

# **- TP 3 : première partie - Codage de canal – Code détecteur et correcteur d'erreur**

*par Édouard Lumet & [REDACTED]*

4,5/5

*très bon compte rendu !!*

## Sommaire

Introduction.....	3
2. Rappels de logiques combinatoire et séquentielle.....	4
2.1. A la découverte de fonctions logiques combinatoires élémentaires.....	4
2.2. Synthèse d'une première fonction logique combinatoire : le multiplexeur.....	4
2.3. Un premier pas vers la mémorisation : la bascule.....	6
2.4. La mémorisation d'un mot : le registre.....	6
2.5. En route pour le registre à décalage.....	7
3. Réalisation d'une liaison série.....	9
3.1. Réalisation du convertisseur parallèle/série.....	9
3.2. Réalisation du convertisseur série/parallèle.....	10
3.3. Test de la liaison série.....	11
Conclusion.....	13

## Introduction

Dans ce troisième TP de TELC04, nous abordons une étape préparant à la transmission à travers un canal qui est le codage de canal. Cette étape a pour but de protéger les données qui peuvent subir des erreurs dues à la nature du canal et au bruit s'y appliquant. Pour ce faire, on ajoute à ces données des bits de redondance permettant de détecter et éventuellement de corriger un nombre plus ou moins important d'erreur selon le code appliqué.

Afin de comprendre ce processus, nous allons à l'aide de LabView analyser et appréhender quelques fonctions logiques combinatoires et séquentielles qui sont à la base d'un dispositif de codage de canal. Nous reverrons les notions de logiques élémentaires telles que les fonctions 'ET', 'OU', 'OU exclusif', 'NON' puis le fonctionnement de bascules et registres dans différentes situations si l'on transmet une information en série ou en parallèle.

*bien*

## 2. Rappels de logiques combinatoire et séquentielle

C'est dans cette première partie que nous allons faire quelques révisions sur les fonctions logiques élémentaires ainsi que les registres et les registres à décalage que nous avons déjà manipulé dans le module d'ARCHI1, en programmation assembleur sur Raspberry Pi.

### 2.1. A la découverte de fonctions logiques combinatoires élémentaires

- Le VI *inconnue.vi* comporte trois fonctions logiques combinatoires élémentaires. Les tables de vérité des fonctions sont les suivantes :

A	B	S
0	0	0
1	0	0
0	1	0
1	1	1

A1	B1	S1
0	0	0
1	0	1
0	1	1
1	1	1

A2	B2	S2
0	0	0
1	0	1
0	1	1
1	1	0

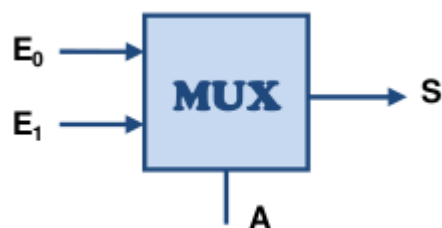
- Les équations correspondant aux tables de vérité précédentes ainsi que les noms des fonctions sont les suivants :
  - $S = A.B$   $\Rightarrow$  fonction **ET (AND)**
  - $S1 = A1+B1$   $\Rightarrow$  fonction **OU (OR)**
  - $S2 = A2 \oplus B2$   $\Rightarrow$  fonction **OU exclusif (XOR)**

NB : ~~la fonction ET est l'opérateur d'identité, la sortie (S) est active que lorsque les entrées (A et B) sont égales.~~ La fonction OU exclusif est l'opérateur de différence, la sortie (S2) est active que lorsque que les entrées (A2 et B2) sont strictement différentes.

*non, il faut que  $a=1$  et  $b=1$ , ce qui est différent de la fonction identité où il faut que  $a=0$  et  $b=0$  ou  $a=1$  et  $b=1$  !!*

*c'est la fonction xnor (ou exclusif inversé) qui représente la fonction d'identité*

### 2.2. Synthèse d'une première fonction logique combinatoire : le multiplexeur



Multiplexeur (MUX) 2 voies vers 1

Si  $A = 0$  alors  $S = E_0$   
 Si  $A = 1$  alors  $S = E_1$

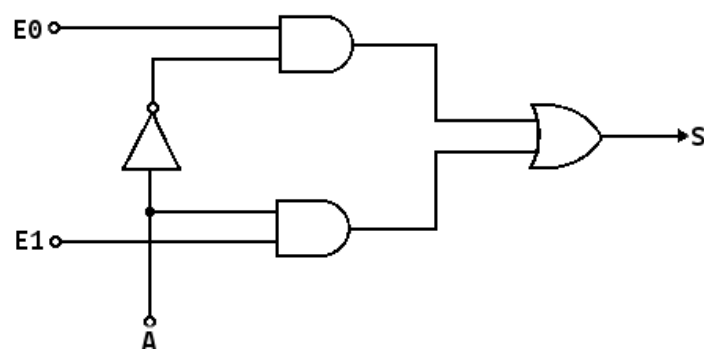
- D'après l'image en bas de page précédente, on détermine la table de vérité du multiplexeur 2 voies vers 1. C'est une sorte d'aiguillage dont l'entrée A est le « sélecteur » permettant de recopier en sortie la première ou la seconde entrée de données.

A	E <sub>1</sub>	E <sub>0</sub>	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

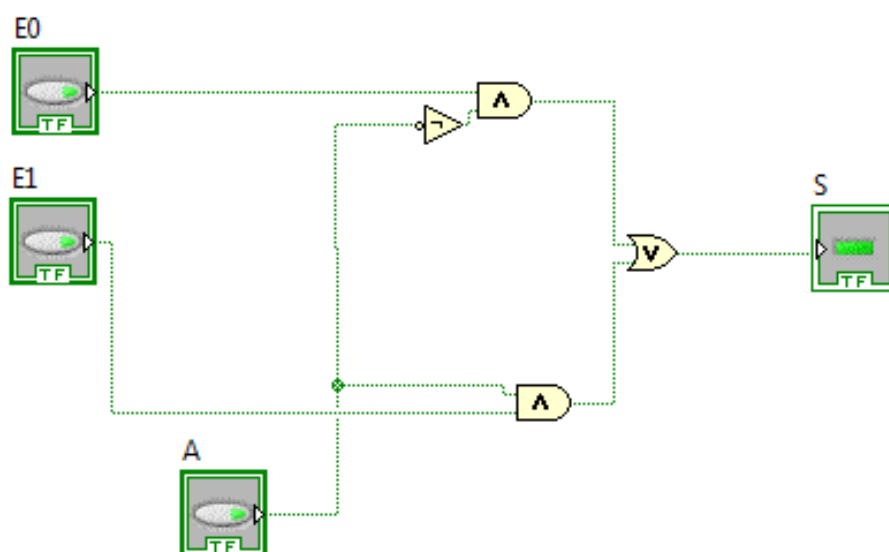
Table de vérité sur MUX 2 vers 1

En observant la table de vérité et d'après l'information donnée avec le schéma du multiplexeur, on sait que les données en E<sub>0</sub> sont recopiées sur la sortie quand A est désactivé (0) et que les données en E<sub>1</sub> sont recopiées sur la sortie quand A est activé (1). On en déduit :  $S = \bar{A}.E_0 + A.E_1$

Le logigramme de ce multiplexeur est le suivant :



- On reproduit le logigramme déterminé ci-dessus sous LabView, dans un VI nommé *MUX 2 vers 1.vi*.
- Le VI que nous utiliserons ainsi comme sous-VI dans la suite de ce TP est le suivant :



tb

### 2.3. Un premier pas vers la mémorisation : la bascule

Le VI *bascule D0.vi* réalise une bascule D qui est un élément mémorisant un bit. Utilisé dans la logique séquentielle, la sortie dépend de l'entrée mais également de l'état antérieur.

- La table de vérité de la bascule est la suivante :

H	D	$Q_n$
Front montant	0	<b>0</b>
Front montant	1	<b>1</b>
Autre événement	X	$Q_{n-1}$

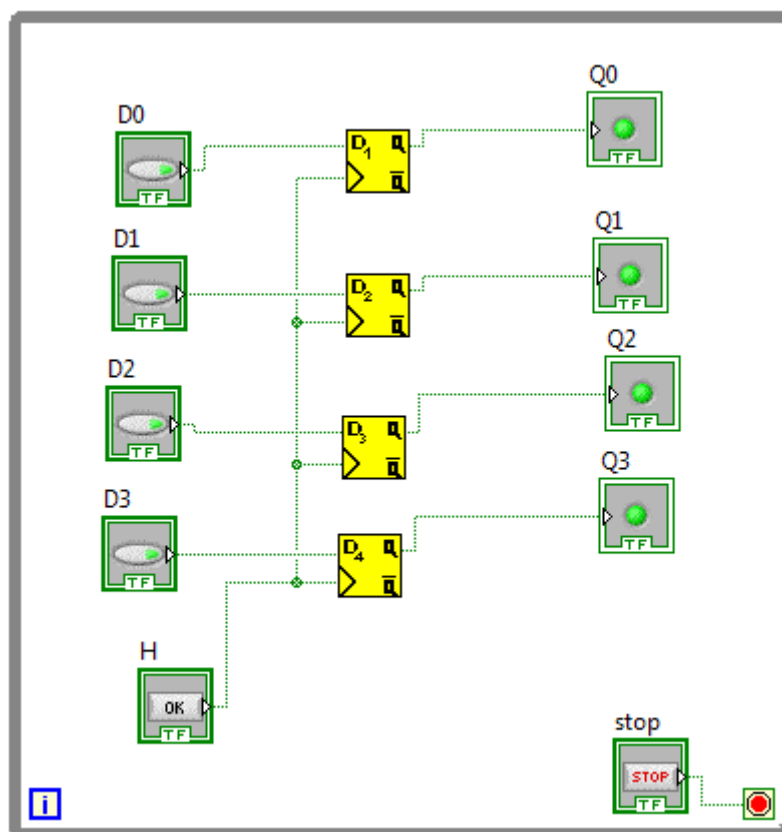
L'événement qui rend l'horloge active est un front montant. Lors de cet événement, l'entrée est directement recopiée sur la sortie alors que lors de tout autre événement, l'état antérieur est alors mémorisé.

*tb*

### 2.4. La mémorisation d'un mot : le registre

Le registre est un ensemble de bascule D, comme vue ci-dessus, permettant de mémoriser un mot de n bits pour n bascules.

- On peut alors créer un VI *registre.vi* permettant de stocker des mots de 4 bits à l'aide de quatre bascules D que l'on trouve dans le répertoire TP3/SousVI fourni.



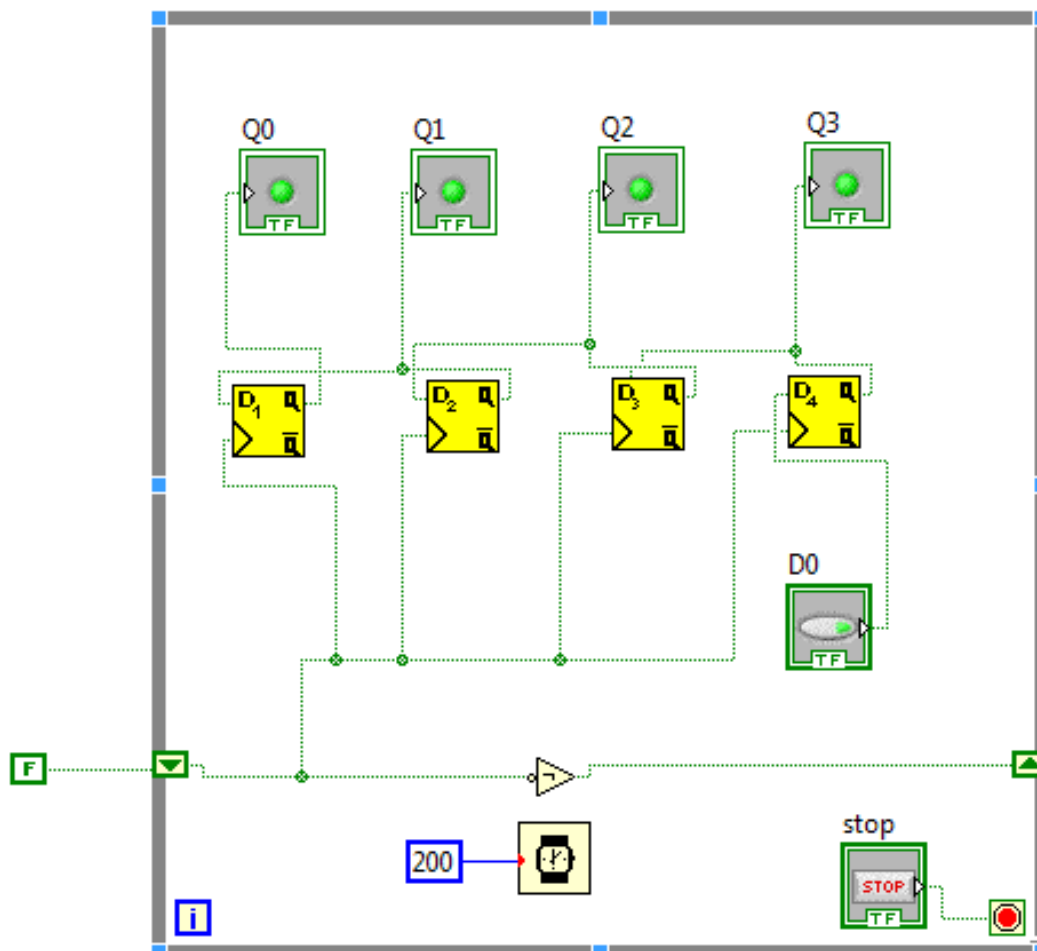
*tb*

## 2.5. En route pour le registre à décalage

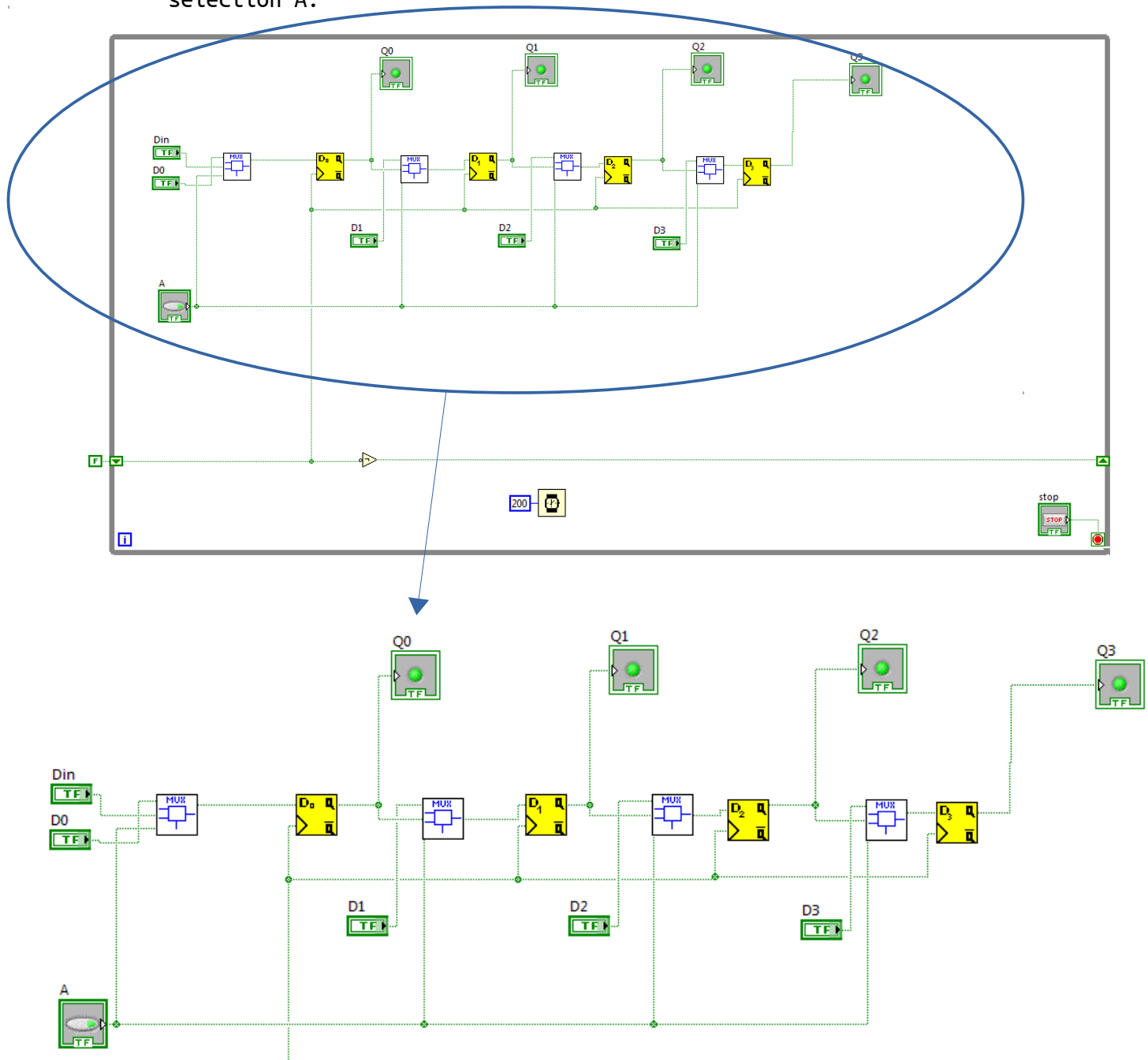
Un registre à décalage est un registre comme nous venons de créer et qui permet de décaler l'information vers la droite si l'on décale les bits de poids fort vers les bits de poids faible ou vers la gauche si l'on décale les bits de poids faible vers les bits de poids fort. C'est le signal d'horloge qui cadence ce décalage.

- On crée un VI *registre-decalage.vi* d'après le VI *registre.vi* précédent comportant une entrée  $D_{in}$ , quatre sorties parallèles  $Q_0$  à  $Q_3$  et qui effectue un décalage vers la gauche. Sur chaque événement « front montant » de l'horloge, on insère un bit de l'entrée  $D_{in}$  puis on décale le tout vers la gauche (les poids faibles vers les poids forts).

*tb*



- En s'appuyant sur le VI créé précédemment et le multiplexeur 2 voies vers 1 créé dans la partie II.1, on complète le VI *reg\_g.vi* qui est un registre comprenant deux modes différents : soit on le charge en parallèle via les 4 entrées  $D_0$  à  $D_3$  dont les données seront ensuite directement copiées en sortie, soit on le charge en série via l'entrée  $D_{in}$  dont les bits seront copiés sur les quatre sorties à la manière du registre à décalage vers la gauche. La sélection des modes s'effectue à l'aide de l'entrée de sélection  $A$ .



*ok - expliquez votre raisonnement pour arriver à la synthèse précédente (comment câble-t-on le multiplexeur ?)*



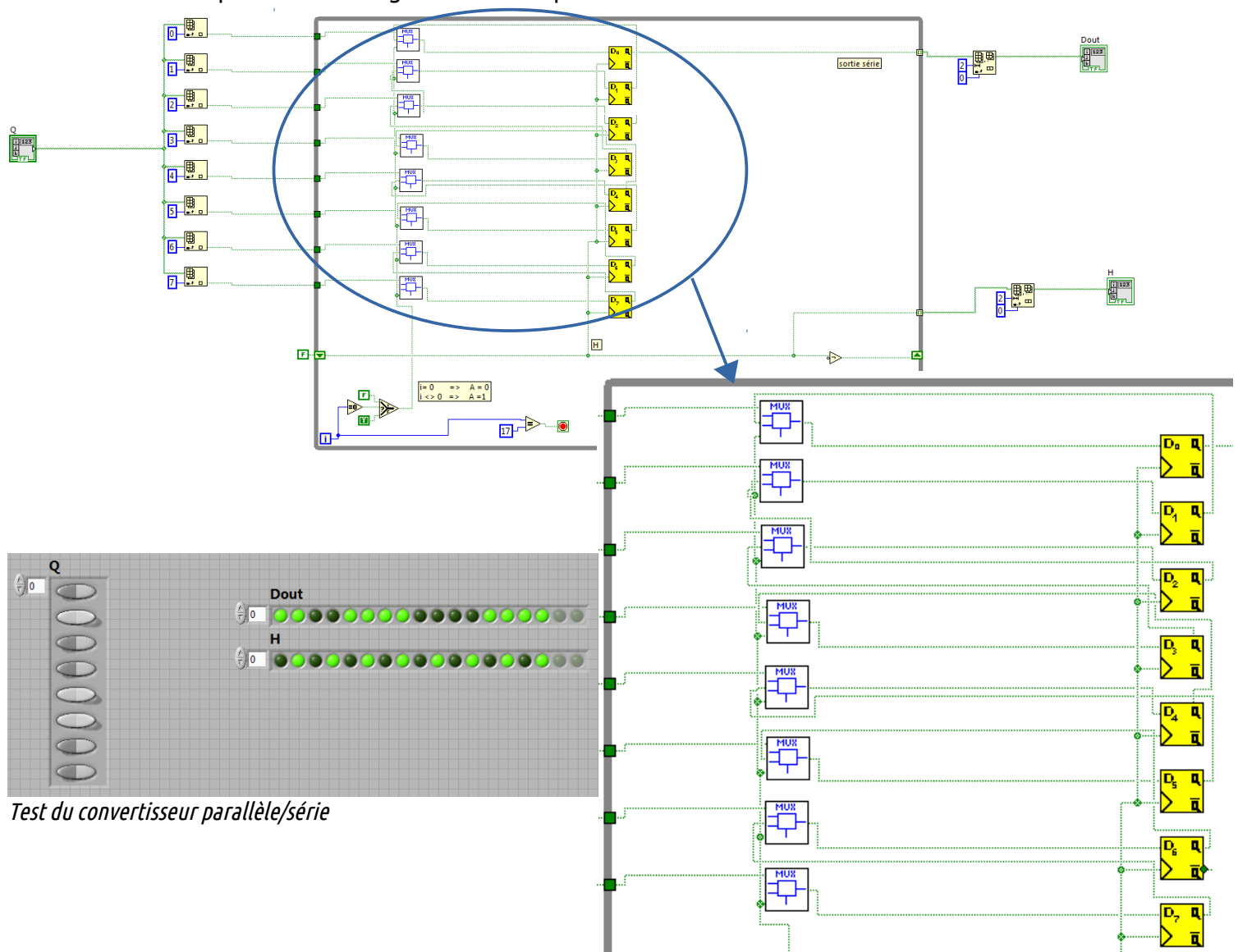
### 3. Réalisation d'une liaison série

Une liaison série repose sur un support de transmission unique entre deux équipements numériques où les bits de données sont envoyés les uns à la suite des autres au rythme d'une horloge identique pour les deux équipements (synchrone). C'est ce type de liaison que nous allons étudier et réaliser.

#### 3.1. Réalisation du convertisseur parallèle/série

La première étape d'une transmission série est la conversion parallèle/série. En effet, le processeur traite des données en parallèle, il faut donc réaliser cette conversion afin de transmettre les données en série sur le canal.

- Le VI *PISO.vi* permet de réaliser cette conversion parallèle/série. On complète son diagramme afin qu'il soit fonctionnel.

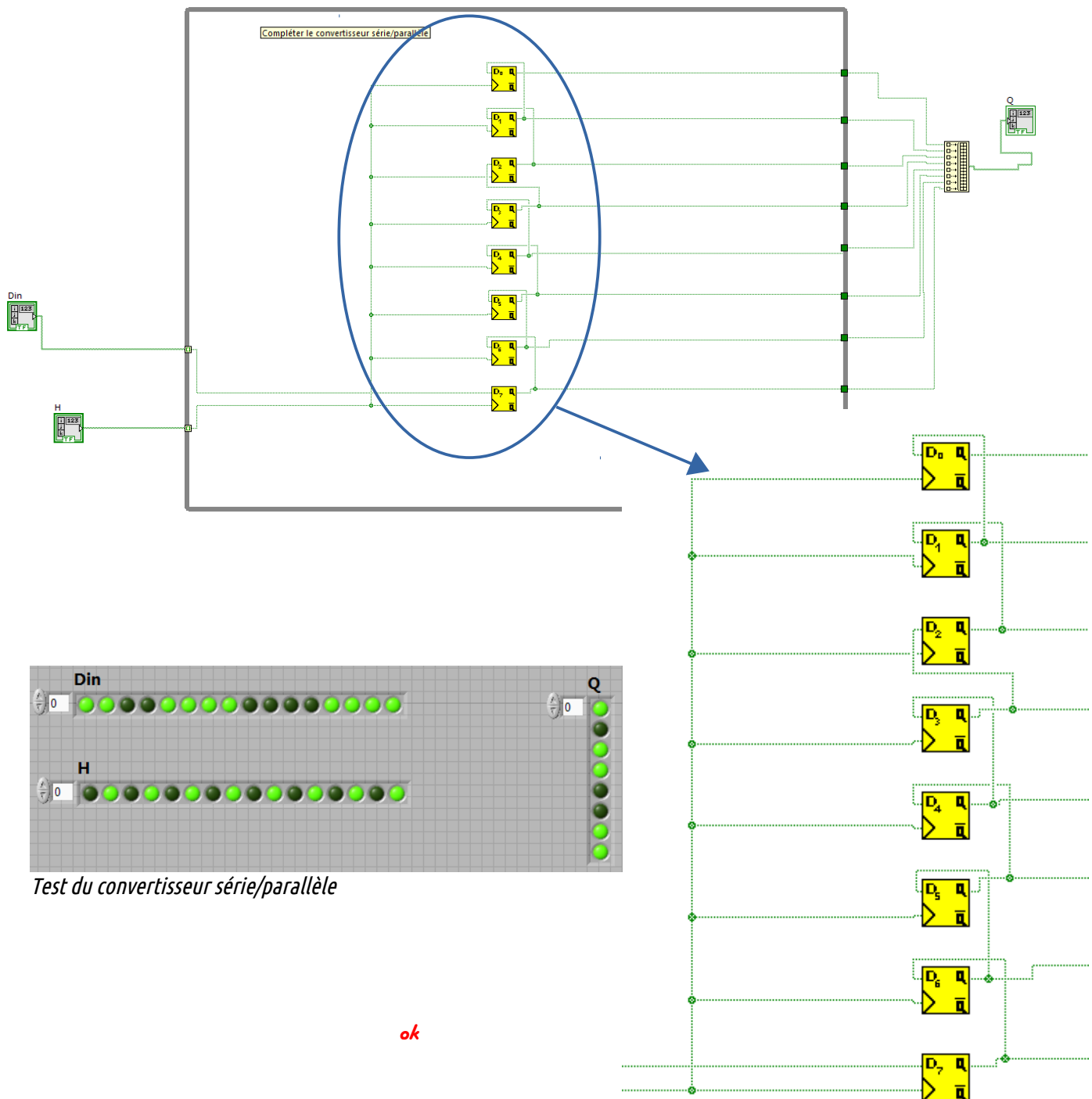


Test du convertisseur parallèle/série

### 3.2. Réalisation du convertisseur série/parallèle

La convertisseur série/parallèle est quant à lui la dernière étape de la transmission série, il permet de fournir les données au processeur sous la forme adéquate.

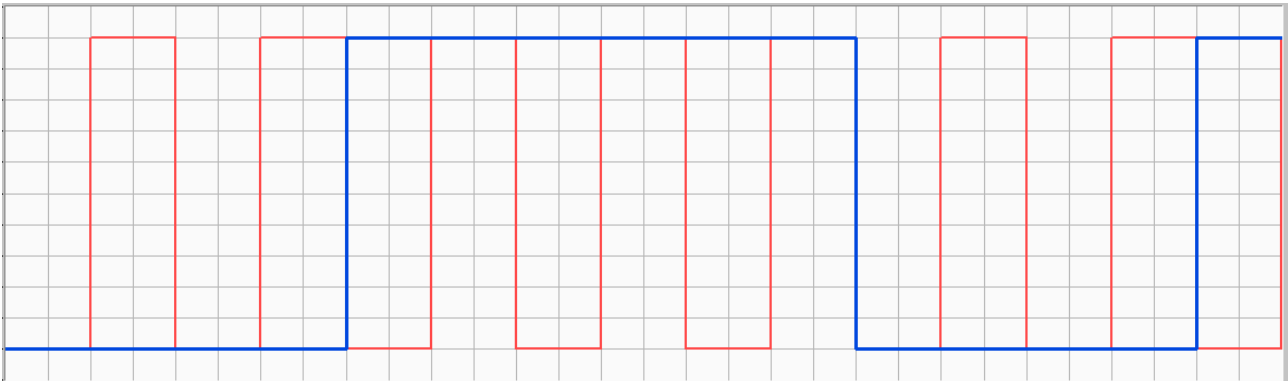
- Comme pour le VI précédent, on complète le diagramme du VI *SIPO.vi* afin qu'il effectue la conversion série/parallèle correctement.



### 3.3. Test de la liaison série

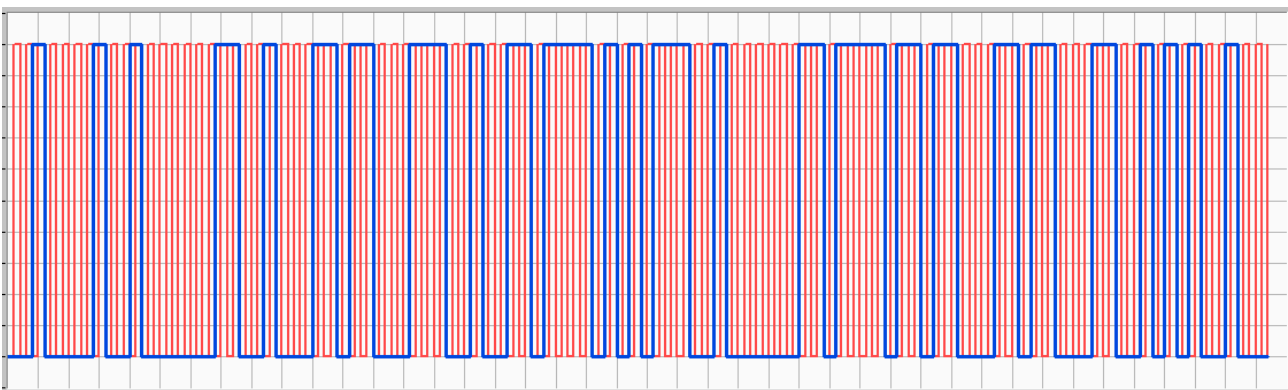
Maintenant que l'intégralité de la liaison est complète et configurée, on teste celle-ci en paramétrant notamment le TEB (taux d'erreur binaire, en nombre de bits erronés par seconde).

- On saisit le caractère « 9 » avec un TEB de 0 puis on exécute le VI. Les signaux transmis sont les suivants :



Le code ASCII du symbole « 9 » est 57 soit en binaire 111001. La séquence de 8 bits est donc « 00111001 », que l'on reconnaît ci-dessus, le bit 0 étant codé par un niveau bas (0V) et le bit 1 étant codé par un niveau haut (1V). L'horloge est le signal carré rouge.

- On saisit ensuite la chaîne « Hello World ! » toujours avec un TEB de 0. La durée de transmission est plus longue mais la chaîne reçue est intacte. Les signaux sont les suivants :



- On réitère l'envoi précédent en faisant varier le TEB, les résultats sont résumés dans ce tableau :

TEB	Qualité transmission
0	Excellente : « Hello World ! »
0,1	Médiocre : « Hglio GnR,d a »
0,01	Moyenne (parfois sans erreur) : « Hello Sorlf ! »
0,001	Excellente : « Hello World ! »
0,0001	

- On effectue les mêmes tests mais cette fois-ci avec une image à l'aide du VI *liaison série (image).vi* :

TEB	Qualité transmission
0	Excellente
0,1	Médiocre, près de la moitié des pixels sont manquants
0,01	Moyenne, plusieurs pixels manquent
0,001	Assez bonne, quelques pixels seulement sont manquants
0,0001	Bonne, deux ou trois pixels sont manquants



TEB = 0,1 (1 erreur / 10 bits)



TEB = 0,001 (1 erreur / 1000 bits)



TEB = 0,01 (1 erreur / 100 bits)



TEB = 0,0001 (1 erreur / 10000 bits)

- Pour améliorer la qualité de la transmission, il faut ajouter un codage de canal qui ajoutera de la redondance à notre information afin de résister au bruit s'appliquant sur le canal.

## Conclusion

Nous avons appris dans cette première partie à créer sous LabView des circuits composés de fonctions logiques combinatoires et séquentielles. Nous avons notamment créé un multiplexeur, des registres basiques ou à décalage en manipulant des bascules.

Pour finir, après avoir complété la liaison série à l'aide des éléments créés précédemment, nous nous sommes rapidement aperçu que sans protection, les données transmises sont très sensibles au bruit, altérant l'information émise. C'est ainsi que nous découvrirons dans la deuxième partie de ce TP, la mise en place d'un codage de canal pour ajouter de la redondance à notre information et d'un code détecteur et correcteur d'erreur en réception pour détecter et corriger les informations afin de la retrouver telle qu'elle a été - probablement - transmise.

*ok*