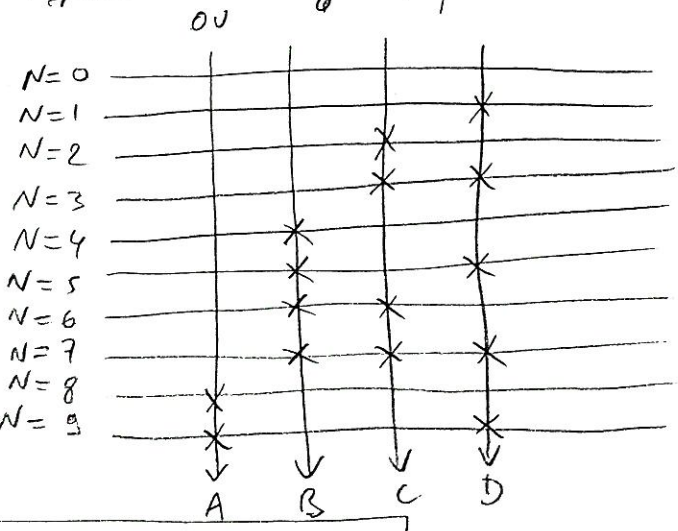
TD2 — Logique Combin.2

1

Table de codage

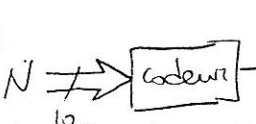
N	ABCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

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

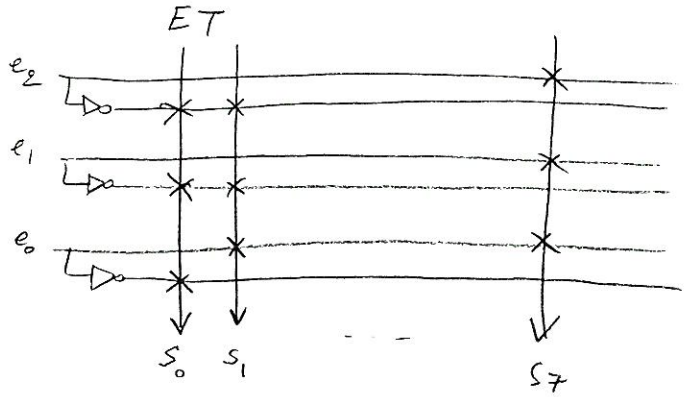
1 seule sortie active à la fois ≡ Décodeur ≡ Matrice ET

Table de Décodage

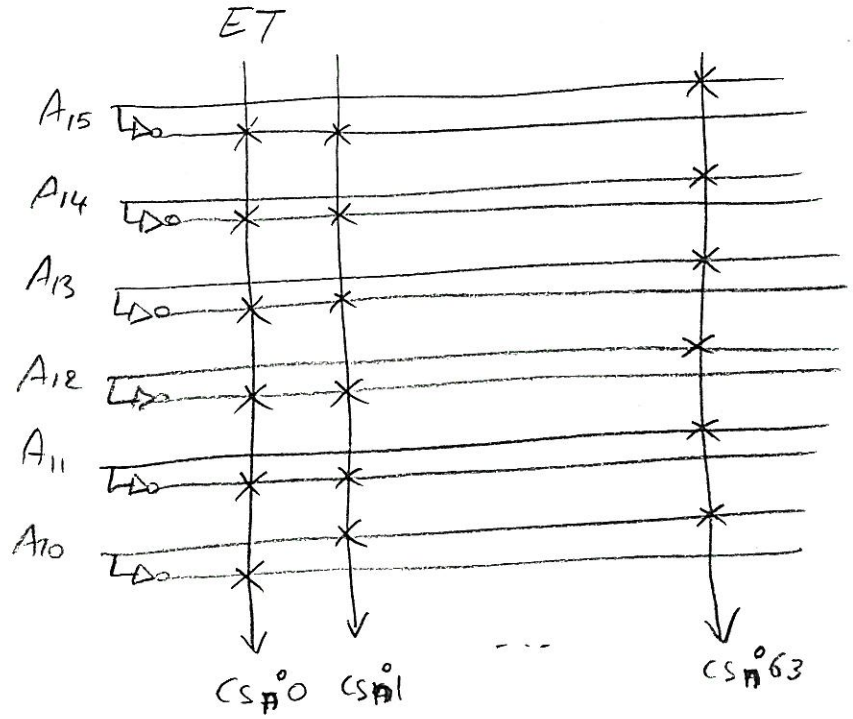
e_2 e_1 e_0	S_i
0 0 0	$S_0 = 1$ ($S_i = 0$ de $i = 1$ à 7)
0 0 1	$S_1 = 1$ ($S_0 = 0$; $S_i = 0$ de $i = 2$ à 7)
0 1 0	$S_2 = 1$ ($S_0 = S_1 = 0$; $S_i = 0$ de $i = 3$ à 7)
0 1 1	$S_3 = 1$
1 0 0	$S_4 = 1$
1 0 1	$S_5 = 1$
1 1 0	$S_6 = 1$
1 1 1	$S_7 = 1$

etc...

Représentation symbolique



4



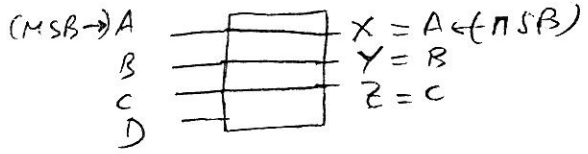
5

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

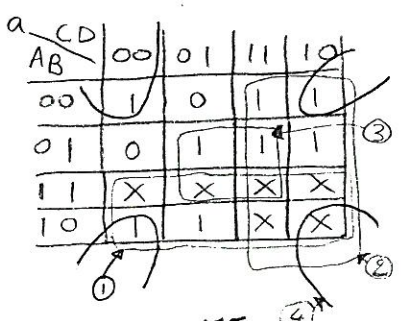
a) idem que ~~l'ex.~~ 7. mais les 6 dernières lignes sont indéfinies (X) pour a b c d e f g

	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).

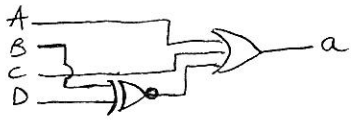
N	N(BCD) A	B	C	D	N(7 segments) a	b	c	d	e	f	g
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1
A	1	0	1	0	1	1	1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

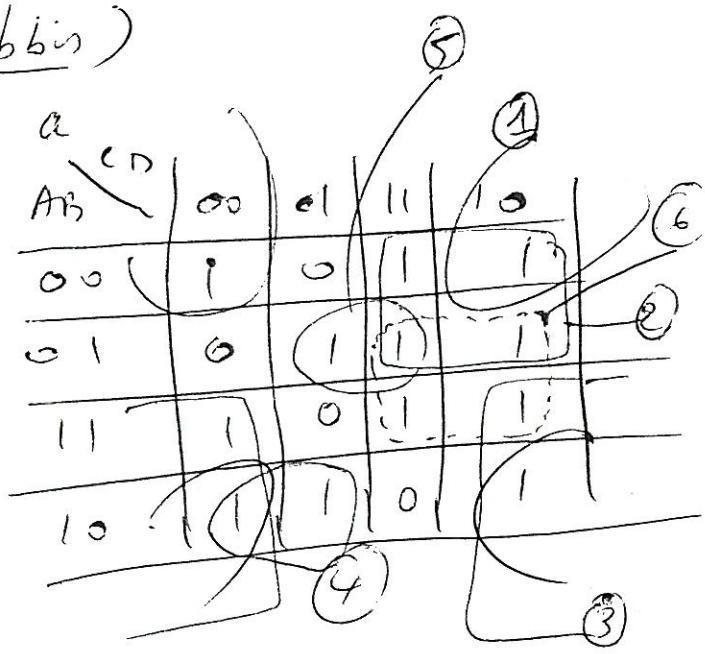


$a = A + C + BD + \overline{B}\,\overline{D} = A + C + \overline{B \oplus D}$

① ② ③ ④



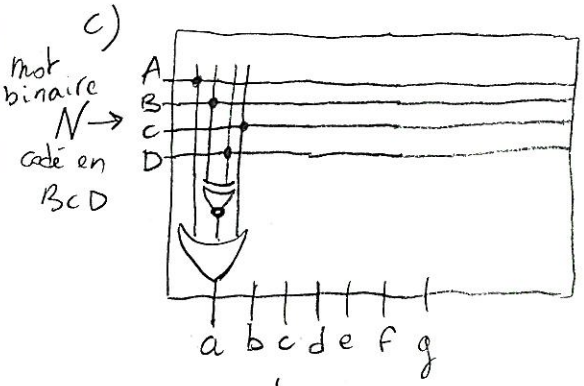
bbis)

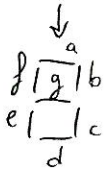


$a = \overline{B}\,\overline{D} + \overline{A}C + A\overline{D} + A\overline{B}\overline{C} + \overline{A}BD + BC$

0 ① ② ③ ④ ⑤ ⑥

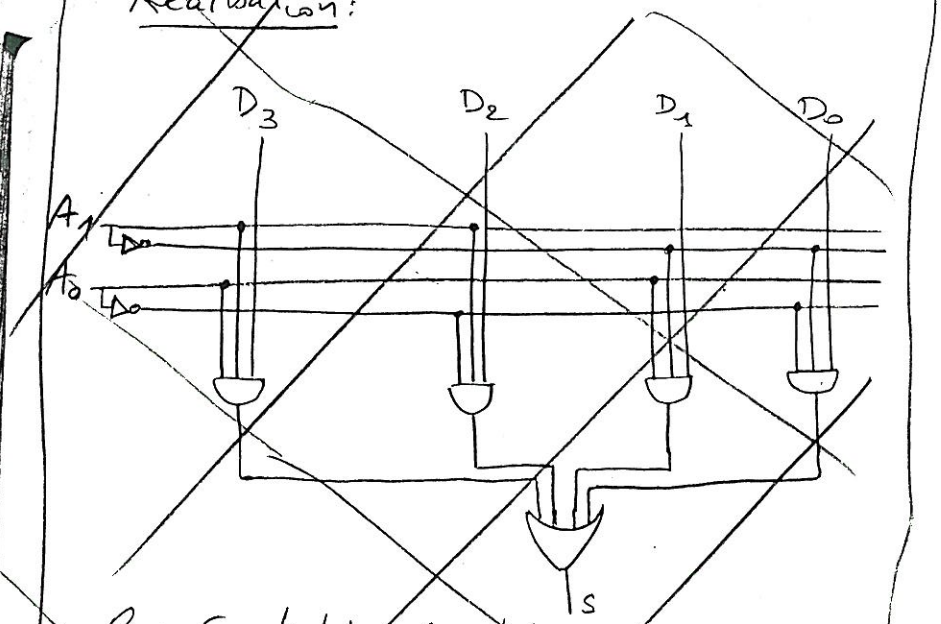
c)



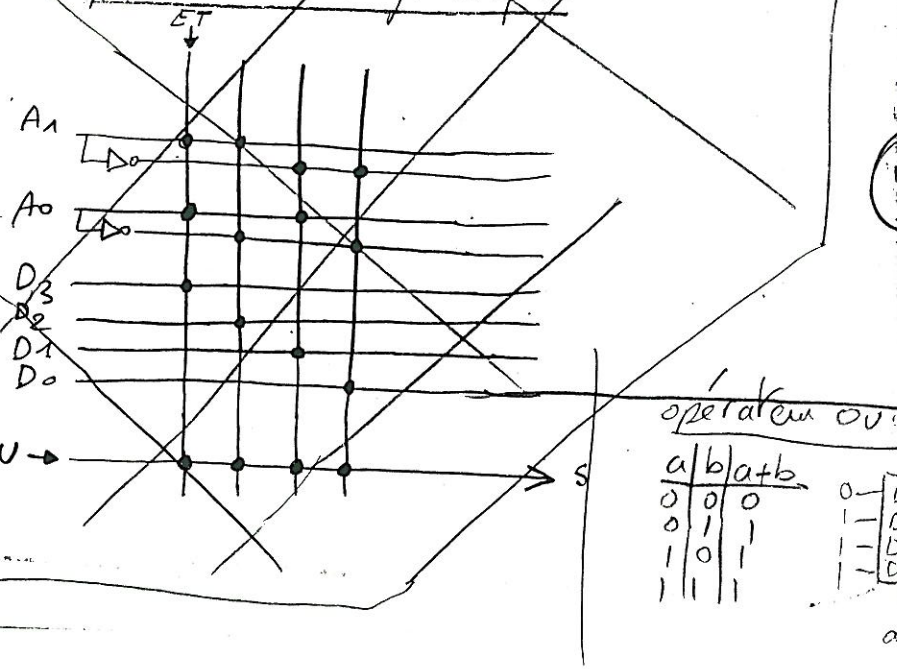


(7)

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 :

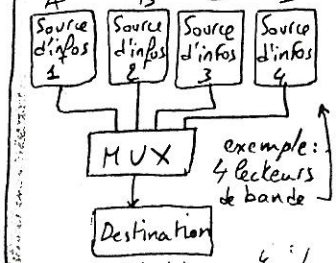
Sélection d'un mot binaire parmi plusieurs mots binaires

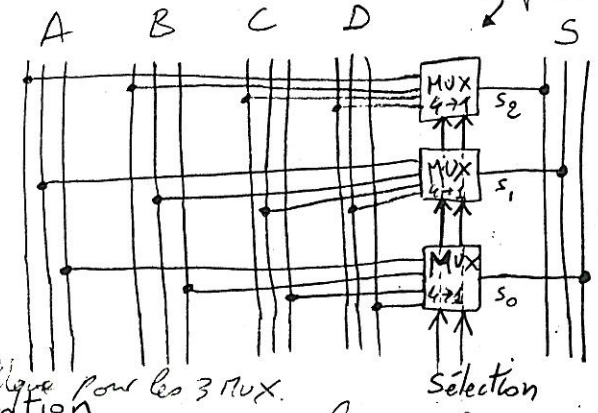
ex : sélection d'un mot de 3 bits parmi 4 mots de 3 bits :

⟹ 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

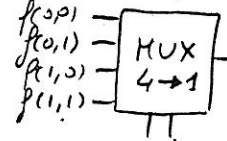
Matérialisation d'une fonction logique :

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 $b = j$

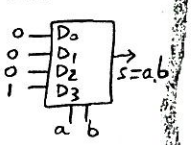
→ utilisation d'un multiplexeur à 4 entrées de données, donc 2 entrées d'adresse :



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 $2^n \to 1$

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

(10)

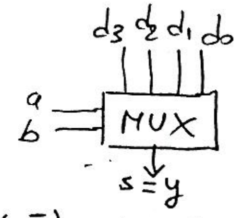
On a :

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

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

a b

noté a_1 (MSB) — noté a_0 (LSB)



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

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

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

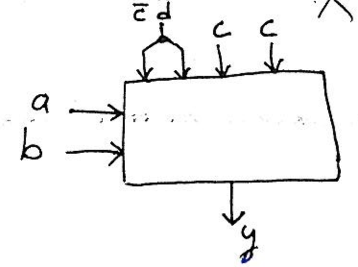
— d_1 : ———— : $cd + c\bar{d} = c$

— d_2 : ———— : $\bar{c}\bar{d}$

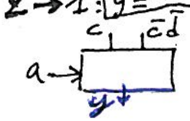
— d_3 ———— : $\bar{c}\bar{d}$

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



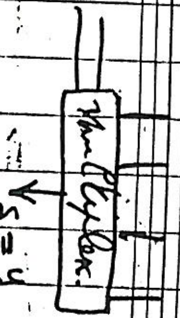
On pourrait toujours faire comme dans l'exercice précédent (6) utiliser 4 variables pour les adresses du MUX (le MUX est alors un MUX 16→1 avec en entrées de données des cst des 1, mais pour réduire la taille du MUX, on peut entrer des données variables)

Soit un pb. donné : voici sa table de Karnaugh →

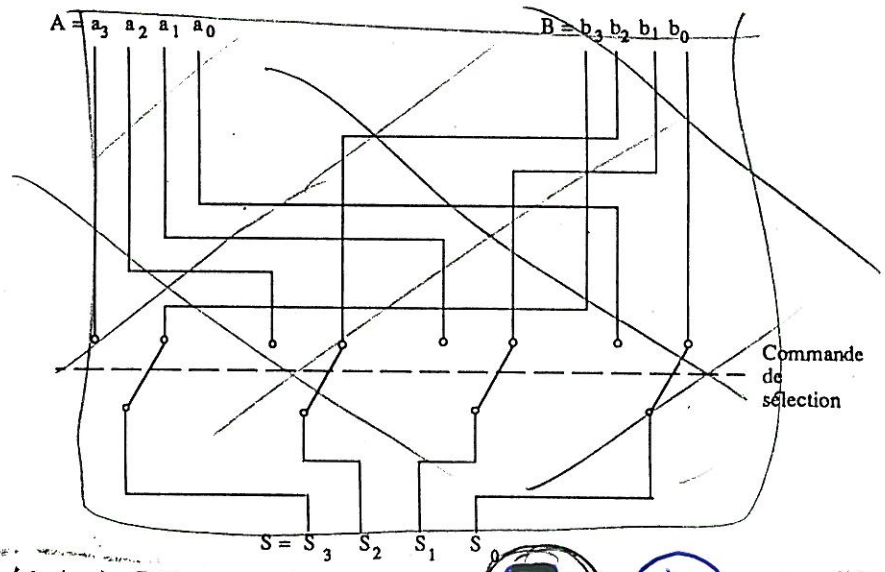
(y) cd \ ab

	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.



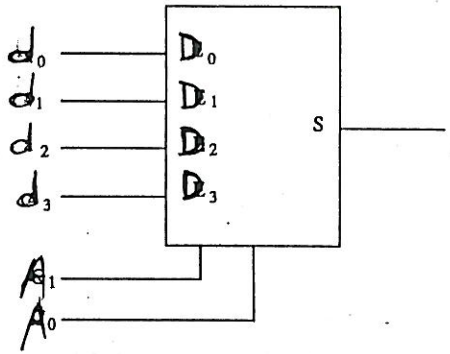
$y = c\bar{a} + \bar{c}\bar{d}a$



Conversion parallèle-série

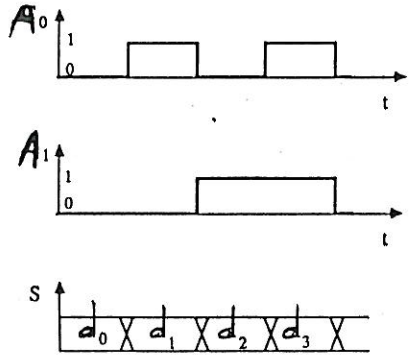
Soit un mot binaire $D = d_3 d_2 d_1 d_0$ disponible en mode parallèle, c'est-à-dire sur quatre fils, chaque fil étant affecté à un élément binaire du mot.

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

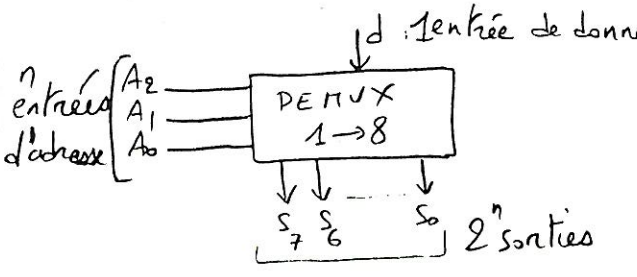
$$S = \overline{C_1}\,\overline{C_0} E_0 + \overline{C_1} C_0 E_1 + C_1 \overline{C_0} E_2 + C_1 C_0 E_3$$

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 :



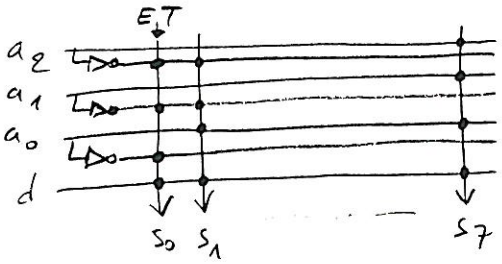
$A_2 A_1 A_0$	S_i
0 0 0	$S_0 = d$
0 0 1	$S_1 = d$
1 1 1	$S_7 = d$

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

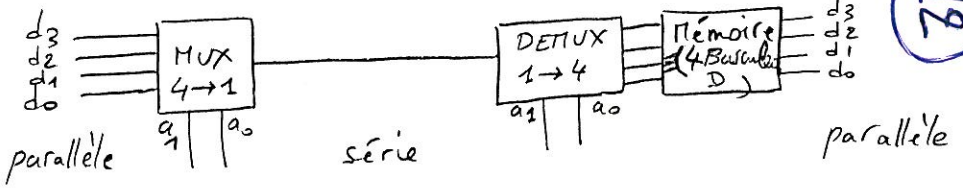
ex : $S_3 = \bar{a}_2 a_1 a_0 . d$

12.4.5.3 Réalisation : (Représentation Symbolique)



12.4.5.4 Applications

12.4.5.4.1 Conversion série / parallèle

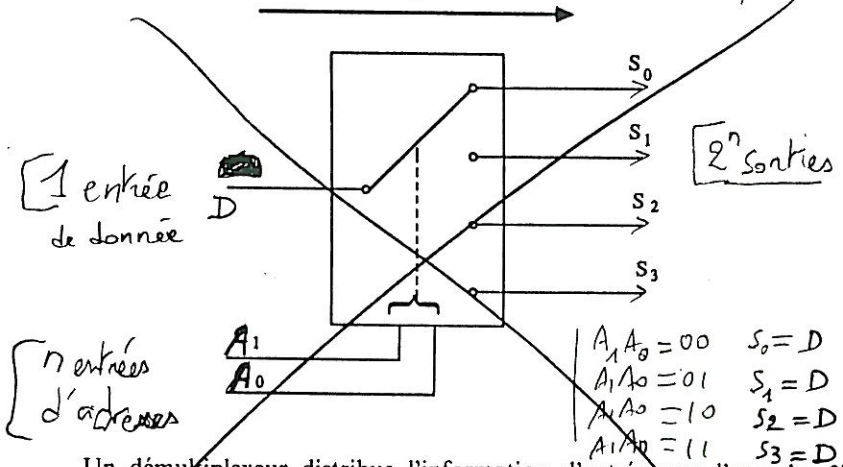


12.4.5.4.2 Affichage multiplexé

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

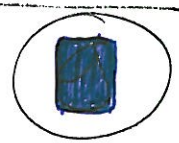
Le démultiplexeur aiguille 1 Donnée sur 1 parmi 2^n sorties



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

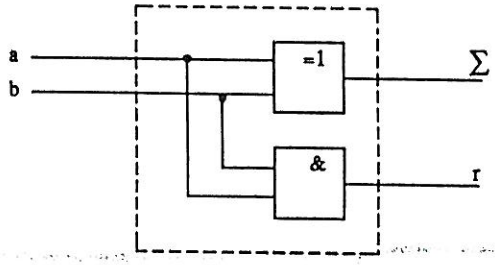
$\Sigma = a \oplus b$

— et la retenue r :

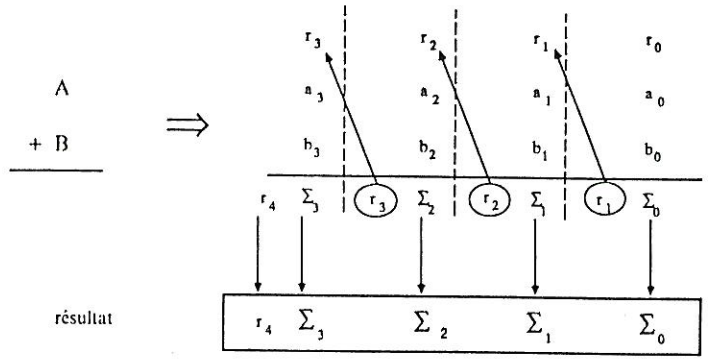
a \ b	0	1
0	0	0
1	0	1

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.

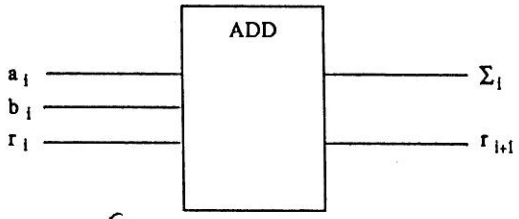


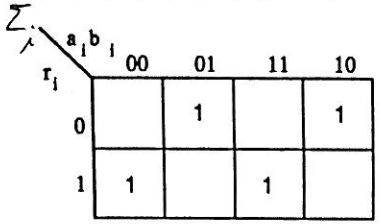
La structure de l'additionneur de deux nombres est alors répétitive. Une cellule élémentaire peut donc être utilisée pour chaque poids. Elle est appelée **additionneur complet.** L'addition globale est réalisée par la mise en cascade des cellules au sens des retenues.

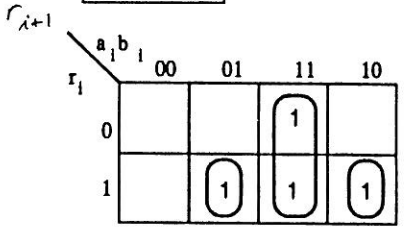
L'additionneur complet est défini par la table de vérité ci-après :

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



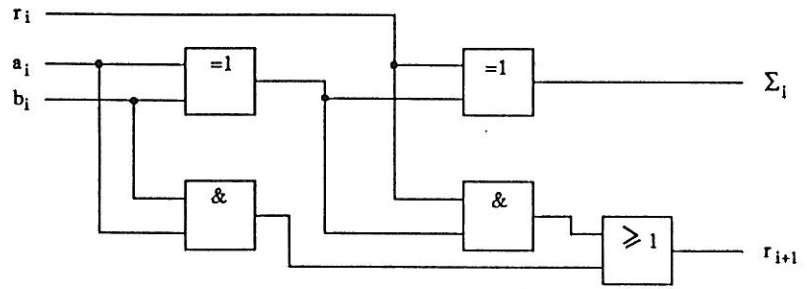




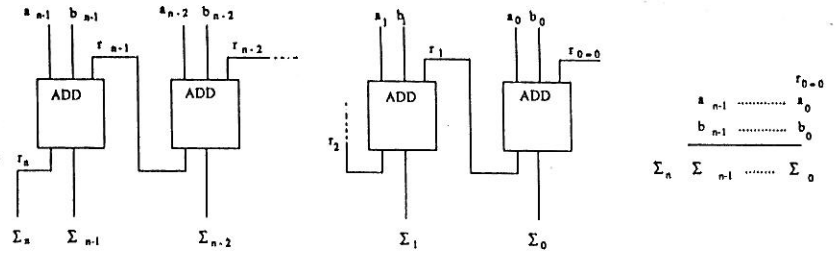
$\Sigma_i = a_i \oplus b_i \oplus r_i$

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

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.

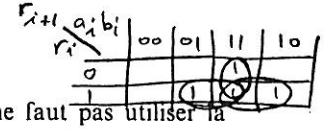


Cette solution est intéressante d'un point de vue du matériel parce que répétitive. Par contre, comme le résultat d'une addition ne peut pas être obtenu instantanément, le temps maximum mis pour obtenir le résultat est directement proportionnel au nombre d'additionneurs. En effet, après le premier temps 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** ».

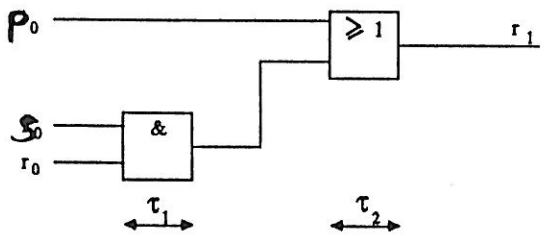
En reprenant le tableau de Karnaugh relatif au calcul de la retenue il vient :

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



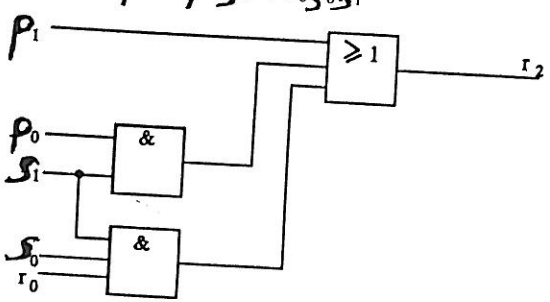
Afin d'éviter des temps de calcul cumulatifs, il ne faut pas utiliser la relation en tant que relation de récurrence, c'est-à-dire qu'il ne faut pas utiliser un résultat de calcul pour le calcul suivant. Il faut systématiquement recalculer chaque terme, ce qui donne en posant $\Sigma_i = a_i + b_i$ et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

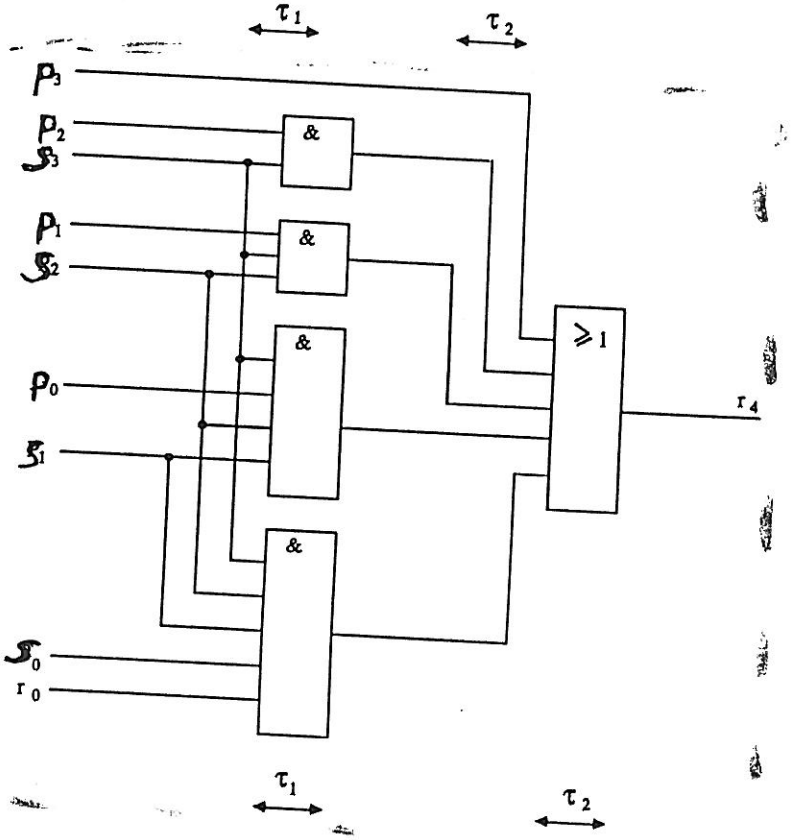
$$r_1 = P_0 + r_0 \Sigma_0$$



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 :

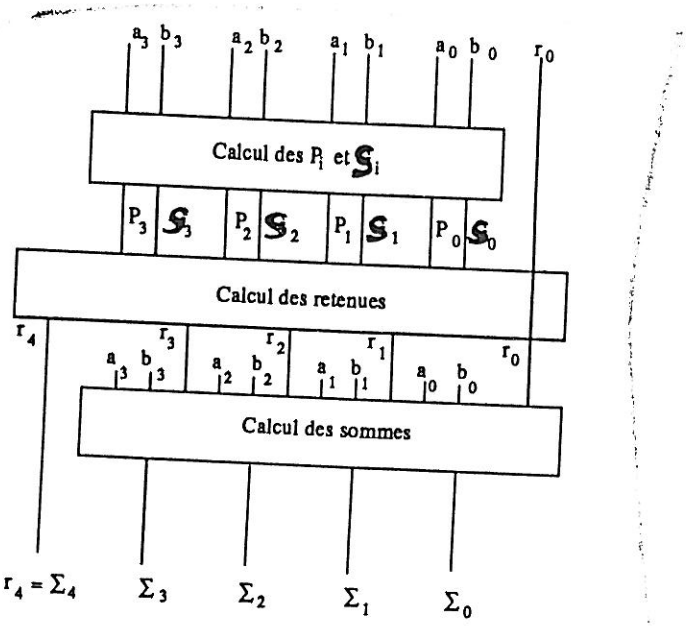
$$r_3 = p_2 + r_2 s_2 = p_2 + (p_1 + p_0 s_1 + r_0 s_0 s_1) s_2$$
$$= p_2 + p_1 s_2 + r_0 s_0 s_1 s_2$$

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.

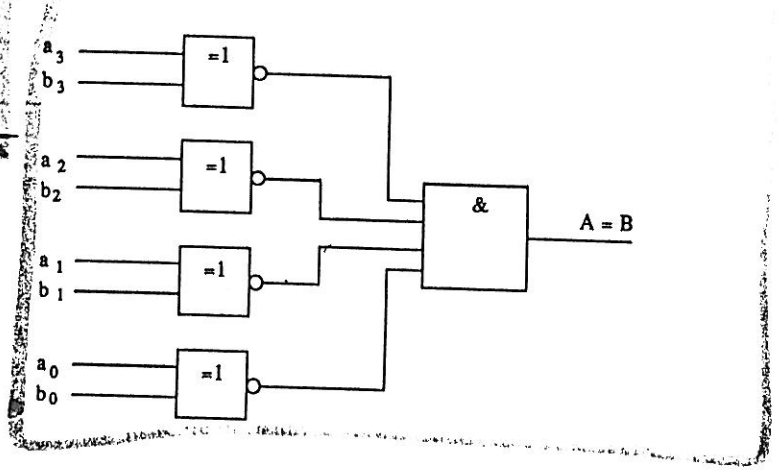
3.1.1. Comparateur donnant l'égalité de deux nombres

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

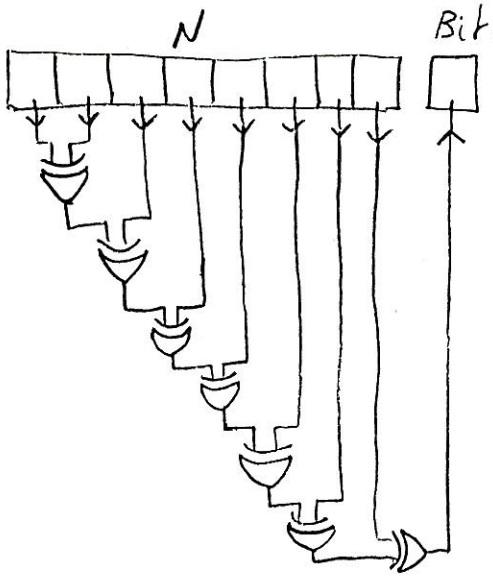
Par analogie avec l'additionneur, la conception d'un comparateur complet pour des nombres de quatre éléments binaires peut se faire de deux façons différentes.

— **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

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

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

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

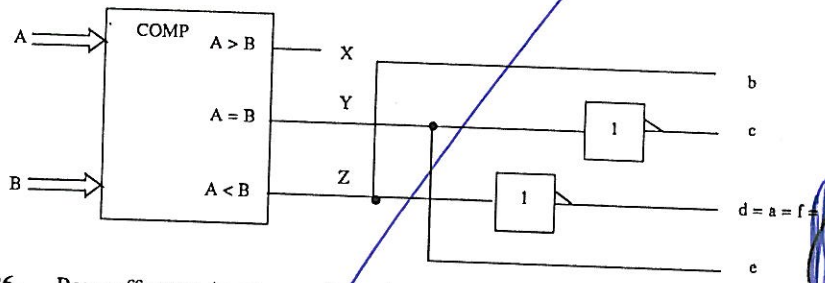
$$\boxed{k_1 = \overline{m_0 \oplus m_1 \oplus m_2}}$$ ($\oplus$ est associatif)

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

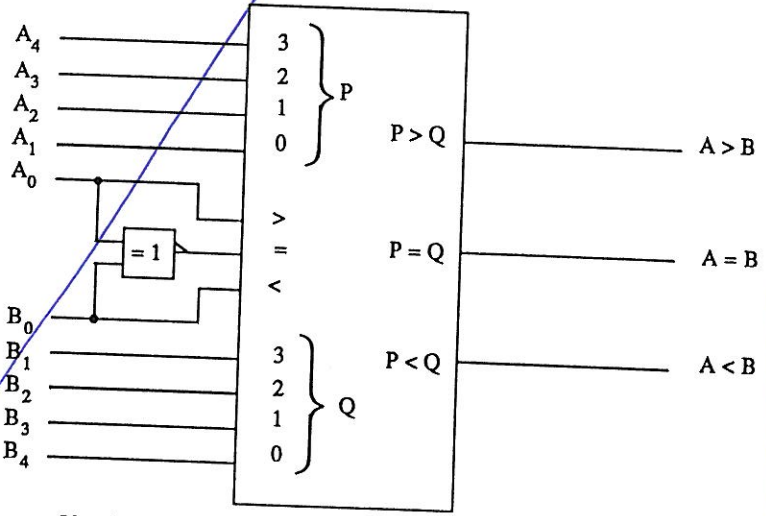
	X $(A > B)$	Y $(A = B)$	Z $(A < B)$	a	b	c	d	e	f	g
S	1	0	0	1	0	1	1	0	1	1
E	0	1	0	1	0	0	1	1	1	1
I	0	0	1	0	1	1	0	0	0	0

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.



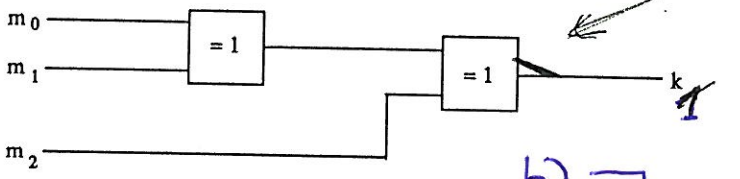
Un circuit de détection d'égalité doit néanmoins être rajouté pour valider l'égalité globale.

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	

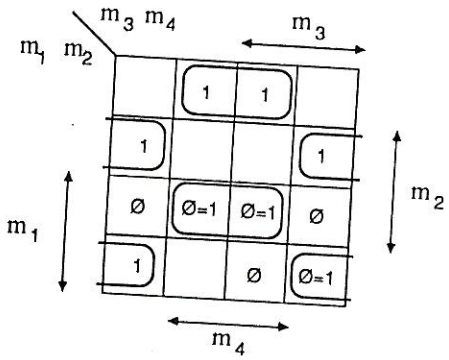
$$k_1 = m_1\overline{m}_2\overline{m}_4 + m_1m_2m_4 + \overline{m}_1\overline{m}_2m_4 + \overline{m}_1m_2\overline{m}_4$$
$$= m_1(\overline{m}_2\overline{m}_4 + m_2m_4) + \overline{m}_1(\overline{m}_2m_4 + m_2\overline{m}_4)$$
$$= m_1(\overline{m_2 \oplus m_4}) + \overline{m}_1(m_2 \oplus m_4)$$
$$=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

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.



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

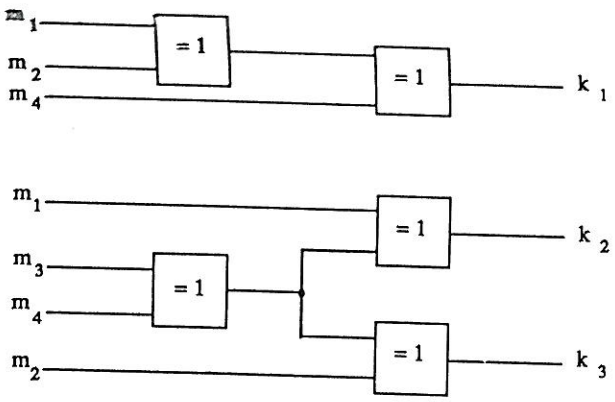
k_2 résulte du test de parité sur les éléments binaires 3. 6. 7. d'où :

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

De même

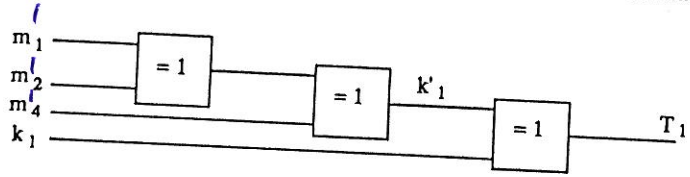
$$k_3 = m_2 \oplus m_3 \oplus m_4$$

Le schéma



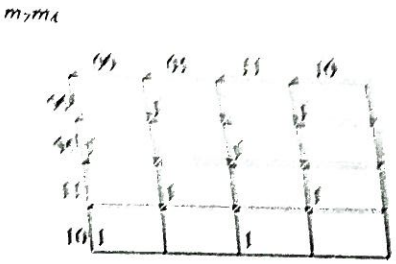
2°) Il y a deux solutions :

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

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

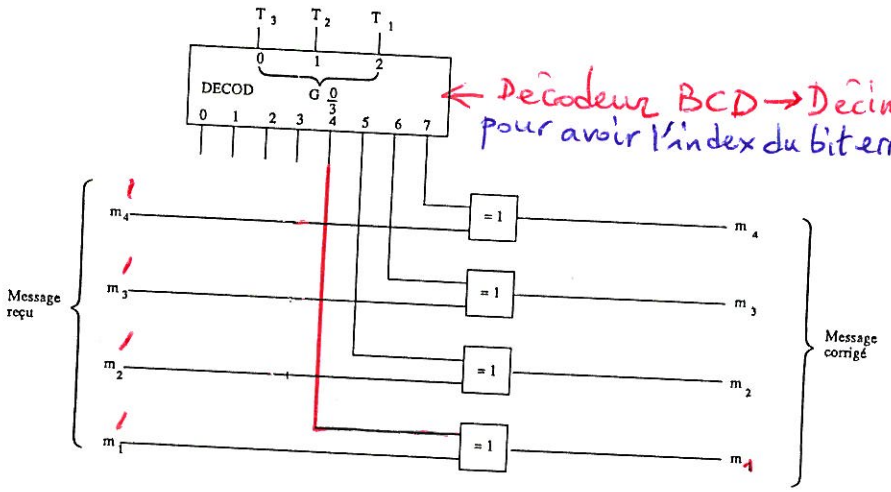
et

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

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

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) $d = \overline{m'_2 \oplus m'_1 \oplus m'_0 \oplus k_1}$

_____

TP2. LOGIQUE COMBINATOIRE 2

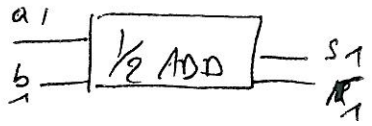
Logique Combin. 2

Corrigé Log 2. [redacted]

[redacted]

3.1. Les additionneurs

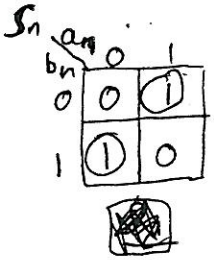
a) additionneurs 2 bits

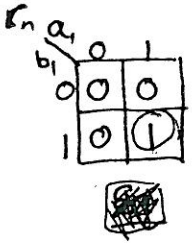


a_n	b_n	S_n	R_n
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

r_n [redacted] $= a_n \cdot b_n$

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

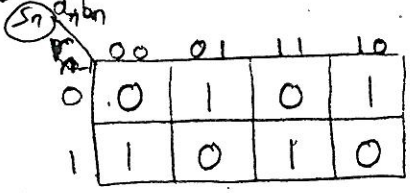




[redacted]

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

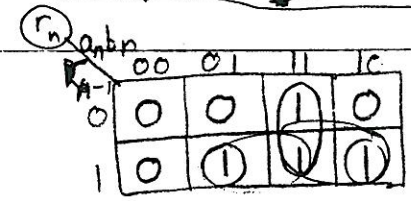


[redacted]

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

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

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



[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							

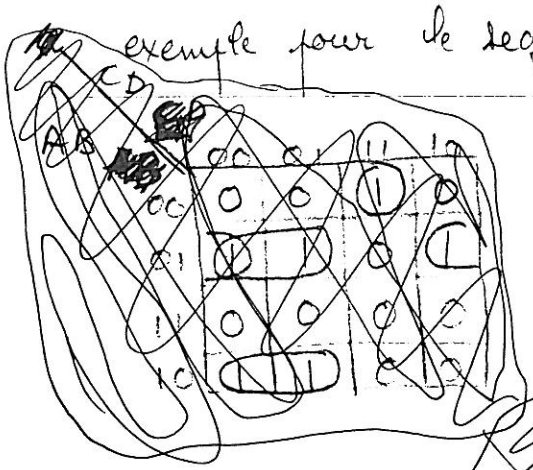
X

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

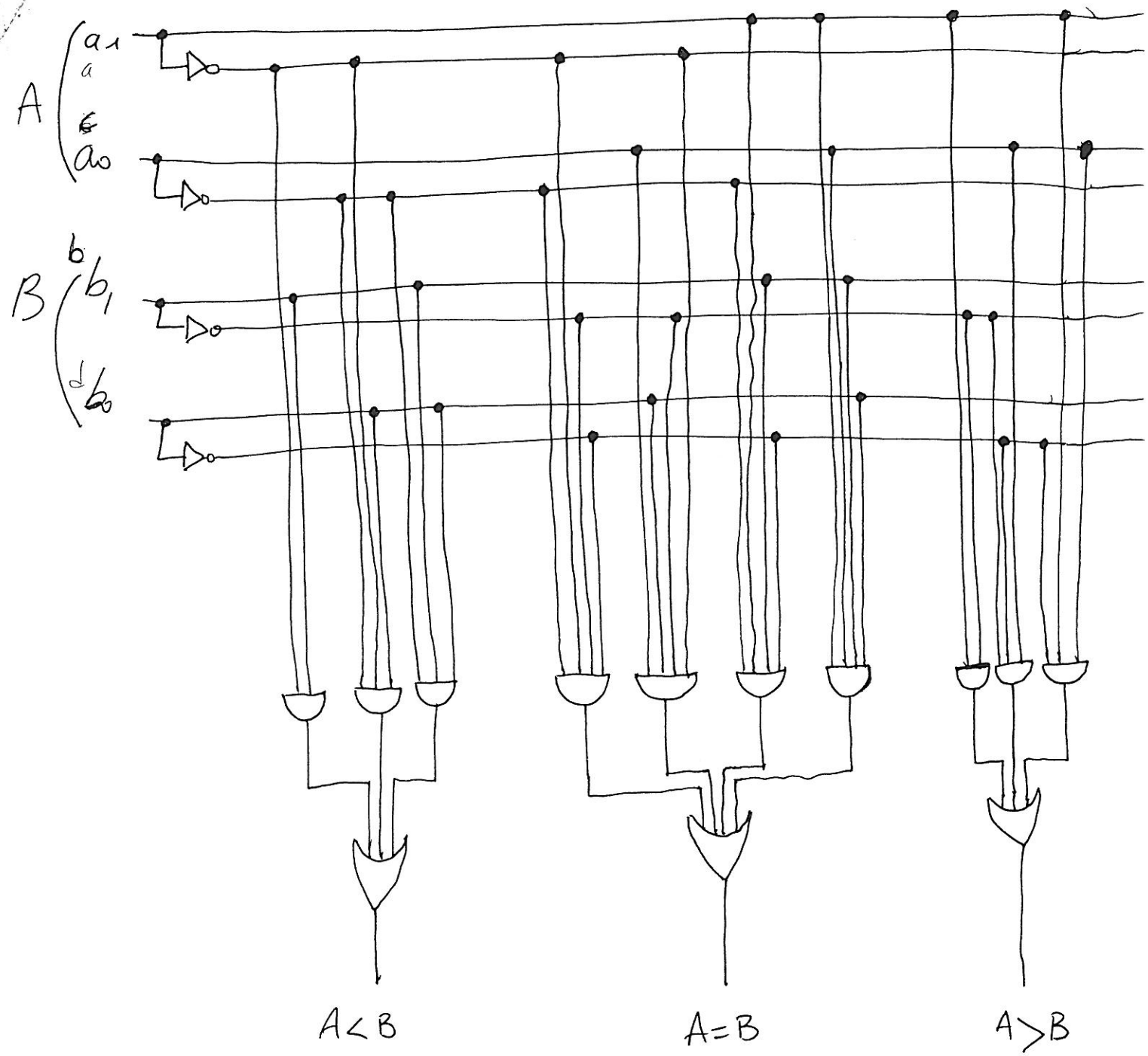
A<B

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 :



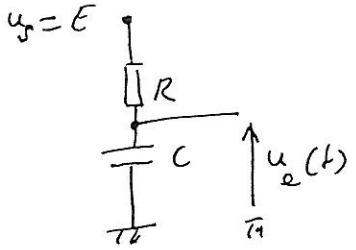
TP Electronique n°2 (suite) CORRIGÉ

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. : $u_e(0) = U_1$

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

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

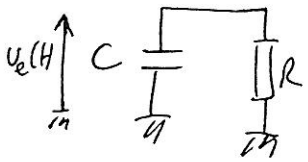
$$\rightarrow \boxed{u_e(t) = E(1 - e^{-t/\tau}) + U_1 e^{-t/\tau}} \quad 0 < t < t_1$$

* $t_1 = ?$ 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. : $u_e(0) = U_2$:

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

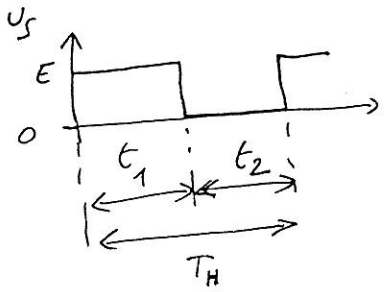
$$\left(\begin{array}{l} u_e(0) = U_2 \\ "u_e(\infty)" = 0 \end{array} \right)$$

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

* $t_2 = ?$ 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 : $K_0 = \frac{t_1}{T_H} = \frac{1}{2} \rightarrow \frac{\tau . \ln\left(\frac{U_1 - E}{U_2 - E}\right)}{\tau . \ln\left(\frac{U_2 (U_1 - E)}{U_1 (U_2 - E)}\right)} = \frac{1}{2}$

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

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

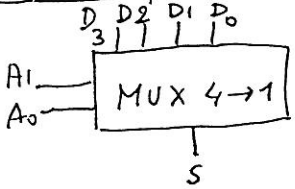
$$\rightarrow \left(\frac{U_1 - E}{U_2 - E}\right)^2 = \frac{U_2}{U_1}\left(\frac{U_1 - E}{U_2 - E}\right) \rightarrow \frac{U_1 - E}{U_2 - E} = \frac{U_2}{U_1}$$

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

* A.N : ...

3.4. Multiplexeur

a)

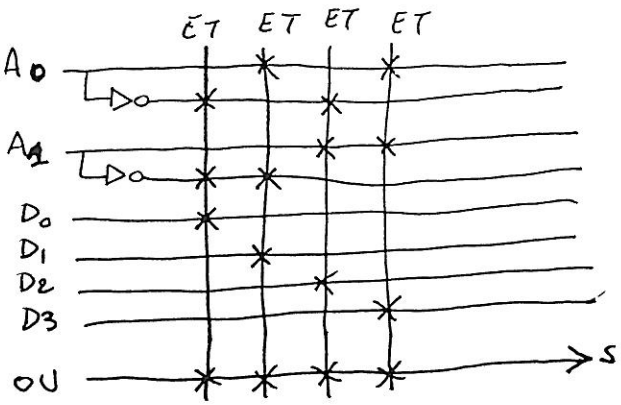


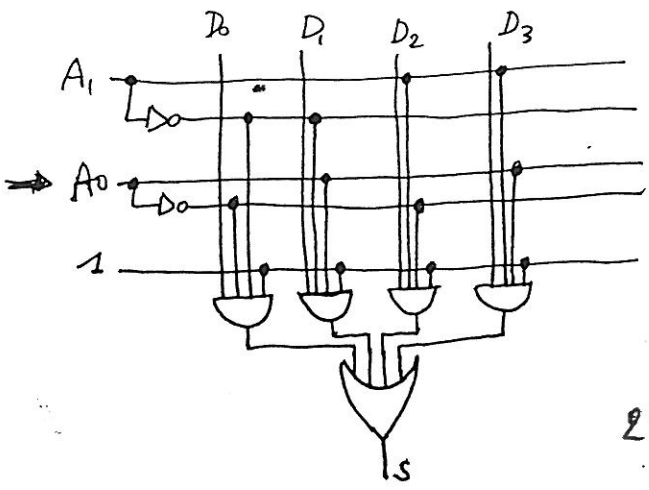
b) Table de vérité :

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

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

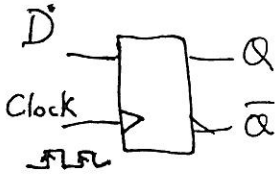
d)



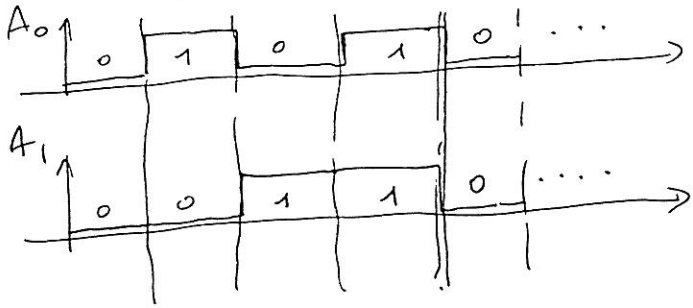


c) Rappel : bascule D ↓ :

Clock	D	Q_n
X	X	Q_{n-1}
↓	0	0
↓	1	1

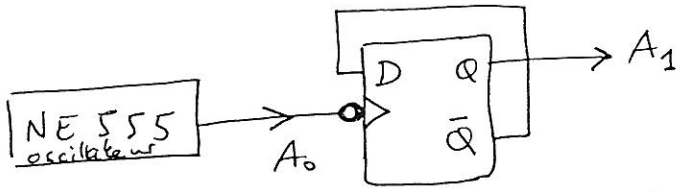


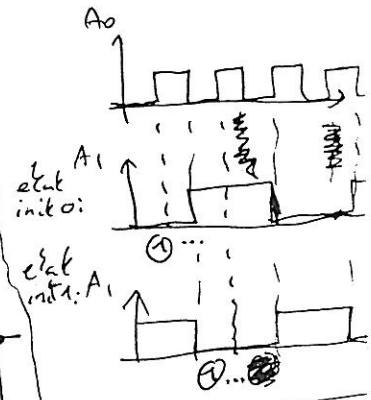
On désire la séquence suivante pour les adresses A_1, A_0 :



A_1 : division par 2 de A_0.

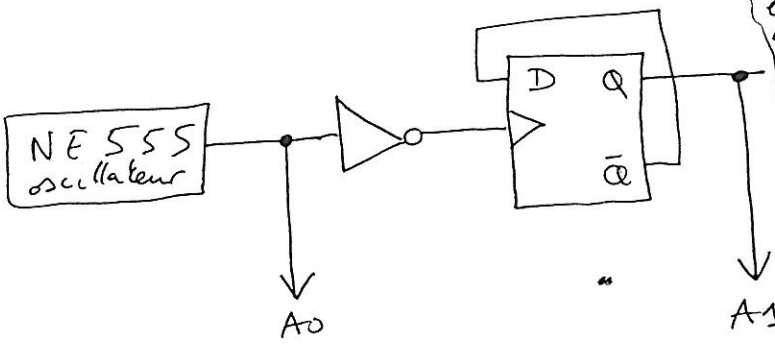
Si A_0 représente l'horloge de la bascule D ↓ générant A_1, il suffit d'une part de complémenter l'horloge car on voit que le changement d'état de A_1 a lieu pour un front descendant de A_0, et d'autre part d'envoyer la sortie $\bar{Q}$ de la bascule D en entrée de D, car à chaque front descendant de 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)





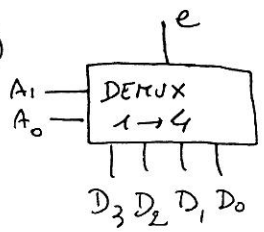
① : 1ère phase de la séquence

ou encore :



3.5. Démultiplexeur

a)



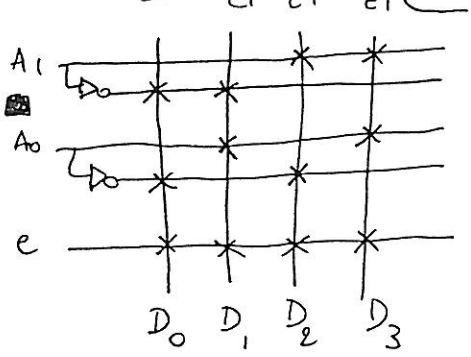
b) Table de vérité :

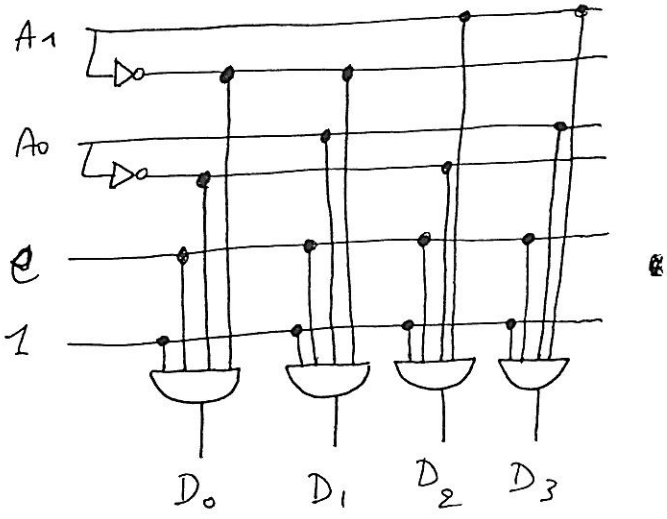
A_1 A_0	D_i
0 0	$D_0 = e$
0 1	$D_1 = e$
1 0	$D_2 = e$
1 1	$D_3 = e$

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

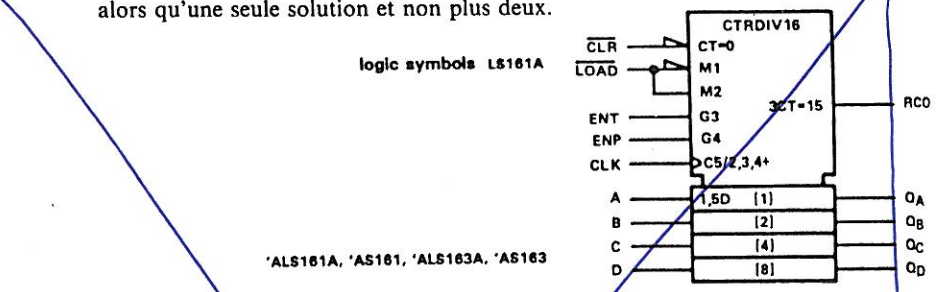
4.6.1. Affichage d'un digit :

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)

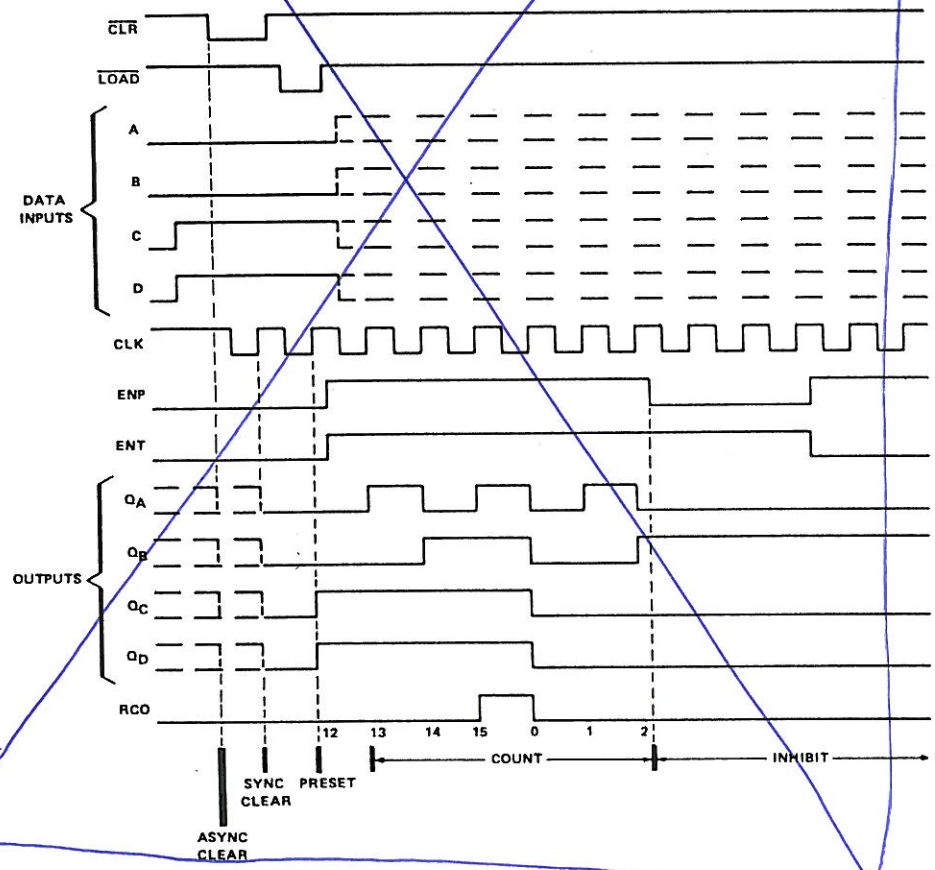
sorties complémentées

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



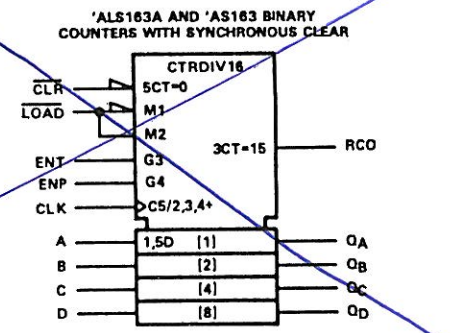
'ALS161A, 'AS161, 'ALS163A, 'AS163

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

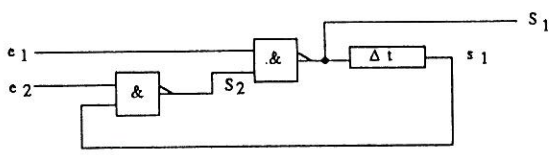
1.

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 :



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

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

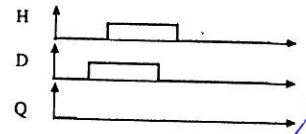
d'où la matrice d'excitation :

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

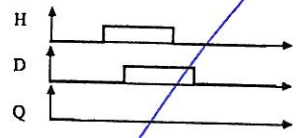
TD ELN n° 3
Logique séquentielle 1
Corrigé

9. Mêmes exercices avec une bascule D latch.

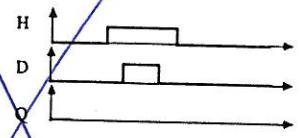
a.



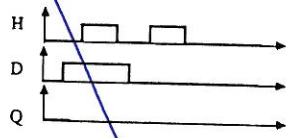
b.



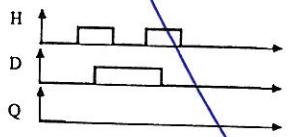
c.



d.

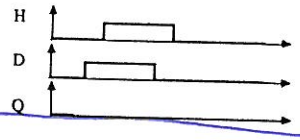


e.

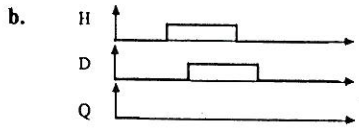


10. Mêmes exercices pour une bascule D synchronisée par un front positif (*positive edge triggered*).

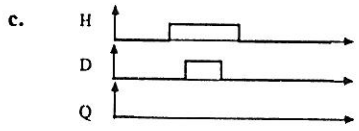
a.



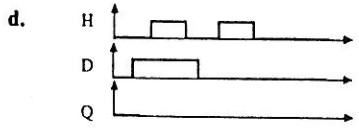
b.



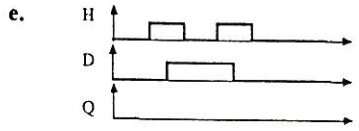
c.



d.



e.



Rappels :

D

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

JK

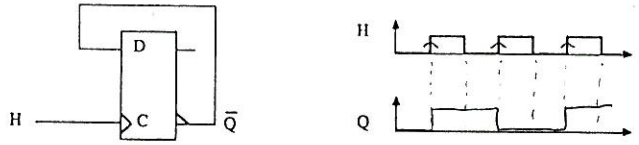
J	K	Q_n
0	0	Q_{n-1}
0	1	0
1	0	1
1	1	$\overline{Q}_{n-1}$

ou encore :

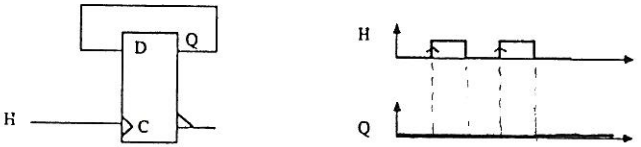
transition $Q_{n-1} \to Q_n$	J K
$0 \to 0$	0 X
$0 \to 1$	1 X
$1 \to 1$	X 0
$1 \to 0$	X 1

2. Compléter les chronogrammes pour chacun des schémas suivants :

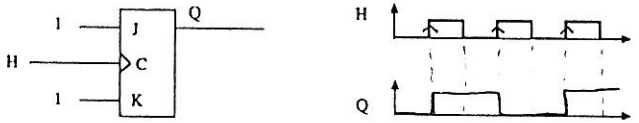
a.



b.



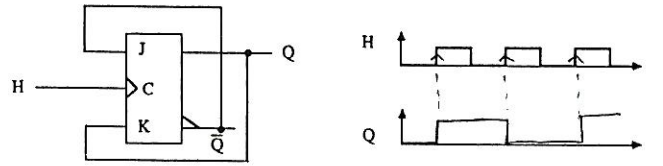
c.



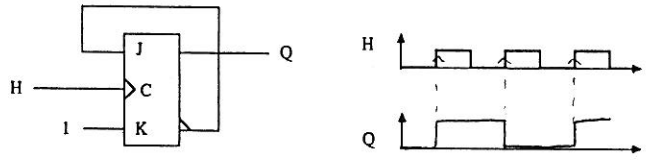
Diviseur par 2 de la fréq. de H.

2 suite

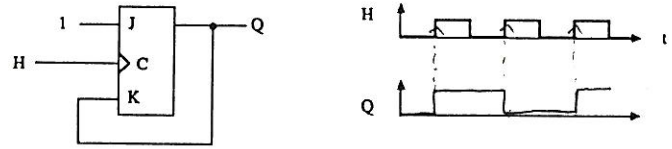
d.



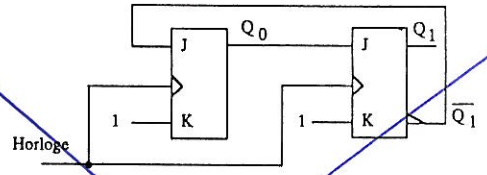
e.



f.

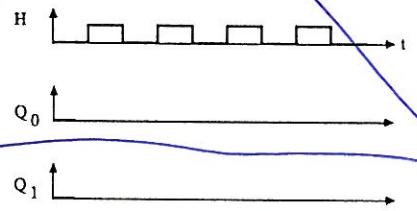


12. On donne le schéma suivant :

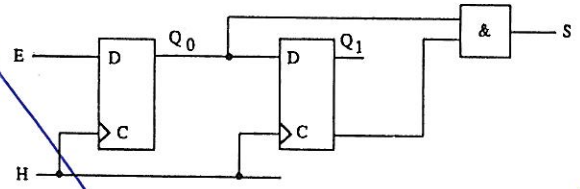


Initialement $Q_0 = Q_1 = 0$. Quelles seront les valeurs des sorties Q_0 et Q_1 ?

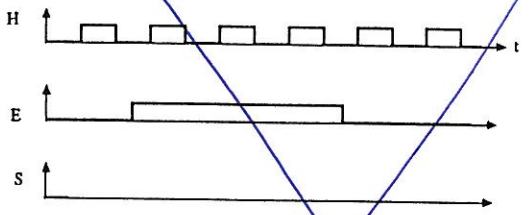
a. Après une impulsion d'horloge.
b. Après deux impulsions d'horloge.
c. Après trois impulsions d'horloge.
d. Compléter le chronogramme suivant.



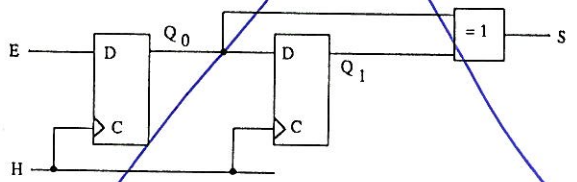
13. Analyser le fonctionnement du montage suivant :



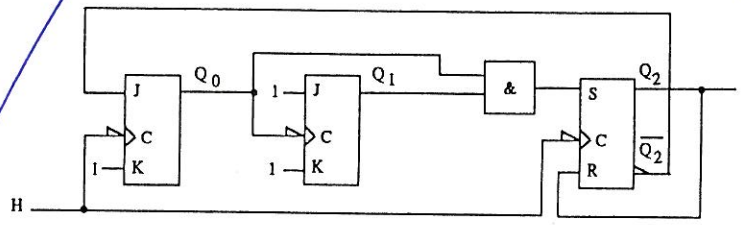
et compléter le chronogramme.



14. Même exercice avec le montage.



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

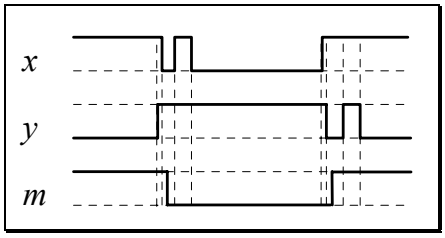


30

TD 3 CORRIGE. LOGIQUE SEQUENTIELLE 1

3. Bascule RS à entrées complémentées

COMMENTAIRES

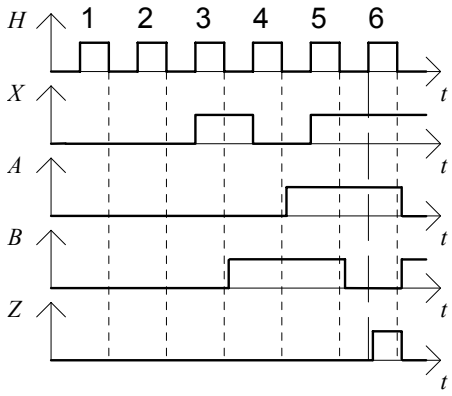


$x = \overline{R}$	$y = \overline{S}$	$m = Q_n$	***Fonction***
1	1	Q_{n-1}	*Mémorisation*
0	1	0	*RESET (Mise à 0 de Q)*
1	0	1	*SET (Mise à 1 de Q)*
0	0		*Combinaison interdite*

Application anti-rebond de la bascule RS

4. Système séquentiel

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

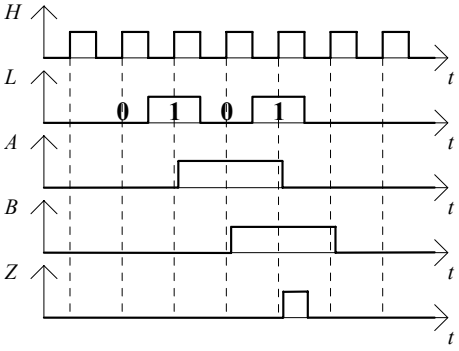


TD 3 ANNEXE CORRIGE. LOGIQUE SEQUENTIELLE 1

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

1. *Solution à logique câblée (1)*

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



$$\begin{cases} J_A = L \\ K_A = B \end{cases}$$

$$\begin{cases} J_B = A \\ K_B = \overline{A} \end{cases}$$

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

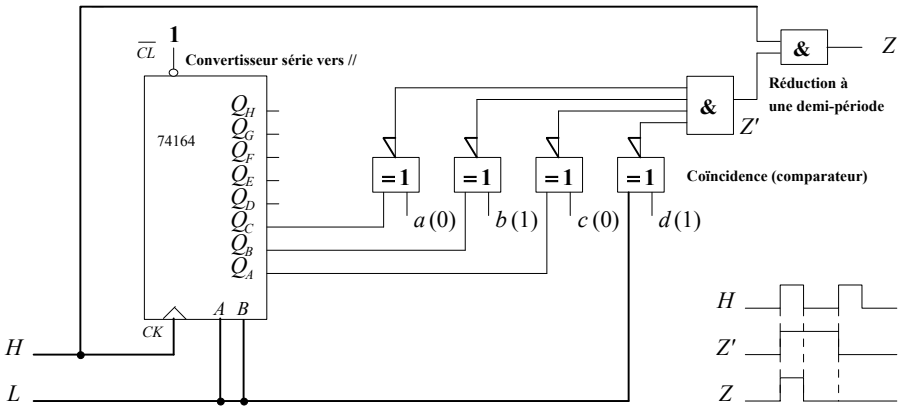
1.2. Séquence à détecter :

Séquence à détecter $a\ b\ c\ d$

0	1	0	1

2. *Solution à logique câblée (2)*

2.1.

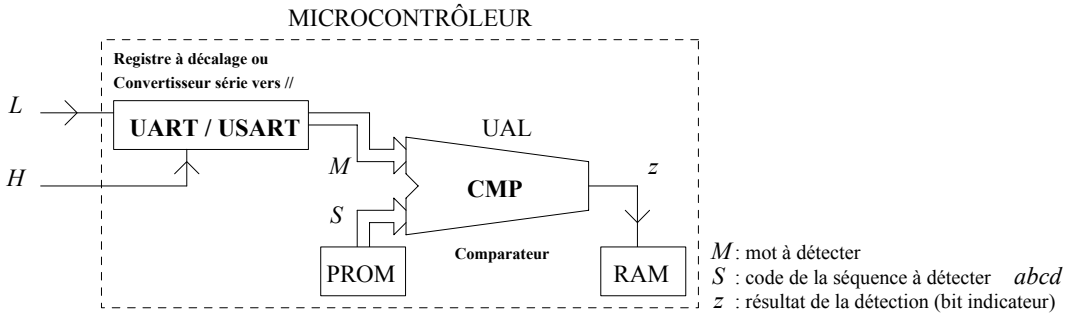


2.2.

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.



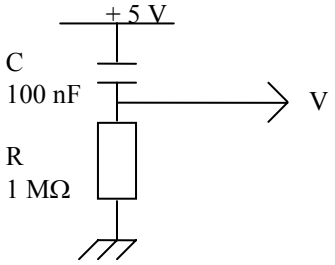
3.2.

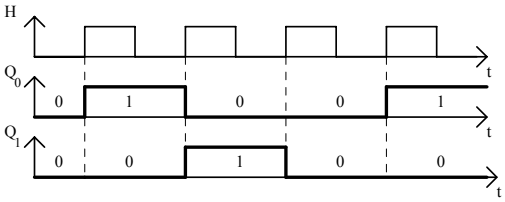
Avantage
• Le programme logé dans le microcontrôleur peut être modifié pour réaliser d'autres opérations que la simple détection d'une séquence.

TP 3 CORRIGE. LOGIQUE SEQUENTIELLE 1

3. Bascules JK (1)

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 $Q_1Q_0 = 00$).





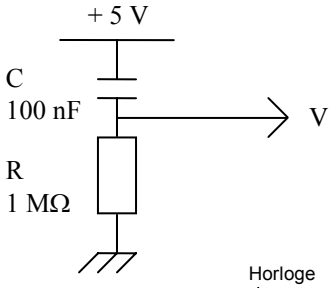
- On a un compteur par 3 : $Q_1Q_0 = 00 \rightarrow Q_1Q_0 = 01 \rightarrow Q_1Q_0 = 10 \rightarrow Q_1Q_0 = 00 \rightarrow$ ...

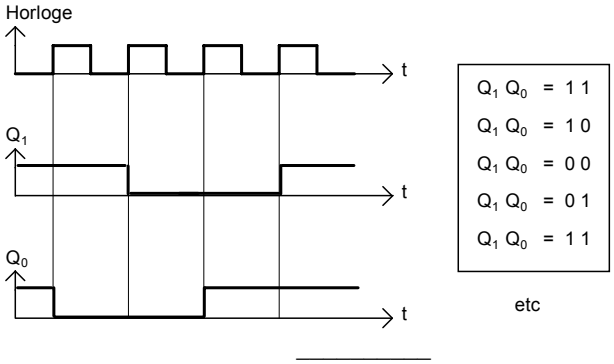
Avec l'initialisation $Q_1Q_0 = 11$, on a : $Q_1Q_0 = 11 \rightarrow Q_1Q_0 = 00 \rightarrow Q_1Q_0 = 01 \rightarrow Q_1Q_0 = 10 \rightarrow Q_1Q_0 = 00 \rightarrow$ ...

Le montage est autocorrecteur.

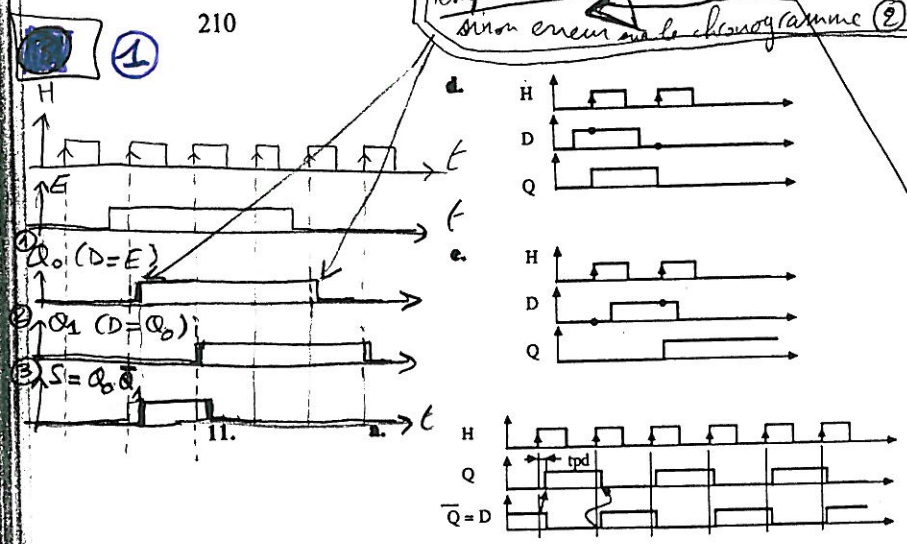
4. Bascules JK (2)

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 $Q_1Q_0 = 00$).





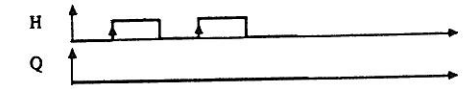
Temps de retard à faire figurer obligatoirement sinon erreur sur le chronogramme ②

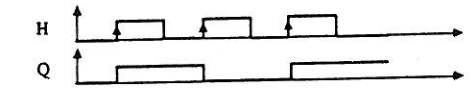


d.

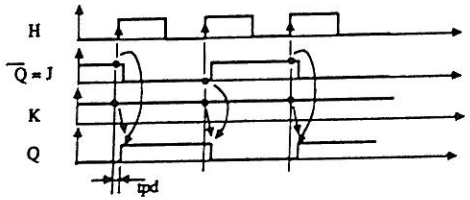
e.

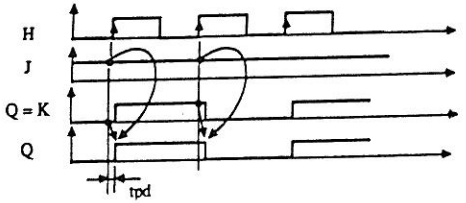
Pour mettre en évidence le phénomène décrit ci-dessus, il est indispensable de représenter la sortie Q avec le décalage temporel dû au principe de causalité (t_{pd} = temps de réaction de la bascule).

b.


c.


d. *Idem* c.

e.


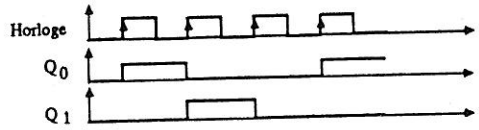
f.


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.

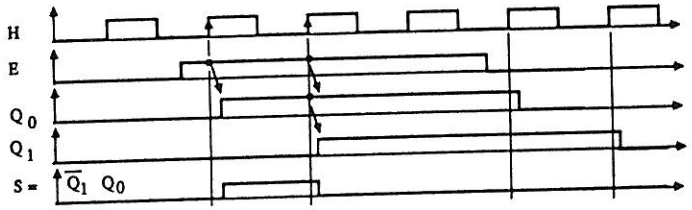
c. Après la seconde impulsion d'horloge $Q_0 = 0$ et $Q_1 = 1$.
Donc $J_0 = 0$ et $K_0 = 1$ ⇒ La sortie passera à 0 ainsi que Q_1, puisque $J_1 = 0$ et $K_1 = 1$.

d. D'où le chronogramme :



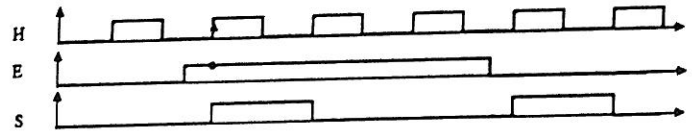
Les bascules sont synchronisées sur des fronts montants.

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

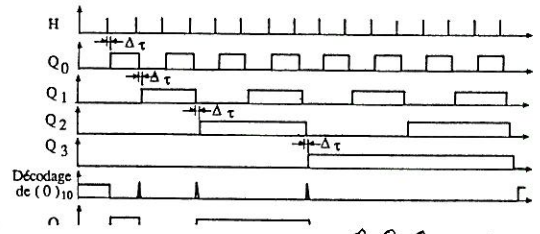
34

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

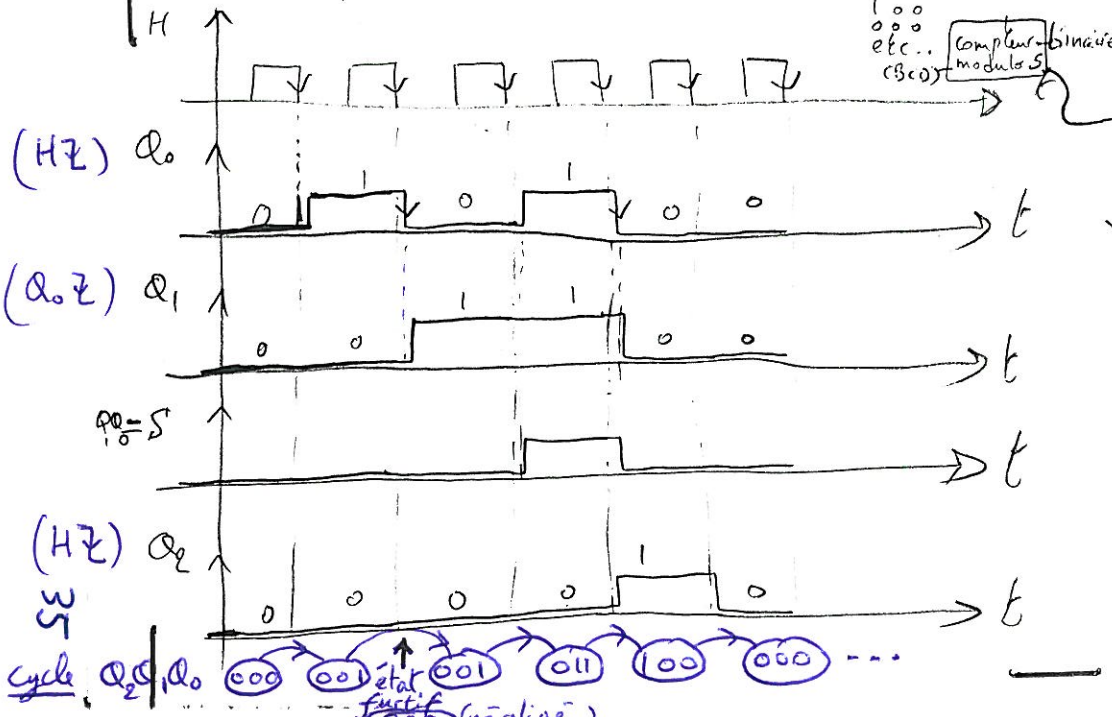
$Q_1Q_0 = S$



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

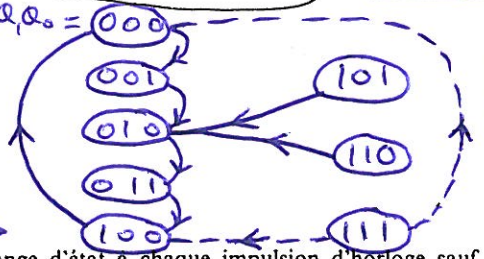
compteur binaire modulo 5

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
.	.	.

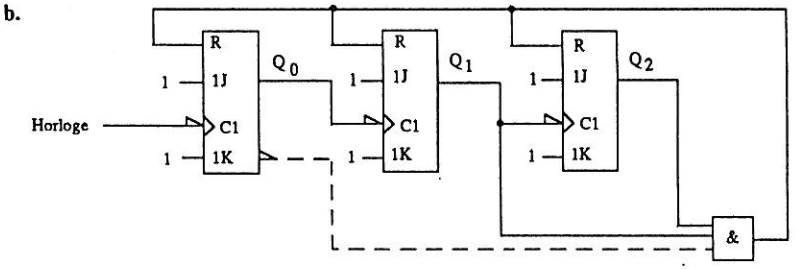


le compteur est autocorrecteur

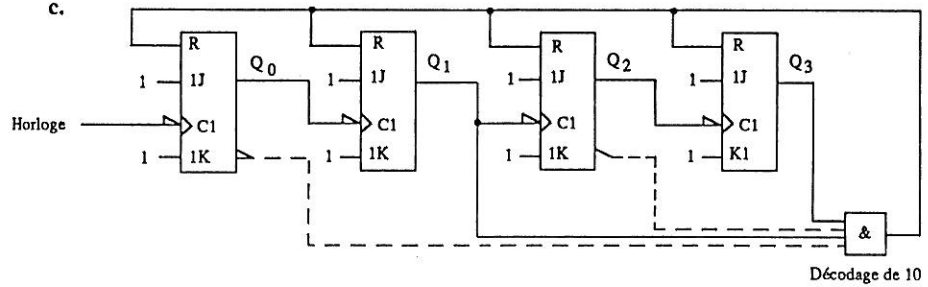
La première bascule change d'état à chaque impulsion d'horloge sauf si le contenu du compteur est 4. Ce qui impose le schéma :

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 ----)

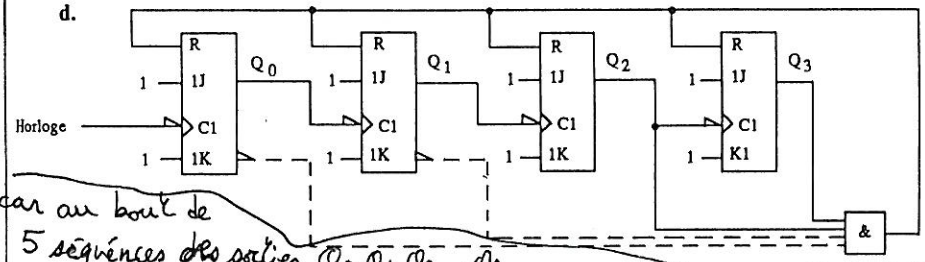
b.



c.



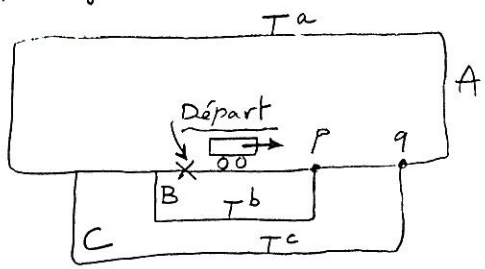
d.



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 :

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

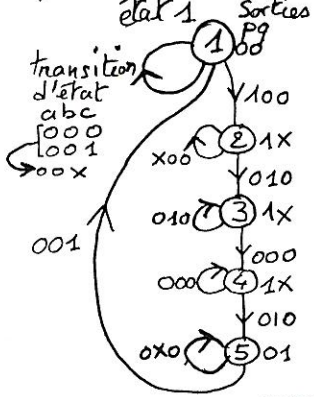
contacts :

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

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

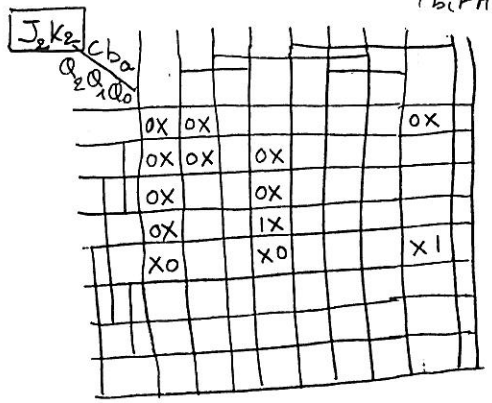
Exemple : utilisation de bascules JK

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

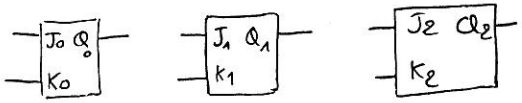
3 variables Q → 3 bascules JK :

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 :

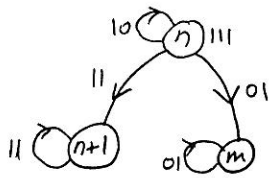
P9 \ Q1Q0 / Q2	00	1X	1X	1X
	XX	XX	XX	01

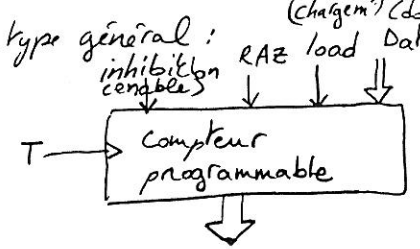
→ $p = Q_0 + Q_1 \bar{Q}_2$
$q = Q_1$

Séquenceur programmable

13.6.3 Utilisation d'un compteur programmable

Toute transition d'état est du type général :





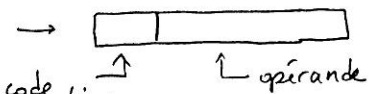
→ Commandes appliquées : - on reste dans l'état n : → enable
- on passe à n+1 : → rien (fonctionnement normal du compteur)
- on passe à m : load, data
- initialisation du compteur : RAZ

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 :

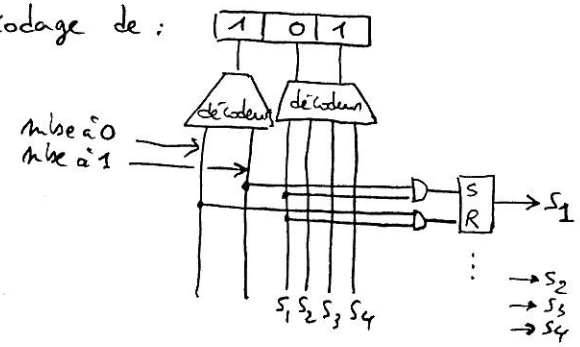


code opération : 1 : mise à 1
0 : mise à 0

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

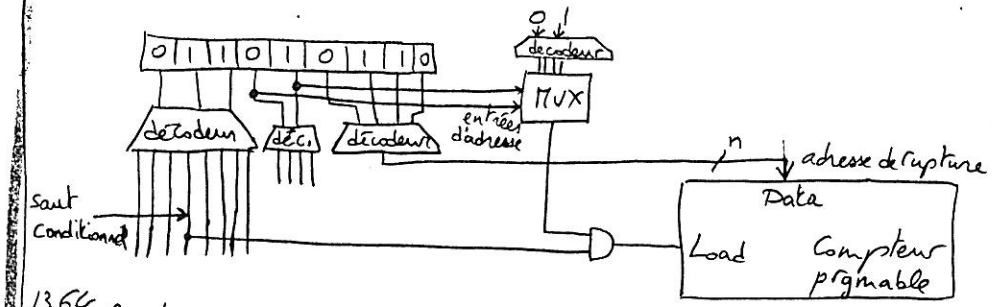
opération :		
	000	mise à 0 d'une sortie
	001	mise à 1 d'une sortie
	010	inhibition conditionnelle
	011	rupture conditionnelle
	100	rupture inconditionnelle
	101	RAZ conditionnelle
	110	RAZ inconditionnelle

opérande : sortie
condition
adresse de rupture

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

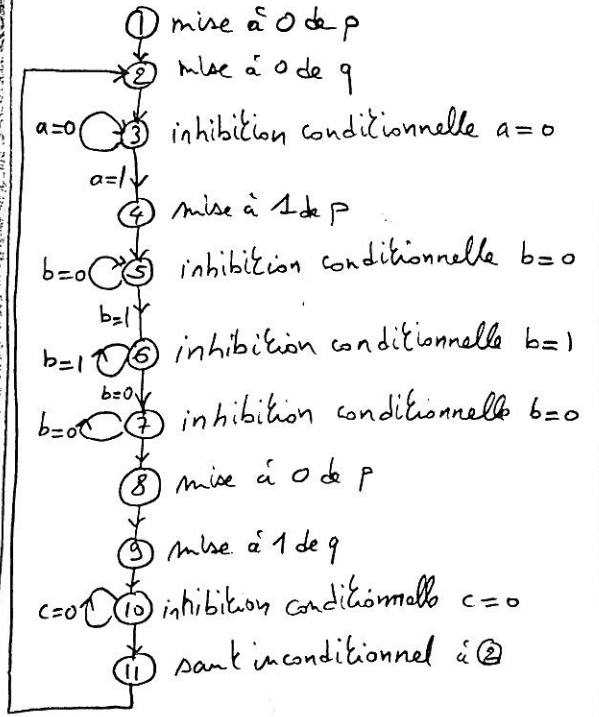
→ décodage de : 011 01 0110

condition codée sur 2 bits



13.64.

application sur l'exemple du train électrique :

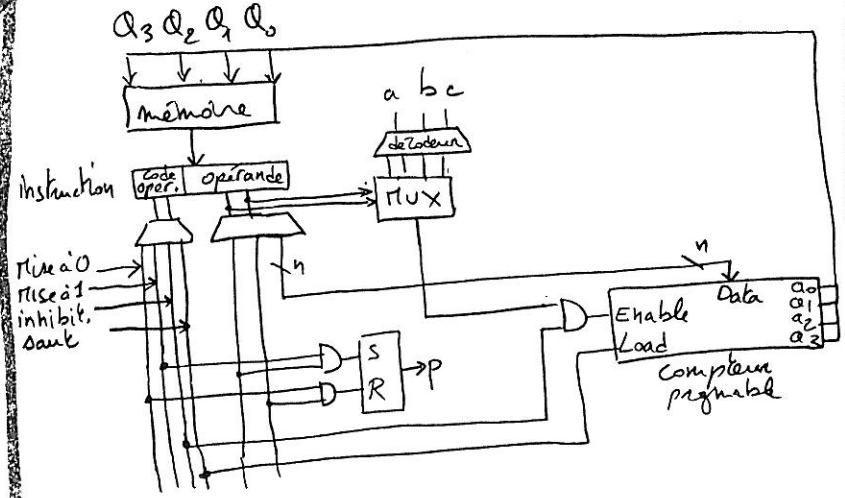


Codage :

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

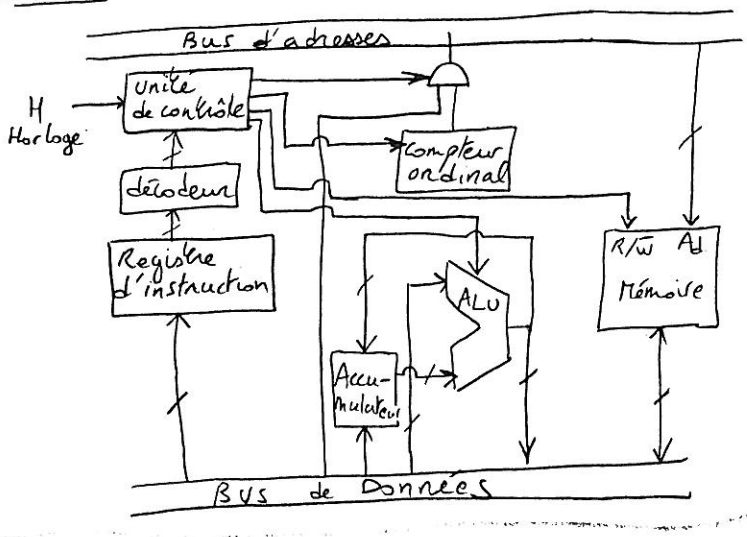
Programme :	état	mnémonique	code binaire
	0000	MAZ P	0000
	0001	MAZ Q	0001
	0010	INH A0	1000
	0011	MAU P	0100
	0100	INH B0	1001
	0101	INH B1	1011
	0110	INH B0	1001
	0111	MAZ P	0000
	1000	MAU Q	0101
	1001	INH C0	1010
	1010	JMP 2	1100

Matérialisation :

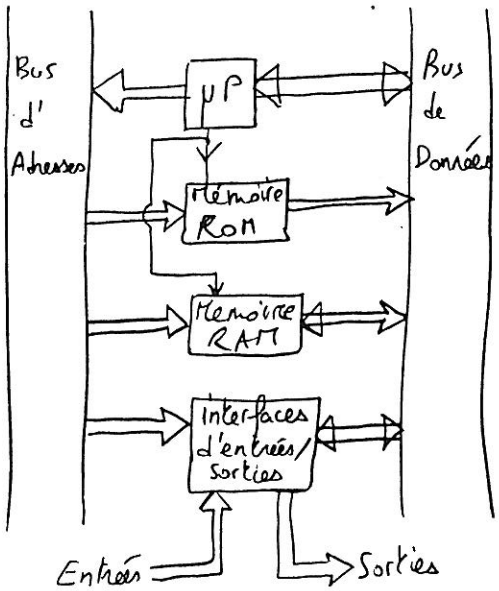


13.6.5 Utilisation d'un microprocesseur

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)

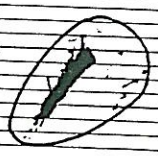
13.7. Conclusion

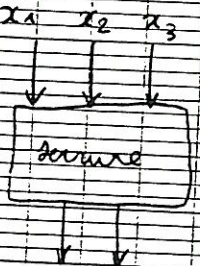
Logique câblée ≡ utilisation de portes combinatoires et séquentielles
- Avantage : rapidité du système
- inconvenient : structure figée → manque de souplesse pour une modification - adaptation du système

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

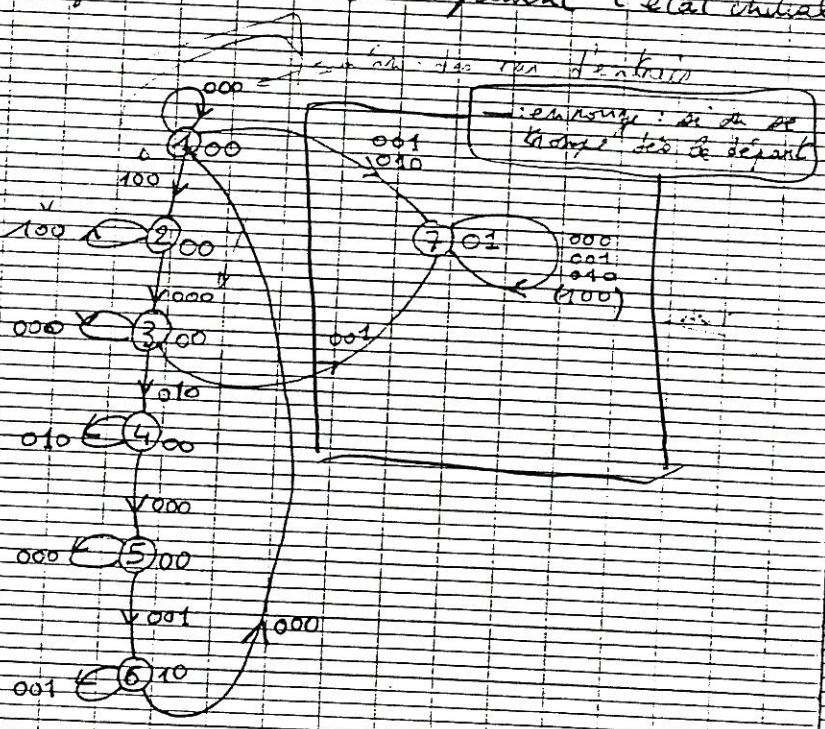


Table des états :

$x_3x_2x_1$									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 eq. 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 eq. On décrète qu'ils le sont car ça nous arrange. (on encadre ▭)
3.5 non eq. car ça contient 7. ⇒ on barre ~~3.5~~ puis ~~2.4~~ ~~2.5~~ ~~3.4~~

états | 2-3 équivalents
| 4-5 équivalents

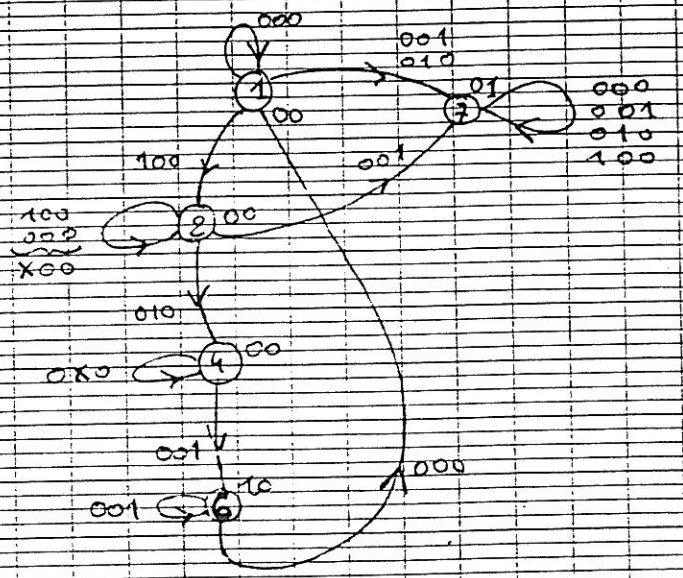
nouvelle table des états avec simplification dûe aux équiv.

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

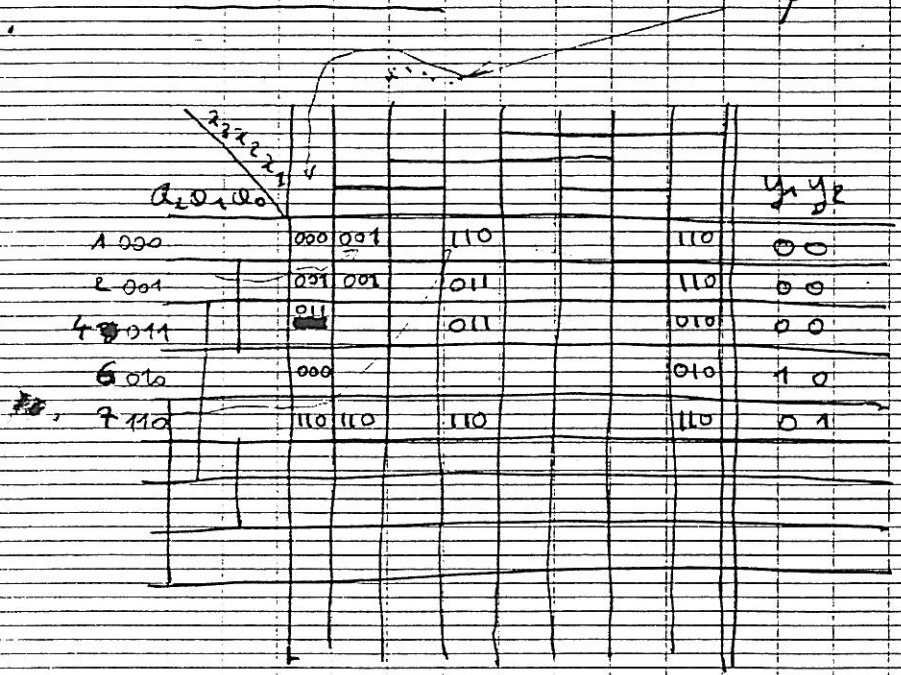
pour le voir, on fait un nouveau schéma de fluence :



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.



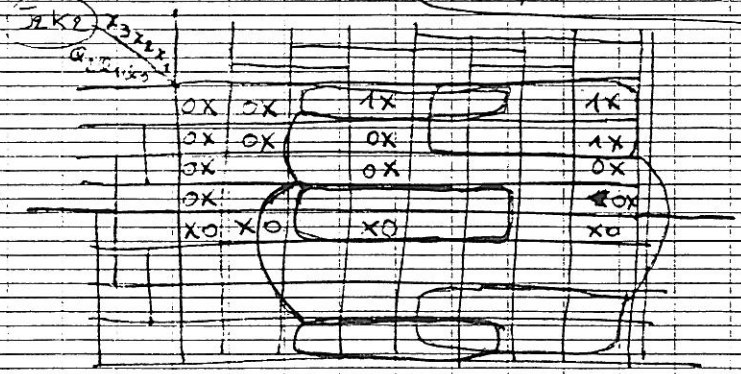
répétition de la table de la page précédente avec codage des états :

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

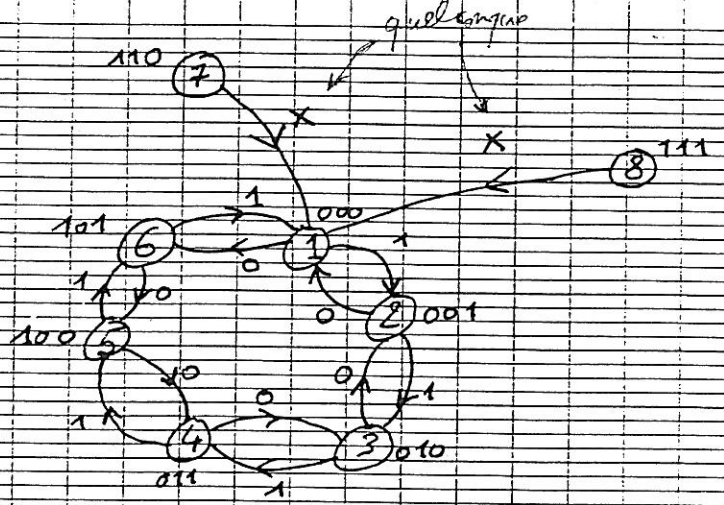
les cases vides sont indifférentes à nous de faire ces meilleurs regroupements

(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 $y_2 y_1 y_0$

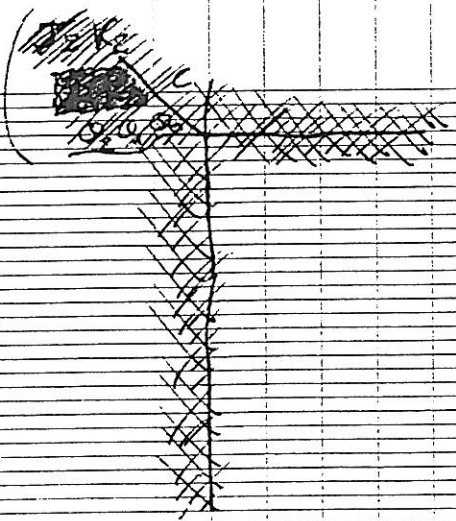


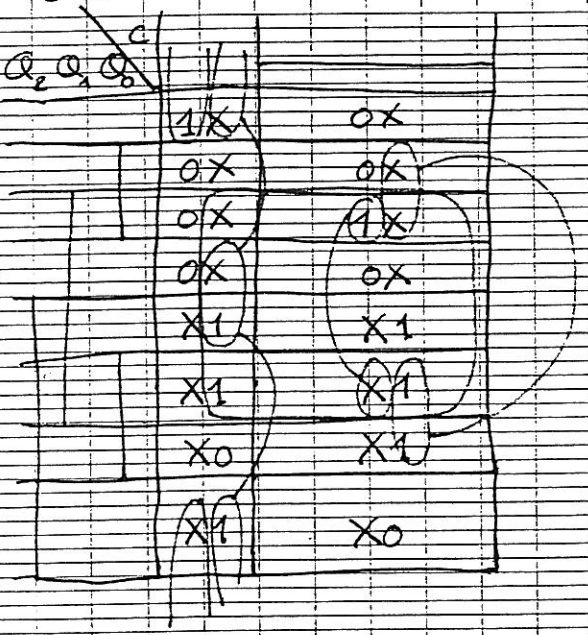
C			$y_2\ y_1\ y_0$
1	6	2	0 0 0
2	1	3	0 0 1
3	2	4	0 1 0
4	3	5	0 1 1
5	4	6	1 0 0
6	5	1	1 0 1
7	1	1	1 1 0
8	1	1	1 1 1

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

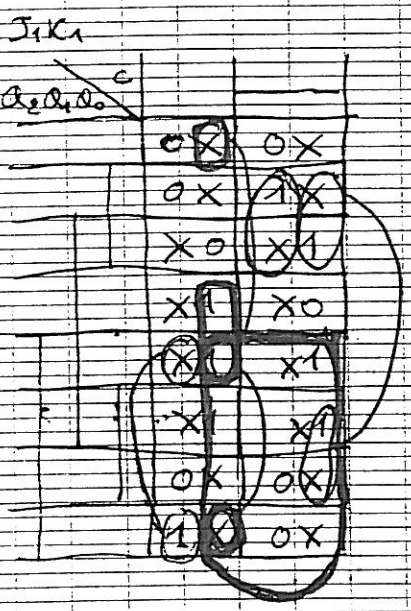
$\Rightarrow$ $y_2 = Q_2$, $y_1 = Q_1$, $y_0 = Q_0$

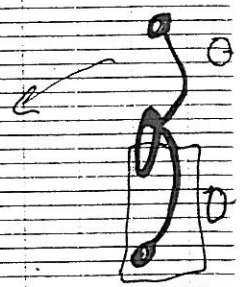




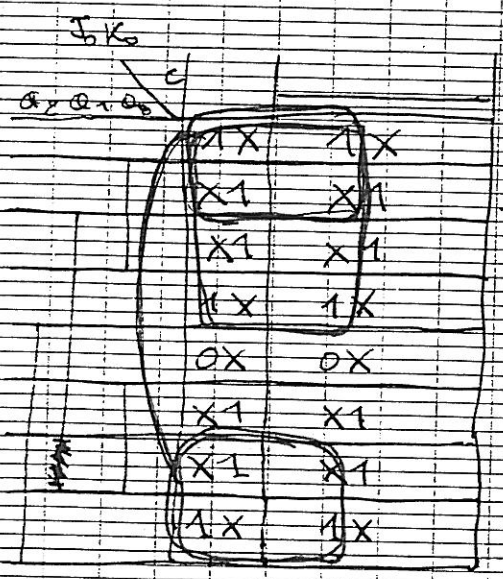
$$J_2 = \bar{c}\,\bar{Q}_1\bar{Q}_0 \vee c\,Q_1Q_0$$
$$K_2 = Q_1 \vee \bar{c}\,\bar{Q}_0 \vee cQ_0$$

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





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



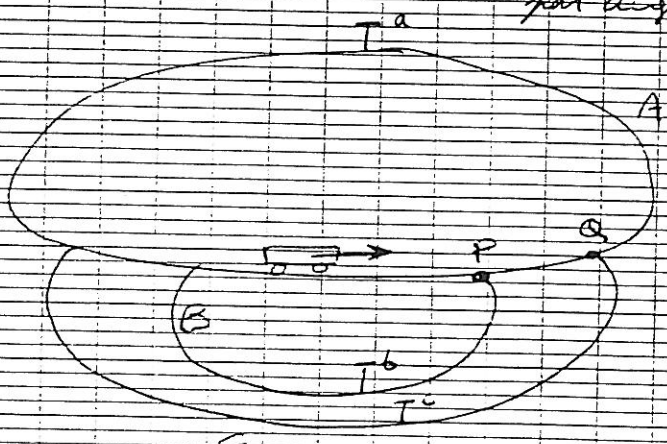
$$\begin{cases} J_2 = \bar{c}\,\bar{Q}_1\bar{Q}_0 \vee c\,Q_1 Q_0 \\ K_2 = Q_1 \vee c\,\bar{Q}_0 \vee c\,Q_0 \end{cases}$$

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

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

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 $\begin{cases} P \\ Q \end{cases}$ $\begin{cases} 0 : \text{non dévié} \\ 1 : \text{dévié} \end{cases}$

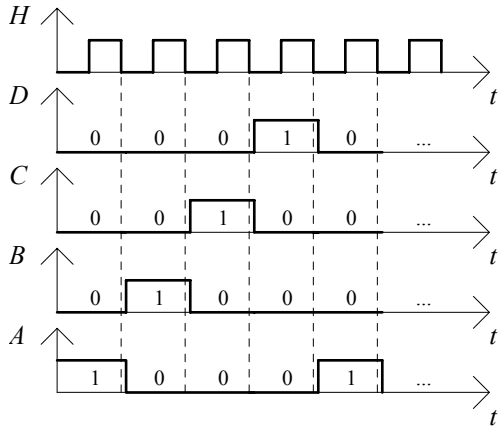
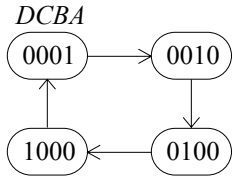
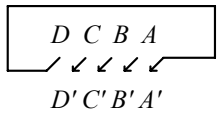
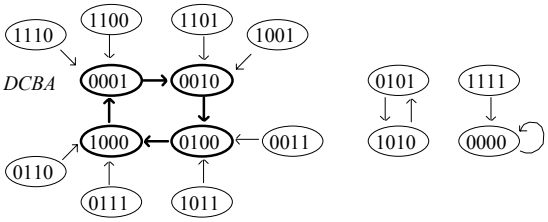
T: contact - a, b, c $\begin{cases} 0 & \text{au repos} \\ 1 & \text{passage du train} \end{cases}$

itinéraire → A, B, B, C

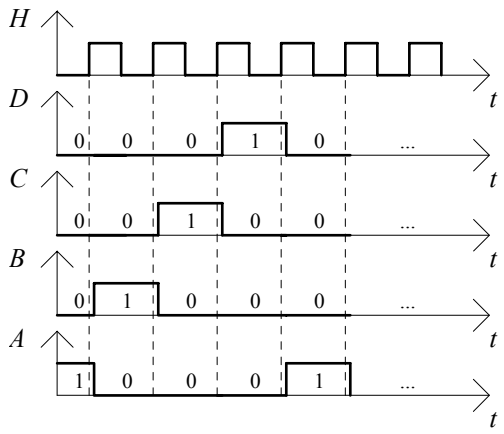
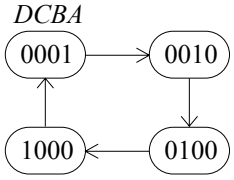
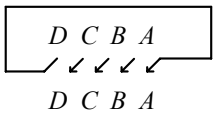
TD 4 CORRIGE. LOGIQUE SEQUENTIELLE 2

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

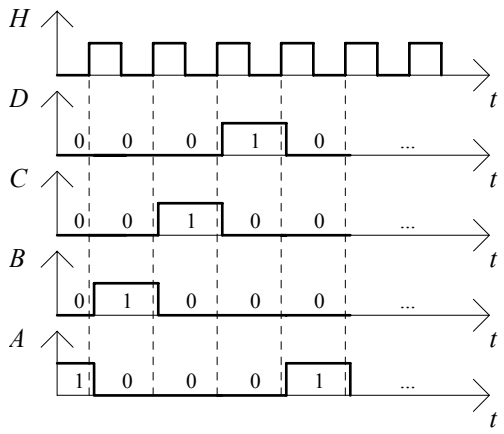
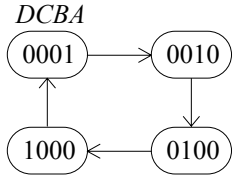
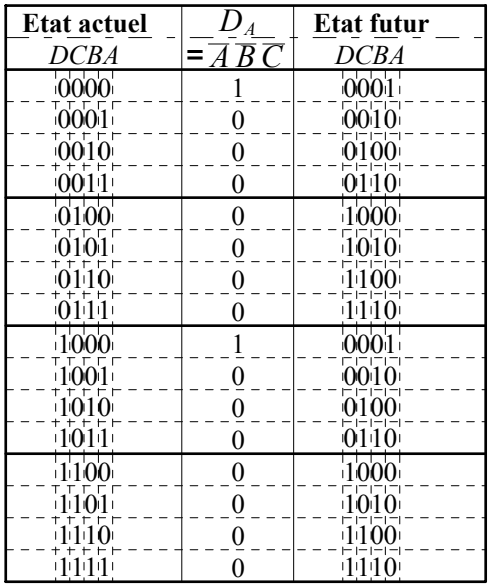
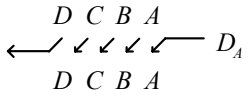
1. Compteur en anneau à bascules JK

	Résultat	*Commentaires*
1.		
2.		Compteur modulo 4 câblé en registre à décalage
3.	**Etat actuel** *DCBA* / **Etat futur** *DCBA* 0000 → 0000 0001 → 0010 0010 → 0100 0011 → 0100 0100 → 1000 0101 → 1010 0110 → 1000 0111 → 1000 1000 → 0001 1001 → 0010 1010 → 0101 1011 → 0100 1100 → 0001 1101 → 0010 1110 → 0001 1111 → 0000	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 : $DCBA = 0011 \rightarrow 0110 \rightarrow 1100 \rightarrow 1001 \rightarrow 0011$

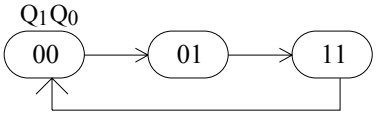
3. Compteur ouvert à bascules D

	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

4. 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

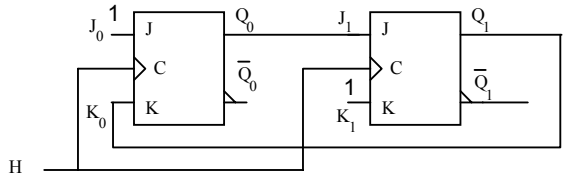
$$\begin{cases} J_0 = 1 \\ K_0 = Q_1 \end{cases}$$

. 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

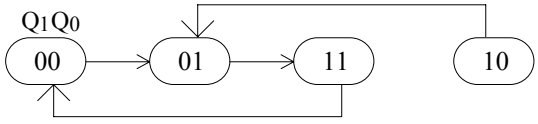
$$\begin{cases} J_1 = Q_0 \\ K_1 = 1 \end{cases}$$

. Schéma de câblage :



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

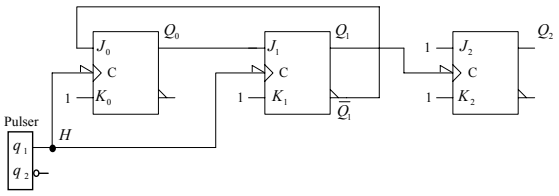
Le compteur est donc **autocorrecteur** :



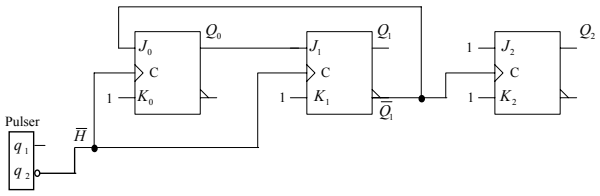
d) Si le compteur n'avait pas été autocorrecteur, le rendre autocorrecteur se ferait en reprenant la synthèse *a)* et en éliminant les choix *xx* effectués dans les tables de Karnaugh pour les forcer selon le cycle autocorrigé.

TP 4 CORRIGE. LOGIQUE SEQUENTIELLE 2

3. Compteurs



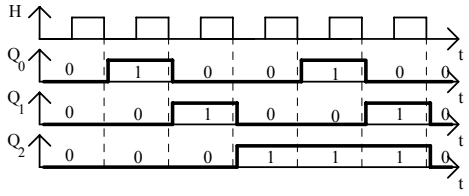
(1) ≡



(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

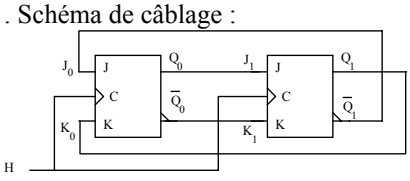
$$\begin{cases} J_0 = \overline{Q}_1 \\ K_0 = Q_1 \end{cases}$$

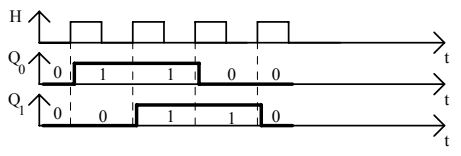
. Table de Karnaugh établissant les entrées J_1K_1 de la bascule de sortie Q_1 :

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

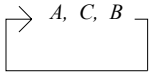
$$\begin{cases} J_1 = Q_0 \\ K_1 = \overline{Q}_0 \end{cases}$$

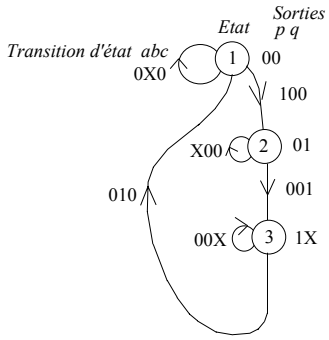
. Schéma de câblage :





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	p	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)	000	100	110	010	011	111	101	001	p	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

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

Voir livre VHDL (Meaudre & all …).

2. Programme VHDL de l'Opérateur OU exclusif

Voir livre VHDL (Meaudre & all …).

3. Programme VHDL d'un Multiplexeur 2 → 1

Voir livre VHDL (Meaudre & all …).

4. Programme VHDL d'une Bascule D latch

Voir livre VHDL (Meaudre & all …).

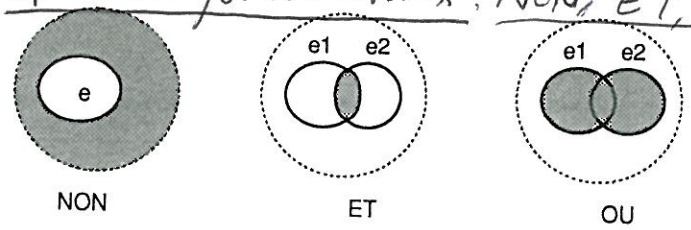


Figure III-5

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.

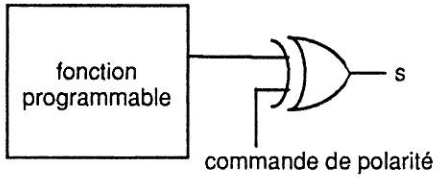


Figure III-13

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 :

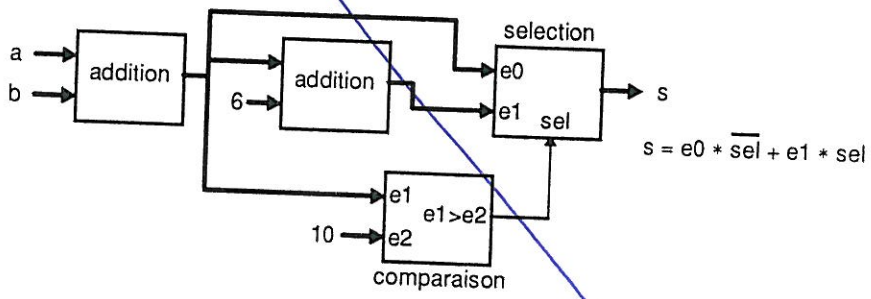


Figure III-14

Symbole et logigramme

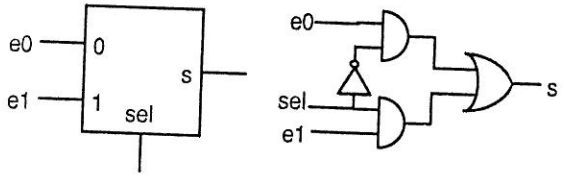


Figure III-15

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

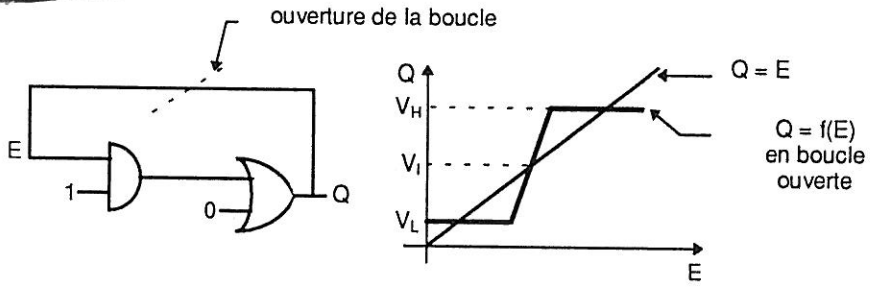


Figure III-17

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].

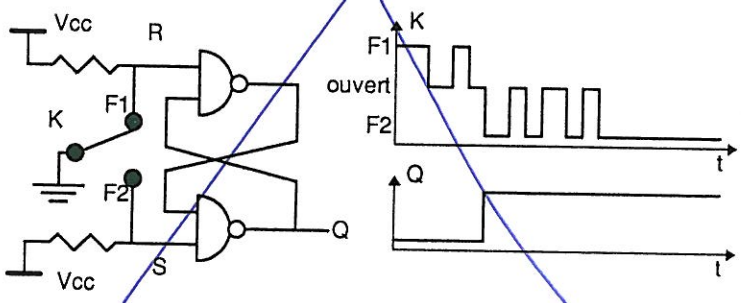


Figure III-20

[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).

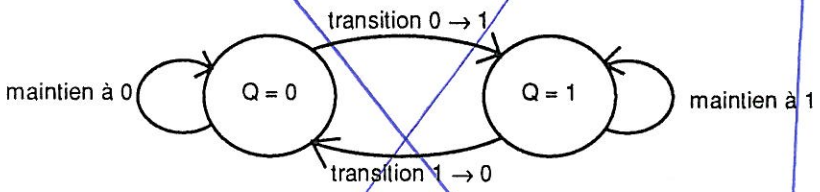


Figure III-23

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 :

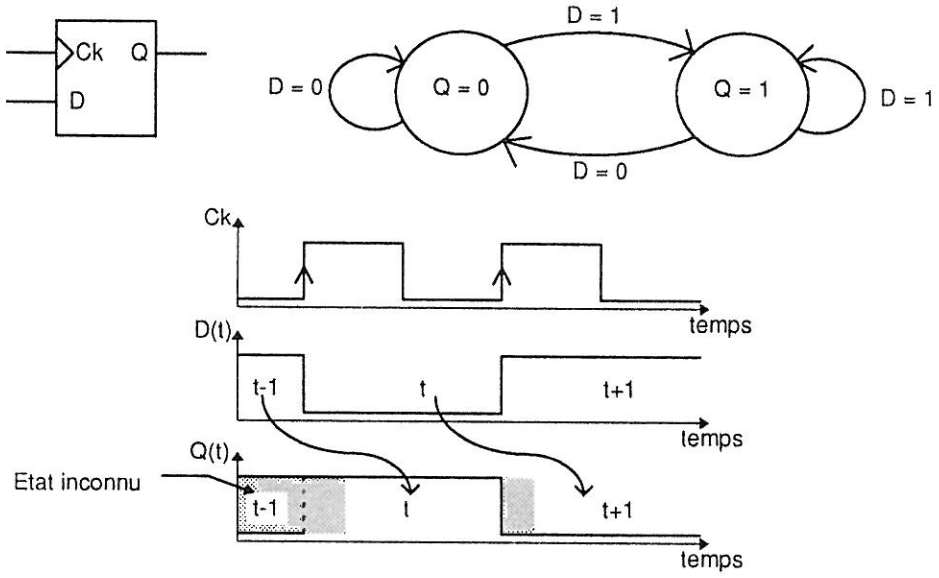


Figure III-24

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.

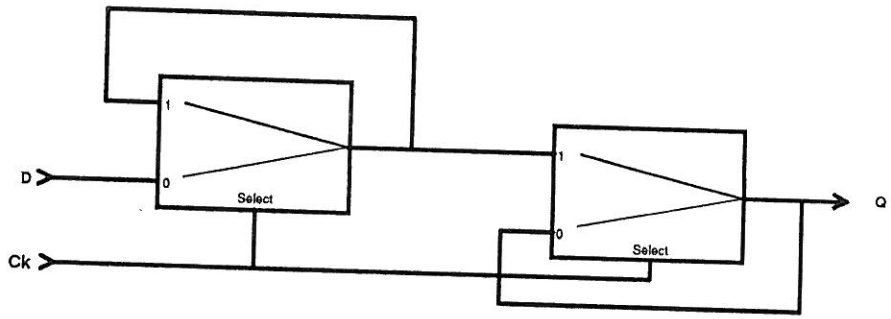


Figure III-25

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

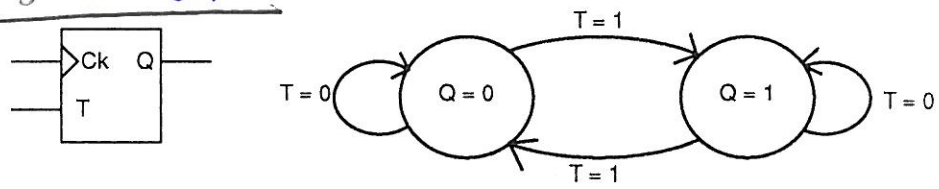


Figure III-26

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 :

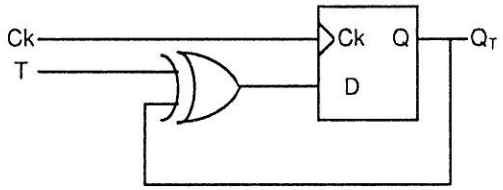


Figure III-27

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") :

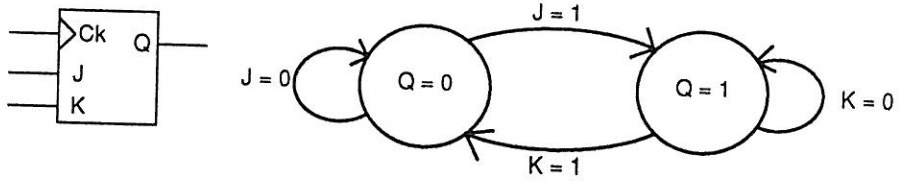


Figure III-28

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

1. Programme VHDL d'une Bascule RS (asynchrone)
Voir livre VHDL (Meaudre & all …).

2. Programme VHDL d'une Bascule D (>0 edge triggered)
Voir livre VHDL (Meaudre & all …).

3. Programme VHDL d'une Bascule T (>0 edge triggered)
Voir livre VHDL (Meaudre & all …).

4. Programme VHDL d'une Bascule JK (>0 edge triggered)
Voir livre VHDL (Meaudre & all …).

_____

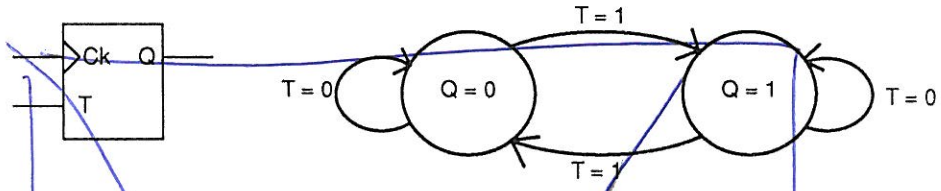


Figure III-26

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 :

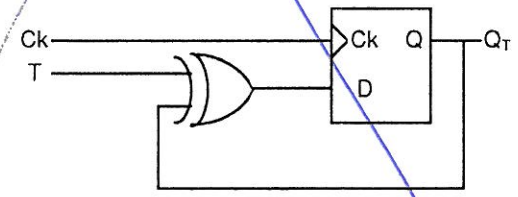


Figure III-27

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") :

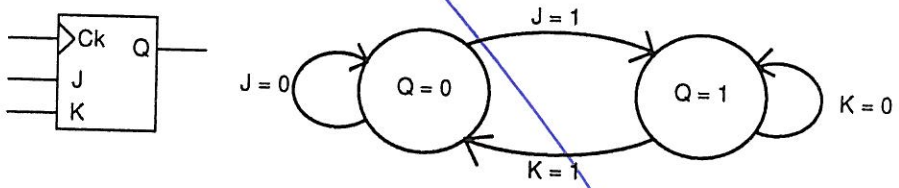


Figure III-28

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 :

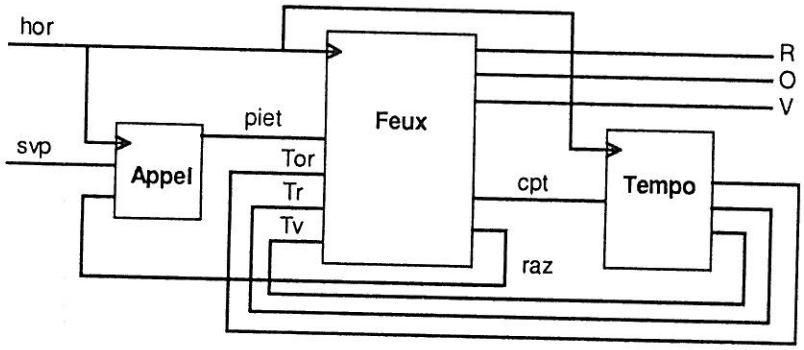


Figure V-17

[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 :

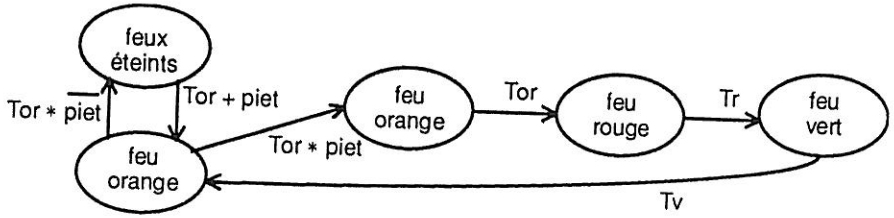


Figure V-18

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) :

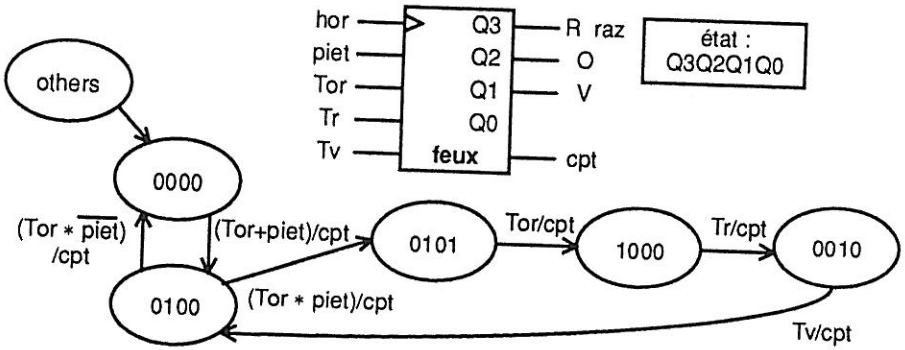


Figure V-19

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.

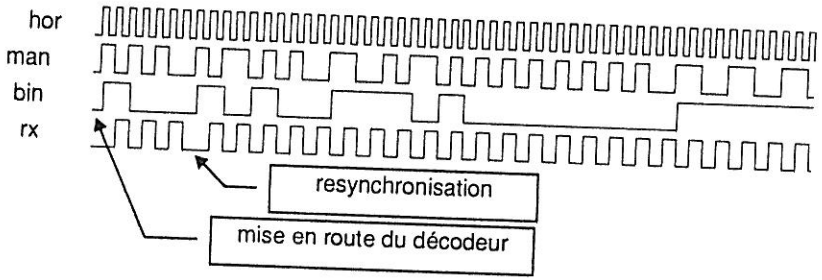


Figure V-20

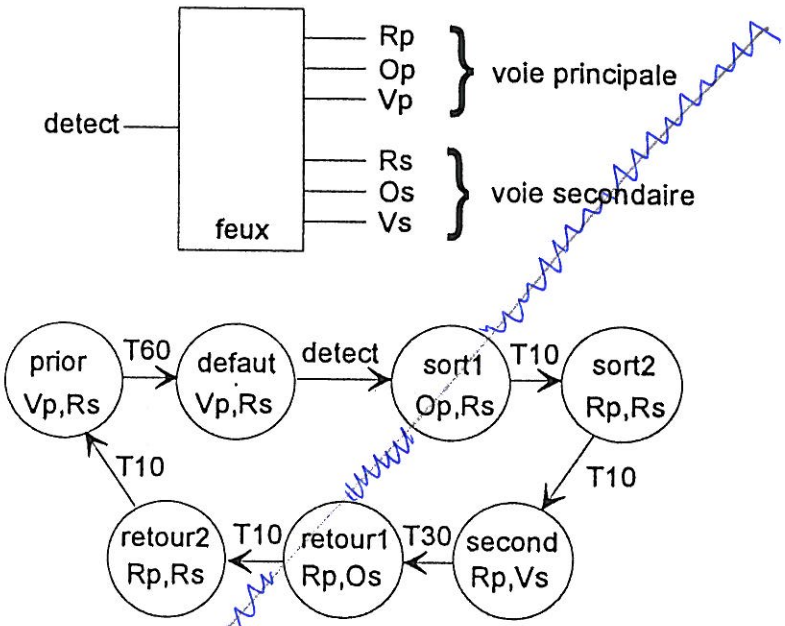


Figure II-1 : *Une vue fonctionnelle de la commande de feux*

outils de modélisation, utilisables pour simuler le fonctionnement d'un système, elles ne sont pas toutes synthétisables. Ce premier aperçu va tenter de placer la scène et de donner un exemple de la différence qui peut exister entre modélisation et synthèse.

II.2.1. L'exemple incontournable : une commande de feux tricolores

Rares sont les livres traitant d'électronique numérique qui ne proposent pas une version de feu rouge, respectons donc la tradition. Dans cet exemple nous sommes évidemment obligés d'utiliser des instructions que nous n'avons pas encore étudiées, nous tenterons de les garder suffisamment simples pour que leur sens soit évident.

Ce que l'on veut obtenir

Un chemin de campagne coupe une route à grande circulation. Les flots de véhicules qui franchissent le carrefour sont contrôlés par un feu tricolore :

- Par défaut le feu est au vert pour la grande route, et au rouge pour le chemin.

- Quand arrive un véhicule sur le chemin, un détecteur de passage provoque le changement des feux pour 30 secondes, puis les feux reviennent à la situation par défaut.
- Si plusieurs véhicules se suivent sur le chemin de campagne, il ne faut pas bloquer la grande route. Le retour à la situation par défaut au bout des 30 secondes est donc inconditionnel, indépendant de l'état du détecteur de passage. De plus, la situation par défaut doit durer au moins 1 minute.
- Le passage d'une situation à l'autre se fait par une séquence de sécurité qui prend 20 secondes : 10 secondes de feux orange d'un côté et rouge de l'autre, 10 secondes de feux rouges des deux côtés.

Dire ce que l'on veut faire fournit presque la solution. Traduisons ce qui précède par un synoptique et un diagramme de transitions (figure II-1). Précisons que nous sommes encore très loin des circuits électroniques, nous ne nous préoccupons donc pas encore d'horloges ou de bascules.

Sur le diagramme de transitions de la figure II-1 chaque état porte un nom (première ligne) et rend actives les sorties indiquées (deuxième ligne). Les sorties non mentionnées dans un état sont inactives, les feux correspondants sont éteints. Les noms des sorties rappellent leurs rôles : Rp pour feu rouge sur la voie principale, Rs pour son homologue de la voie secondaire etc.

À côté de chaque transition figure une condition qui doit être vraie pour que cette transition puisse être effectuée.

L'ensemble synoptique - diagramme de transitions est suffisamment simple pour que l'on en déduise directement un programme VHDL dont voici une version :

```
-- feu_fonc.vhd
-- exemple fonctionnel non synthétisable

entity feux is
   port ( detect : in bit ;
          Rp, Op, Vp, Rs, Os, Vs : out bit );
end feux ;

architecture vue_esprit of feux is
   type etat_feu is (defaut, prior, sort1,
      sort2, second, retour1, retour2);
   constant T10 : time := 10 sec ;
   constant T30 : time := 30 sec ;
   constant T60 : time := 60 sec ;
   signal automate : etat_feu ;
begin

sorties : process
begin
   Rp <= '0' ; Op <= '0' ; Vp <= '0' ;
   Rs <= '0' ; Os <= '0' ; Vs <= '0' ;
```

```
    case automate is
       when defaut  => Vp <= '1' ; Rs <= '1' ;
       when sort1   => Op <= '1' ; Rs <= '1' ;
       when sort2   => Rp <= '1' ; Rs <= '1' ;
       when second  => Rp <= '1' ; Vs <= '1' ;
       when retour1 => Rp <= '1' ; Os <= '1' ;
       when retour2 => Rp <= '1' ; Rs <= '1' ;
       when prior   => Vp <= '1' ; Rs <= '1' ;
    end case ;
    wait on automate ;
end process sorties ;

sequence : process
begin
    case automate is
       when defaut  => if detect = '0' then
             wait until detect = '1' ;
          end if ;
          automate <= sort1 ;
          wait on automate ;
       when sort1   => wait for T10 ;
             automate <= sort2  ;
             wait on automate ;
       when sort2   => wait for T10 ;
             automate <= second ;
             wait on automate ;
       when second  => wait for T30 ;
             automate <= retour1;
             wait on automate ;
       when retour1 =>   wait for T10 ;
             automate <= retour2;
             wait on automate ;
       when retour2 => wait for T10 ;
             automate <= prior  ;
             wait on automate ;
       when prior   => wait for T60 ;
             automate <= defaut ;
             wait on automate ;
    end case ;
end process sequence ;
end vue_esprit ;
```

Dans cet exemple nous n'avons pas cherché à minimiser la longueur du programme, préférant refléter strictement, dans le code source, la structure du diagramme de transitions. Le fonctionnement se comprend bien si on réalise que :

- chaque processus est une boucle sans fin dont le code s'exécute séquentiellement ;

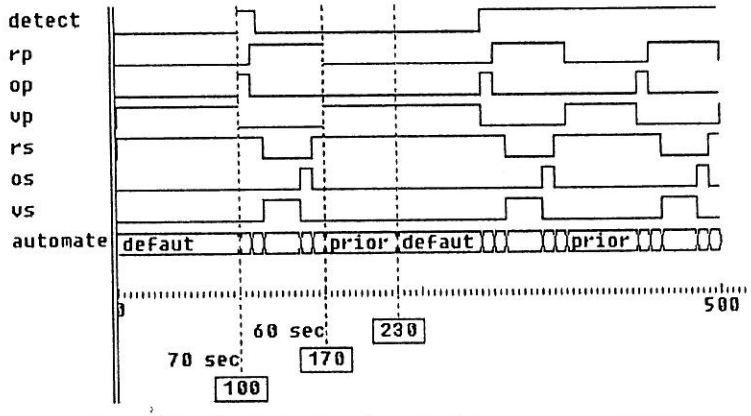


Figure II-2 : *Simulation fonctionnelle de la commande de feux*

- les deux processus s'exécutent simultanément, en parallèle : le processus « sequence » suit le diagramme de transitions tandis que le processus « sorties » calcule les valeurs des commandes des feux en fonction de l'état du précédent ;
- la modification de la valeur d'un signal ne prend effet que quand le simulateur rencontre la fin du processus ou une instruction `wait`, par défaut un signal conserve sa valeur initiale ;
- un processus peut se mettre en attente d'un événement (modification de la valeur d'un signal) ou d'un temps « simulateur », grâce à l'instruction `wait`.

L'examen rapide d'un résultat de simulation (simulateur V-system de la société Model Technology), figure II-2, nous fournit une présomption de preuve de la correction de notre programme.

Un cycle complet de blocage de la voie principale dure bien 70 secondes ; si une file de véhicules occupe la voie secondaire, notre algorithme assure bien une alternance des feux verts au rythme de 60 secondes pour la voie principale et 30 secondes pour la voie secondaire.

Pour passer à un modèle synthétisable il nous faut quelque peu étoffer certains points de notre solution, c'est l'objet du paragraphe suivant.

La façon de le réaliser

Le modèle de commande de feux précédent passe pudiquement sous silence deux aspects :

- Comment réalise-t-on les temporisations `T10`, `T30` et `T60` ? Trois signaux sont créés à cet effet, pilotés par une temporisation qui est réinitialisée quand l'automate est dans l'état `defaut`.

- Comme tout système séquentiel qui se respecte, le fonctionnement de l'automate doit être une machine d'états synchrone ; il faut donc prévoir un signal d'horloge. Pour ne pas compliquer inutilement la solution, nous admettrons qu'une horloge de fréquence 1Hz est disponible.

Le synoptique de la figure II-3 reflète ces modifications ; le diagramme de transitions n'a pas de raison d'être modifié.

Un bloc de temporisation naïf peut être obtenu par un simple compteur, capable de compter 130 périodes d'horloge[4] (somme de tous les temps d'attente), qui est remis à zéro quand l'automate est dans l'état `defaut`.

D'où un programme VHDL synthétisable :

```
-- feu_synt.vhd
-- exemple synthétisable
entity feux is
   port ( hor, detect : in bit ;
      Rp, Op, Vp, Rs, Os, Vs : out bit );
end feux ;

architecture synthese of feux is
   type etat_feu is (defaut, prior, sort1,
      sort2, second, retour1, retour2);
   signal T10, T30, T60 : bit ;
   signal temps : integer range 0 to 129 ;
   signal automate : etat_feu ;
begin
sorties : process
```

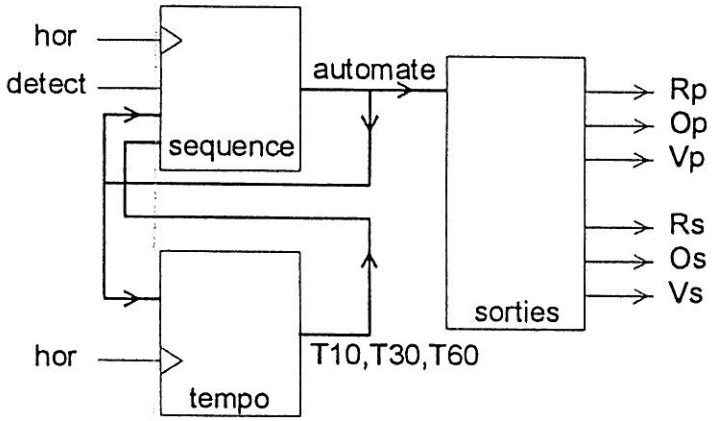


Figure II-3 : *Commande de feux plus réaliste*

4. Solution évidemment stupide si on se pose la question du nombre de bascules utilisées, mais ce n'est pas notre propos ici.


```
   begin
      Rp <= '0' ; Op <= '0' ; Vp <= '0' ;
      Rs <= '0' ; Os <= '0' ; Vs <= '0' ;
      case automate is
         when defaut  => Vp <= '1' ; Rs <= '1' ;
         when sort1   => Op <= '1' ; Rs <= '1' ;
         when sort2   => Rp <= '1' ; Rs <= '1' ;
         when second  => Rp <= '1' ; Vs <= '1' ;
         when retour1 => Rp <= '1' ; Os <= '1' ;
         when retour2 => Rp <= '1' ; Rs <= '1' ;
         when prior   => Vp <= '1' ; Rs <= '1' ;
      end case ;
         wait on automate ;
   end process sorties ;

   sequence : process
   begin
      wait until hor = '1' ;
      case automate is
         when defaut  => if detect = '1' then
                  automate <= sort1 ;
                  end if ;
         when sort1   => if T10 = '1' then
                  automate <= sort2  ;
                  end if ;
         when sort2   => if T10 = '1' then
                  automate <= second ;
                  end if ;
         when second  => if T30 = '1' then
                  automate <= retour1;
                  end if ;
         when retour1 =>   if T10 = '1' then
                  automate <= retour2;
                  end if ;
         when retour2 => if T10 = '1' then
                  automate <= prior  ;
                  end if ;
         when prior   => if T60 = '1' then
                  automate <= defaut ;
                  end if ;
      end case ;
   end process sequence ;

   tempo : process
   begin
      wait until hor = '1' ;
      if automate = defaut or temps = 129 then
         temps <= 0 ;
```

```
    else
       temps <= temps + 1 ;
    end if ;
end process tempo ;

calcul_Ti : process
begin
    T10 <= '0' ; T30 <= '0' ; T60 <= '0' ;
    case temps is
       when 9   => T10 <= '1' ;
       when 19  => T10 <= '1' ;
       when 49  => T30 <= '1' ;
       when 59  => T10 <= '1' ;
       when 69  => T10 <= '1' ;
       when 129 => T60 <= '1' ;
       when others => null ;
    end case ;
       wait on temps ;
end process calcul_Ti ;

end synthese ;
```

Vu de loin le résultat de simulation est identique au précédent, ce qui est rassurant. En examinant le chronogramme de près, cependant, on constaterait que, quand le signal `detect` est toujours actif, l'état `defaut` de l'automate dure une période d'horloge, au lieu d'un temps nul dans le modèle purement fonctionnel. Cette durée minimum est caractéristique d'un système synchrone.

Concernant la description de l'automate, la différence majeure réside dans le mécanisme d'attente du processus correspondant : seul un événement relatif à l'horloge est susceptible de le faire évoluer. Pourquoi les processus `sorties` et `calcul_Ti` génèrent-ils des fonctions combinatoires, alors que les processus `sequence` et `tempo` génèrent des machines d'états synchrones, est une histoire qui mérite une attention toute particulière, mais n'anticipons pas. Nous reviendrons en détail sur ces remarques importantes.

II.2.2. Le couple entity architecture

Un opérateur élémentaire, un circuit intégré, une carte électronique ou un système complet est complètement défini par des signaux d'entrées et de sorties et par la fonction réalisée de façon interne. Ce double aspect - signaux de communication et fonction - se retrouve à tous les niveaux de la hiérarchie d'une application. L'élément essentiel de toute description en VHDL, nommé *design entity* dans le langage, est formé par le couple *entity - architecture*, qui décrit l'apparence externe d'une unité de conception et son fonctionnement interne.

La boîte noire : une entité

La déclaration d'entité décrit l'interface entre le monde extérieur et une unité de conception : signaux d'entrées et de sorties et, éventuellement, paramètres génériques (i.e. constantes dont la valeur est fixée par l'environnement de l'unité considérée). La déclaration d'entité peut également contenir des instructions concurrentes *passives*, c'est à dire qui ne contiennent aucune affectation de signaux ; de telles instructions ne génèrent pas de circuit lors du processus de synthèse.

Une même entité peut être associée à plusieurs architectures différentes. Elle décrit alors une classe d'unités de conception qui présentent au monde extérieur le même aspect, avec des fonctionnements internes différents. Le même circuit peut, par exemple, être décrit de façon purement fonctionnelle lors de la conception, et de façon structurelle pour contrôler le bon respect des règles temporelles de ses constituants physiques.

Syntaxe

```
entity_declaration ::=
entity identifier is
[ generic ( generic_list ) ; ]
[ port ( port_list ) ; ]
entity_declarative_part
[ begin
entity_statement_part ]
end [ entity ] [ identifier ] ;
```

La zone déclarative (`entity_declarative_part`) et la zone d'instructions (`entity_statement_part`) contiennent des informations qui sont connues de toutes les architectures qui partagent la même entité. Nous n'en détaillerons pas ici les contenus, certains exemples illustreront leur utilisation éventuelle. Notons cependant qu'en ce qui concerne les instructions, seules des instructions « passives », c'est à dire qui ne provoquent aucun changement de signal, sont légales dans l'entité. Il est, par exemple, possible de vérifier à cet endroit que des spécifications électriques concernant les signaux d'entrées, comme une largeur minimum d'impulsion, sont respectées.

Rappelons que, dans ces descriptions syntaxiques, les mots réservés du langage sont indiqués en caractères gras et que les éléments entre crochets ne sont pas obligatoires.

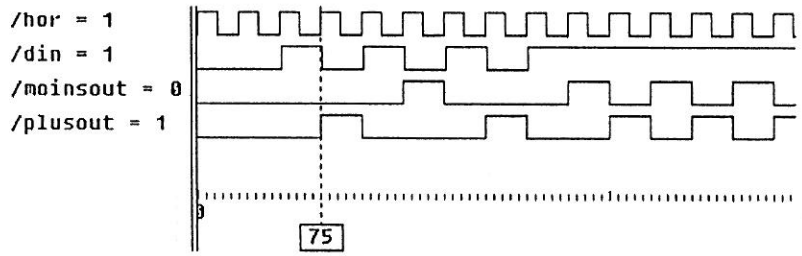
Exemples

Une déclaration d'entité pour un multiplexeur de quatre voies vers une voie

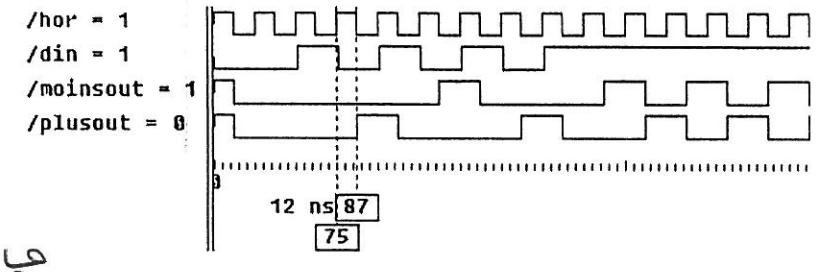
```
entity mux4_1 is
   port(  e0,e1,e2,e3 : in bit ;
          sel : in bit_vector(1 downto 0) ;
         sort : out bit ) ;
end mux4_1 ;
```

```
   moinsout <= etat(1) ;
   encode : process
   begin
      wait until hor = '1' ;
      case etat is
         when mzero =>    if din = '1' then etat <= plus ;
               end if ;
         when pzero =>    if din = '1' then etat <= moins ;
               end if ;
         when moins =>    if din = '1' then etat <= plus ;
               else etat <= mzero ;
               end if ;
         when plus =>    if din = '1' then etat <= moins ;
               else etat <= pzero ;
               end if ;
         when others =>   etat <= mzero ;
      end case ;
   end process encode ;
   end amidir ;
```

amidec ou amidir : simulation fonctionnelle du programme source

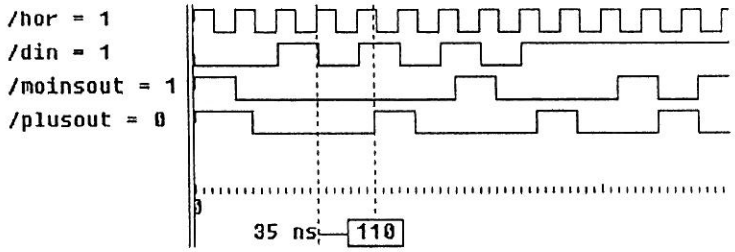


amidir : fonctionnement du circuit



(2) Filtre numérique récursif du 1er ordre

amidec : fonctionnement du circuit

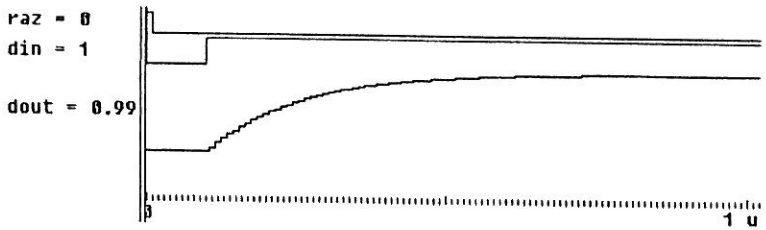


10. Étude d'un filtre numérique

Un filtre numérique récursif du premier ordre peut être modélisé par le programme ci-dessous :

```
entity passe_bas is
   port(hor : in bit ;
        raz : in bit ;
        din : in real ;
       dout : out real := 0.0 ) ;
end passe_bas ;
architecture comporte of passe_bas is
   signal dinret, sort, sortret : real := 0.0 ;
begin
sort <= (30.0 * sortret + din + dinret)/32.0 ;
retards : process
begin
   wait until hor = '1' ;
   if raz = '1' then
      dinret <= 0.0 ;
      sortret <= 0.0 ;
      dout <= 0.0 ;
   else
      dinret <= din ;
      sortret <= sort ;
      dout <= sort ;
   end if ;
end process retards ;
end comporte ;
```

96



Une simulation fonctionnelle fournit la réponse à un échelon de ce filtre :

Dans les courbes précédentes les échelles verticales des signaux d'entrée et de sortie ne sont pas identiques, le gain statique du filtre est égal à 1 (on ne demande pas de le démontrer).

- Ce filtre n'est pas synthétisable dans un circuit programmable de complexité raisonnable. Pourquoi ?
- Une première version synthétisable de ce filtre est donnée par le programme :

```
entity passe_bas is
   port(hor : in bit ;
        raz : in bit ;
        din : in integer range 0 to 255 ;
       dout : out integer range 0 to 255 := 0 ) ;
end passe_bas ;

architecture comporte of passe_bas is
   signal dinret, sort, sortret : integer range 0 to 255
:= 0 ;
begin
sort <= (30 * sortret + din + dinret)/32 ;
retards : process
begin
   wait until hor = '1' ;
   if raz = '1' then
      dinret <= 0 ;
      sortret <= 0 ;
      dout <= 0 ;
   else
      dinret <= din ;
      sortret <= sort ;
      dout <= sort ;
   end if ;
end process retards ;
end comporte ;
```


Combien de bascules doit contenir, au minimum, le circuit cible ?

On applique un échelon d'amplitude 255 à l'entrée du filtre, quelle devrait être, idéalement, la valeur finale de la sortie ?

- La valeur finale atteinte par la version précédente du filtre est 240. Quelle est l'origine de ce dysfonctionnement ? (pour trouver la réponse on pourra, par exemple, se poser la question de la réponse du filtre à un échelon d'amplitude inférieure à 16.)
- Une version corrigée du programme précédent correspond à l'architecture fournie ci-dessous :

```
architecture comporte2 of passe_bas is
   signal dinret : integer range 0 to 255 := 0 ;
   signal sort, sortret : integer range 0 to 8191 := 0 ;
begin
sort <= (15 * sortret)/16 + din + dinret ;
retards : process
begin
   wait until hor = '1' ;
   if raz = '1' then
      dinret <= 0 ;
      sortret <= 0 ;
      dout <= 0 ;
   else
      dinret <= din ;
      sortret <= sort ;
      dout <= (sort + 16)/32 ;
   end if ;
end process retards ;
end comporte2 ;
```

Quelle est l'amélioration ?

- Quel est le coût de cette amélioration en terme de complexité du circuit cible du processus de synthèse ? (justifier cette augmentation de complexité.)
- Une autre version correcte du filtre est celle ci-dessous :

```
architecture comporte3 of passe_bas is
   signal sort, retard : integer range 0 to 8191 := 0 ;
begin
sort <= retard + din ;
retards : process
begin
   wait until hor = '1' ;
   if raz = '1' then
      retard <= 0 ;
      dout <= 0 ;
```

```
    else
       retard <= (15*sort)/16 + din ;
       dout <= (sort + 16)/32 ;
    end if ;
end process retards ;
end comporte3 ;
```

Montrer que la fonction réalisée est la même. Qu'a-t-on gagné ?

- À quelle opération simple correspond une division par 16 ou par 32 ?
- Toutes les versions précédentes utilisent une multiplication, opération complexe. Certains outils de synthèse n'acceptent pas cette opération dans le cas général. Comment pourrait-on contourner cette difficulté ? (On proposera une modification au dernier programme.)
- Le filtre étudié précédemment a une constante de temps liée de façon rigide à la période de son horloge. Comment pourrait on créer un filtre du premier ordre « universel », dont la constante de temps soit paramétrable ? Quel problème cela pose-t-il en synthèse ?

Bibliographie

Méthodes de synthèse et VHDL

R. Airiau, J.M. Bergé, V. Olive, J. Rouillard, *VHDL du langage à la modélisation*, Presses Polytechniques et Universitaires Romandes – 1990.

J. Weber, M. Meaudre, *Circuits numériques et synthèse logique, un outil : VHDL*, Masson – 1995.

Z. Navabi, *VHDL : Analysis and Modeling of Digital Systems*, McGraw Hill – 1993.

K. Skahill, *VHDL for programmable logic*, Addison-Wesley – 1996

R. Lipsett, C. Schaefer, C. Ussery, *VHDL: hardware description and design*, Kluwer Academic – 1989.

J. Armstrong, *Chip Level Modelling in VHDL*, Prentice Hall – 1988.

Introduction to VHDL, Mentor Graphics – 1992.

VHDL Reference Manual, Mentor Graphics – 1994.

AutoLogic VHDL Reference Manual, Mentor Graphics – 1994.

IEEE Standard VHDL Language Reference Manual, Std 1076-1993, IEEE – 1993.

IEEE Standard 1076 VHDL Tutorial, CLSI – 1989.

Librairies des logiciels SYNARIO (DATA IO), V-SYSTEM (Model Tech), WARP (CYPRESS), FPGA EXPRESS (SYNOPSIS).

J.M. Bernard, *Conception structurée des systèmes logiques*, Eyrolles – 1987 .

PAL device Handbook, Advanced Micro Devices – 1988; référence ancienne qui contient une excellente introduction à la synthèse logique.

F.J. Hill, G.R. Peterson, *Computer aided logical design with emphasis on VLSI*, Jonh Wiley & sons – 1993.

C. Mead, L. Conway, *Introduction to VLSI systems*, Addison-Wesley – 1980.

Circuits et opérateurs logiques

Christian Tavernier, *Circuits logiques programmables*, Dunod – 1996

D.A. Hodges, H.G. Jackson, *Analysis and design of digital integrated circuits*, McGraw Hill – 1988.

J. Millman, A. Grabel, *Microelectronics*, McGraw Hill – 1988 ; existe en traduction française chez le même éditeur.

FPGA data book and design guide, ACTEL.

Data Book, ALTERA.

Programmable Logic, Cypress.

Data Book, Handbook, Lattice.

FPGA Applications Handbook, Texas Instrument.

The Programmable Logic Data Book, Xilinx.

ELECTRONIQUE

NUMERIQUE

PROJETS

CORRIGES

PROJETS - CORRIGES

1. Compteur synchrone modulo 16

Voir livre VHDL (Meaudre & all …) page 228 à 232.

2. Commande de feux tricolores pour piétons

Voir livre VHDL (Meaudre & all …) page 14 à 20.

_____

TD 7 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**

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

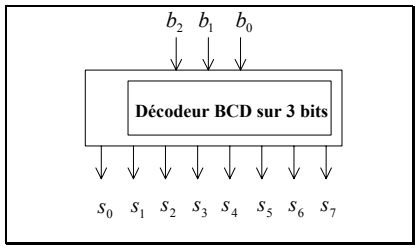
z : a \ bc	00	01	11	10
0	**0**	**1**	**0**	**1**
1	**1**	**0**	**1**	**0**

$$z = a \oplus b \oplus c$$

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**

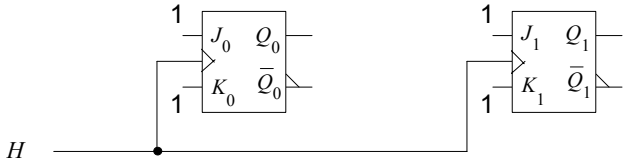
6. Synthèse de Compteur synchrone 2 bits à bascules JK

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

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



ELECTRONIQUE

NUMERIQUE

CORRIGES

ANNEXE

TD5 Corrrigé. MICROPROCESSEUR. MICROCONTRÔLEUR

0. Base Hexadécimale et Binaire

$127_D = 0111\ 1111_B = 7F_H$ ($0111_B = 7_H$ et $1111_B = F_H$)

1. Segment de programme d'addition

Mémoire

Adresse (hexa)	Contenu (hexa)		Contenu (binaire)
0000	86	Instruction 1—LOAD	1000 0110
0001	0A		0000 1010
0002	8B	Instruction 2—ADD	1000 1011
0003	05		0000 0101
0004	8B	Instruction 3—ADD	1000 1011
0005	12		0001 0010
0006	B7	Instruction 4—STORE	1011 0111
0007	20		0010 0000
0008	00		0000 0000
0009	00	END	0000 0000
000A			

Segment de prgm →

Segment de données → Mémoire de données

2000	
2001	
2002	
2003	

← Résultat $33_D = 21_H$ à l'exécution du programme.

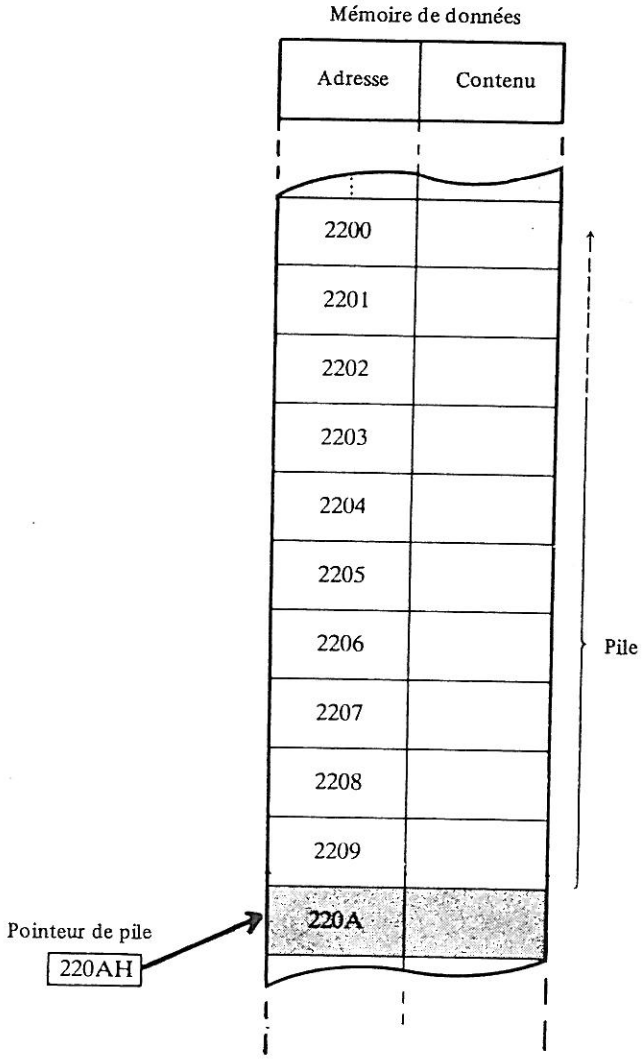
Segment de pile →

Segment de programme pour addition de 10+5+18

2. Opération logique et registre d'état

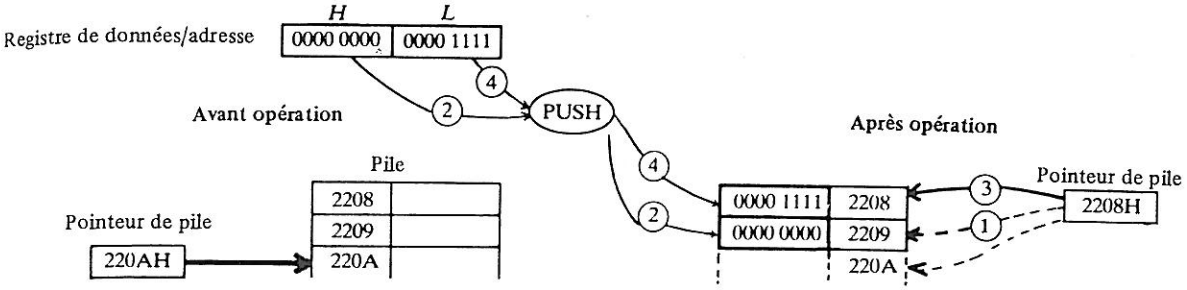
(Accumulateur) = 00_H Z = 1

3. Pile

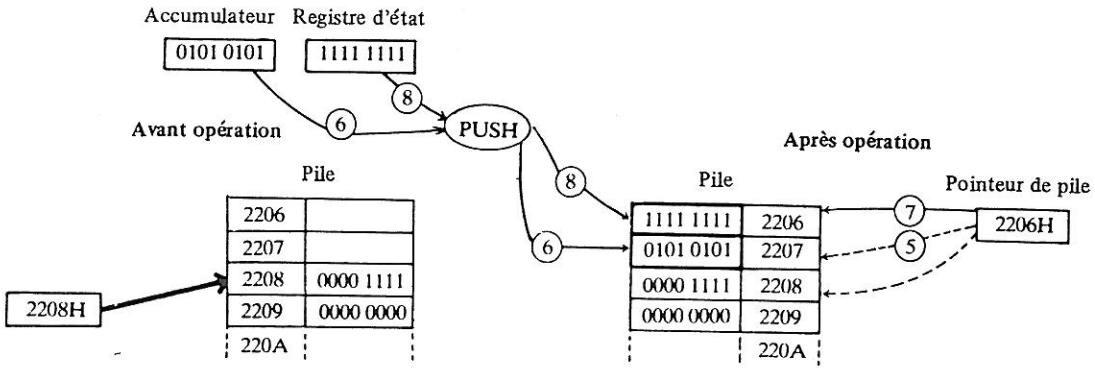


Installation d'une pile en RAM

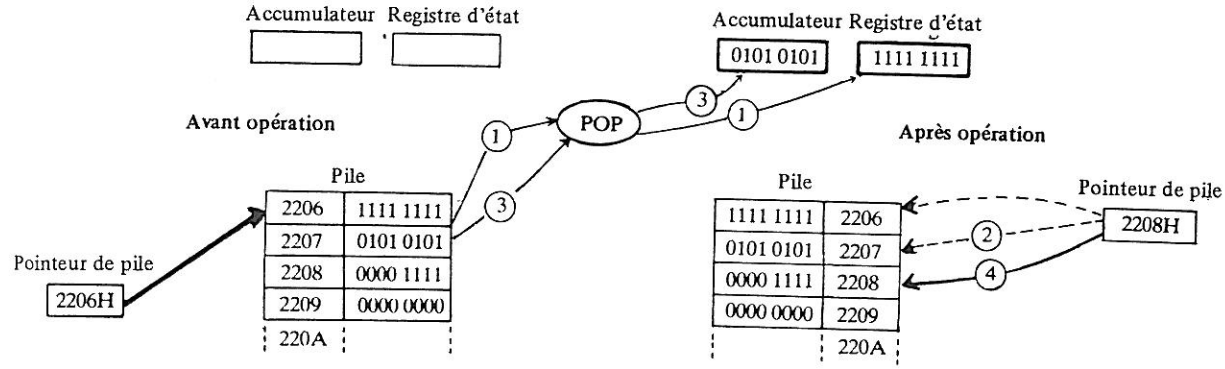
5 4 Ter



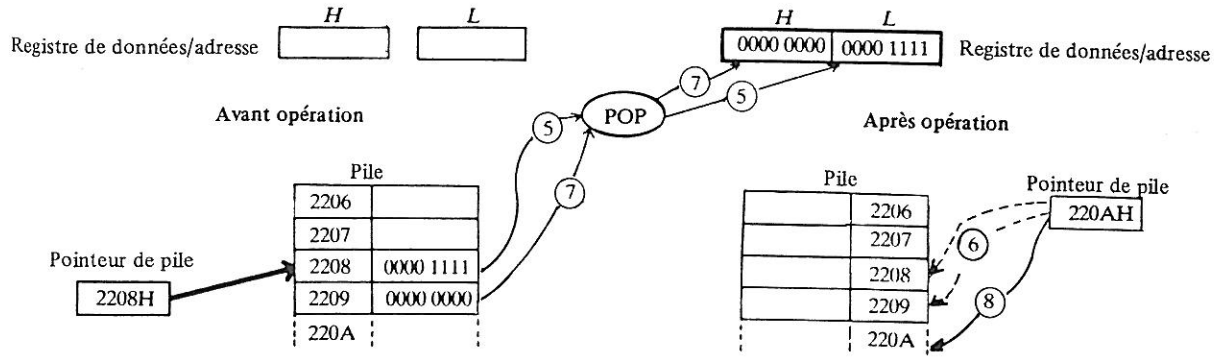
(*a*) Empilement du contenu du registre de données/adresse



(*b*) Empilement de l'accumulateur et du registre d'état



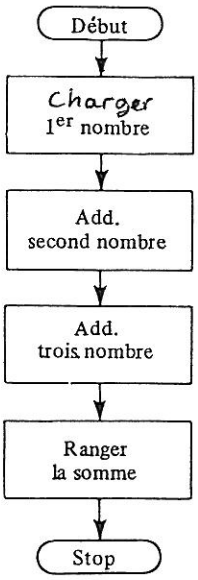
(*c*) Restauration de l'accumulateur et du registre d'état par dépilement



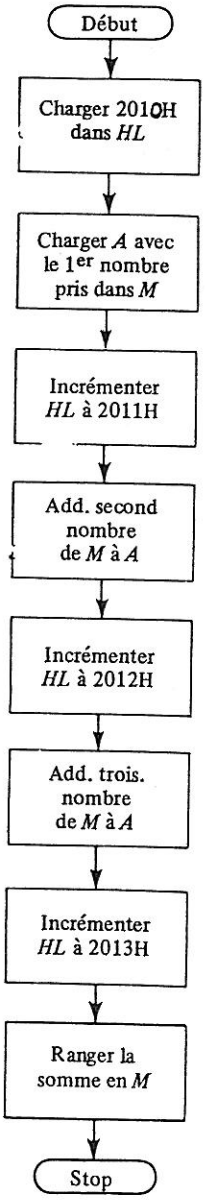
(*d*) Restauration du registre de données/adresse par dépilement

5 5

4. Programmation du microprocesseur (opérations arithmétiques)



Organigramme fonctionnel de l'addition de trois nombres



Organigramme détaillé

UMC : Unité Centrale Microprocesseur

M : Mémoire

Etiquette	Mnémonique	Opérande	Commentaire
	LXI	*H*,2010H	; Charger la paire *HL* de l'adresse 2010H. (La paire *HL* sera utilisée comme pointeur d'adresse)
	MOV	*A*,*M*	; Charger le premier nombre en mémoire 2010H dans l'accumulateur
	INX	*H*	; Incrémenter la paire *HL* à 2011H
	ADD	*M*	; Add. le second nombre en mémoire 2011H à l'accumulateur
	INX	*H*	; Incrémenter la paire *HL* à 2012H
	ADD	*M*	; Add. le troisième nombre en mémoire 2012H à l'accumulateur
	INX	*H*	; Incrémenter la paire *HL* à 2013H
	MOV	*M*,*A*	; Ranger la somme contenue dans l'accumulateur en mémoire 2013H
	HLT		; Stopper l'UCM

(*a*) Programme en assembleur pour additionner 3 nombres et ranger leur somme

Adresse (hexa)	Contenu (hexa)	Etiquette	Mnémonique	Opérande	Commentaire
2020	21		LXI	*H*,2010H	; Charger l'adresse 2010H dans la paire *HL* qui sera utilisée comme pointeur d'adresse
2021	10				
2022	20				
2023	7E		MOV	*A*,*M*	; Charger le premier nombre en mémoire 2010H dans l'accumulateur
2024	23		INX	*H*	; Incrémenter la paire *HL* à 2011H
2025	86		ADD	*M*	; Add. le second nombre en mémoire 2011H à l'accumulateur
2026	23		INX	*H*	; Incrémenter la paire *HL* à 2012H
2027	86		ADD	*M*	; Add. le troisième nombre en mémoire 2012H à l'accumulateur
2028	23		INX	*H*	; Incrémenter la paire *HL* à 2013H
2029	77		MOV	*M*,*A*	; Ranger la somme contenue dans l'accumulateur en mémoire 2013H
202A	76		HLT		; Stopper l'UCM

Programme d'addition et de rangement en code machine et en assembleur

Mémoire de données

Adresse (hexa)	Contenu (hexa)	
2010	0F	
2011	09	
2012	06	
2013		← Somme

Mémoire de programme

2020	21	Instruction 1— Charger *HL*
2021	10	
2022	20	
2023	7E	Instruction 2—Charger 1er nombre dans *A*
2024	23	Instruction 3— Incrémenter *HL*
2025	86	Instruction 4— Add. second nombre
2026	23	Instruction 5— Incrémenter *HL*
2027	86	Instruction 6— Add. troisième nombre
2028	23	Instruction 7— Incrémenter *HL*
2029	77	Instruction 8— Ranger la somme
202A	76	Instruction 9— Stop

Image mémoire et instructions pour l'addition de 3 nombres et le rangement de la somme

5. Programmation du microprocesseur (branchements)

Etiquette	Mnémonique	Opérande	Commentaires
	MVI	A,0FH	; Charger le premier nombre (15_{10}) dans l'accumulateur
	MVI	L,06H	; Charger le second nombre (6_{10}) dans le registre L
	CMP	L	; Comparer (A) et (L) Indicateur $CY = 1$ si $A < L$
	JC	STORE L	; Sauter à STORE L si $CY = 1$ (si $A < L$) ; sinon continuer par l'instruction suivante
	STA	2040H	; Ranger (A) en mémoire 2040H
	HLT		; Stopper l'UCM après rangement de (A)
STORE L	MOV	A,L	; Transférer (L) à l'accumulateur
	STA	2040H	; Ranger (A) en mémoire 2040H
	HLT		; Stopper l'UCM après rangement de (L)

Programme en assembleur utilisant la branche "rangement de A"

Etiquette	Mnémonique	Opérande	Commentaires
	MVI	A,0AH	; Charger le premier nombre (10_{10}) dans l'accumulateur
	MVI	L,0EH	; Charger le second nombre (14_{10}) dans le registre L
	CMP	L	; Comparer (A) et (L) Indicateur $CY = 1$ si $A < L$
	JC	STORE L	; Sauter à STORE L si $CY = 1$ (si $A < L$) ; sinon passer à l'instruction en séquence
	STA	2040H	; Ranger (A) en mémoire 2040H
	HLT		; Stopper l'UCM après rangement de (A)
STORE L	MOV	A,L	; Transferer (L) à l'accumulateur
	STA	2040H	; Ranger (A) en mémoire 2040H
	HLT		; Stopper l'UCM après rangement de (L)

Programme en assembleur utilisant la branche "rangement de L"

6. Programmation du microprocesseur (boucles)

Etiquette	Mnémonique	Opérande	Commentaire
	LXI	*H*,2040	; Charger 2040H dans la paire *HL* qui sera utilisée comme pointeur d'adresse
	XRA	*A*	; RAZ l'accumulateur à 00H [$(A) \oplus (A) = 00H$ dans l'accumulateur]
LOOP	MOV	*M*,*A*	; Ranger le contenu de l'accumulateur en mémoire pointée par la paire *HL*
	INX	*H*	; Incrémenter la paire *HL*
	INR	*A*	; Incrémenter l'accumulateur
	CPI	09H	; Comparer $-(A) = 09H$? Si $(A) = 09H$, l'indicateur de zéro est initialisé à 1
	JNZ	LOOP	; Sauter à LOOP si $Z = 0$ [si $(A) < 09H$]; sinon continuer en séquence
	HLT		; Stopper l'UCM

Version en assembleur du programme

7. Programmation du microprocesseur (sous-programme)

Etiquette	Mnémonique	Opérande	Commentaire
	LXI	*SP*,20C0H	; Charger 20C0H dans le pointeur de pile
START	LXI	*H*,2040H	; Charger 2040H dans la paire *HL* qui servira de pointeur d'adresse du programme principal
	XRA	*A*	; RAZ accumulateur à 00H
	ADD	*M*	: Add. (*A*) à la mémoire 2040H (le premier nombre à ajouter est en 2040H)
	INX	*H*	; Incrémenter la paire *HL* à 2041H
	ADD	*M*	; Add. (*A*) à la mémoire 2041H (le second nombre à ajouter est en 2041H)
	PUSH	*PSW*	; Empiler (*A*) et les indicateurs
	CALL	MULTIPLY	; Appeler le S.P. MULTIPLY en mémoire 2050H
	INX	*H*	; Incrémenter la paire *HL* à 2042H
	MOV	*M,A*	; Ranger le produit en mémoire 2042H
	POP	*PSW*	, Dépiler et restaurer (*A*) et les indicateurs
	CPI	10H	; Comparer (*A*) et 10H [(*A*) − 10H], si (*A*) < 10H, *CY* = 1 ; sinon *CY* = 0
	JC	START	; Sauter à START (mémoire 2003H) si *CY* = 1 ; sinon continuer en séquence
	HLT		; Stopper l'UCM

Programme en assembleur relatif au problème

Etiquette	Mnémonique	Opérande	Commentaires
	PUSH	*H*	; Empiler les contenus des registres *H* et *L* pour sauvegarder le contenu de la paire *HL*
	MVI	*L*,02H	; Charger 02H (facteur d'échelle) dans *L* ; 02H est le *multiplicateur*
	MOV	*H,A*	; Transférer (*A*) → (*H*) ; contenu de *H* = *multiplicande* sauvegarder la somme dans *H*
	XRA	*A*	; RAZ *A*
LOOP	ADD	*H*	; Add. (*H*) + (*A*) ; ranger la somme dans *A*
	STA	2020H	; Ranger (*A*) en mémoire 2020H (2020H sert de mémoire temporaire)
	MOV	*A,L*	; Tranférer (*L*) en *A*
	DCR	*A*	; Décrémenter le registre *A*
	MOV	*L,A*	; Transférer (*A*) en *L*
	LDA	2020H	; Charger le contenu de la mémoire temporaire 2020H en *A* (restauration de *A* à partir de la mémoire 2020H)
	JZ	DONE	; Sauter en DONE (mémoire 2065H) si *Z* = 1 ou si (*L*) est décrémenté à 00H ; sinon continuer en séquence
	JMP	LOOP	; Sauter toujours à LOOP (adresse 2055)
DONE	POP	*H*	; Dépiler et restaurer le contenu de la paire *HL*
	RET		; Retourner au programme principal

Programme en assembleur relatif à l'organigramme
réalisant le sous-programme MULTIPLY
C/a multiplication est réalisée comme un cumul de sommes)