

# Rapport Intermediaire - Conception Circuit Numérique

Davy CLARK & Nolan BUCHET

Septembre 2025

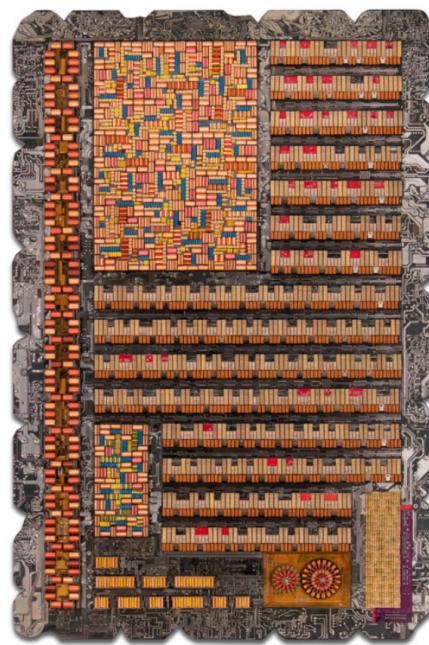


POLYTECH<sup>®</sup>  
NANTES



Pôle Sciences et technologie  
Nantes Université

## Recepteur de Transmission LIN



## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Protocole LIN</b>	<b>4</b>
<b>3</b>	<b>Cahier des Charges</b>	<b>6</b>
<b>4</b>	<b>Description des différentes spécifications définies en travaux dirigés</b>	<b>9</b>
4.1	Interface Microprocesseur . . . . .	9
4.2	Reception Trame . . . . .	10
4.3	FIFO . . . . .	10
4.4	ETAT . . . . .	10
<b>5</b>	<b>Description et justification de la structure fonctionnelle</b>	<b>11</b>
<b>6</b>	<b>Description et justification de la solution architecturelle obtenue pour le circuit</b>	<b>15</b>
6.1	Horloge . . . . .	15
6.2	Architecture Reception Trame . . . . .	15
6.3	Architecture Interface MicroProcesseur . . . . .	19
6.4	Architecture FIFO . . . . .	21
6.5	Implémentation de l'État Interne . . . . .	22
<b>7</b>	<b>Présentation du fonctionnement des fonctions</b>	<b>24</b>
7.1	Interface MicroProcesseur . . . . .	24
7.1.1	Réseau Combinatoire d'Entrée . . . . .	24
7.1.2	Réseau Combinatoire de Sortie . . . . .	24
7.1.3	Registres et Machine à États . . . . .	25
7.1.4	Réseau Synchronisé de Sortie . . . . .	26
<b>8</b>	<b>Simulation des fonctions</b>	<b>27</b>
8.1	Interface Microprocesseur . . . . .	27
8.1.1	Déclarations et signaux . . . . .	27
8.1.2	Instanciation du composant testé . . . . .	27
8.1.3	Environnement de test . . . . .	27
8.1.4	Stimuli supplémentaires . . . . .	28
8.1.5	Analyse du chronogramme de simulation . . . . .	28
<b>9</b>	<b>Synthèse des fonctions</b>	<b>30</b>
9.1	Interface Microprocesseur . . . . .	30
<b>10</b>	<b>Routages des Fonctions</b>	<b>33</b>
10.1	Interface Microprocesseur . . . . .	33
<b>11</b>	<b>Conclusion</b>	<b>37</b>
<b>12</b>	<b>Annexes</b>	<b>38</b>
12.1	Testbench InterfaceMicroporcesseur . . . . .	38

---

## 1 Introduction

Le projet réalisé dans le cadre de l'enseignement de Conception de Circuits numériques a pour objectif de développer des compétences essentielles à la conception de systèmes embarqués, notamment la mise au point d'un circuit utilisant un composant logique programmable.

L'architecture électronique d'un véhicule repose sur une organisation de calculateurs distribués. L'exemple retenu s'inspire du fonctionnement d'un calculateur embarqué dans la portière d'une automobile, chargé de la gestion des rétroviseurs et des vitres électriques.

Dans ce contexte, deux sous-ensembles sont distingués :

- un sous-ensemble de supervision, qui génère les commandes pour les moteurs des rétroviseurs et des vitres électriques,
- un sous-ensemble d'interface, assurant la communication entre le sous-ensemble de supervision et les autres calculateurs du véhicule.

Ce rapport se concentre exclusivement sur ce second sous-ensemble, l'interface microprocesseur, afin d'étudier son rôle et sa conception.

L'un des objectifs principaux est d'appréhender la conception du circuit via la méthode MCSE (Méthode de Conception de Systèmes Électroniques), en mettant l'accent sur les étapes de spécifications et de conception. Le déroulement du rapport suit la logique du diagramme en Y.

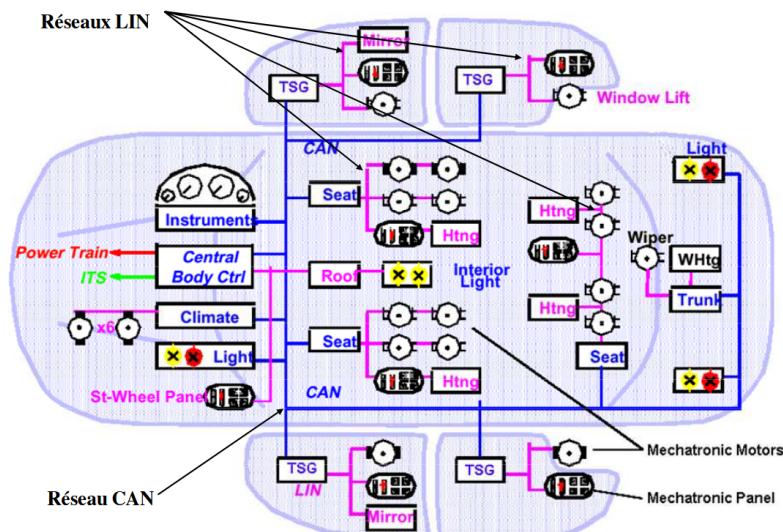


FIGURE 1 – Exemple d'architecture d'un réseau dans un véhicule

## 2 Protocole LIN

### Architecture

Le bus LIN est un système *mono-maître et multi-esclaves*. Un seul maître initie toutes les communications, ce qui rend inutile toute fonction d'arbitrage. Le nombre d'esclaves n'est pas limité par la norme mais dépend des contraintes électriques. L'architecture est dite *flexible*, car on peut ajouter des noeuds esclaves sans modifier les noeuds existants.

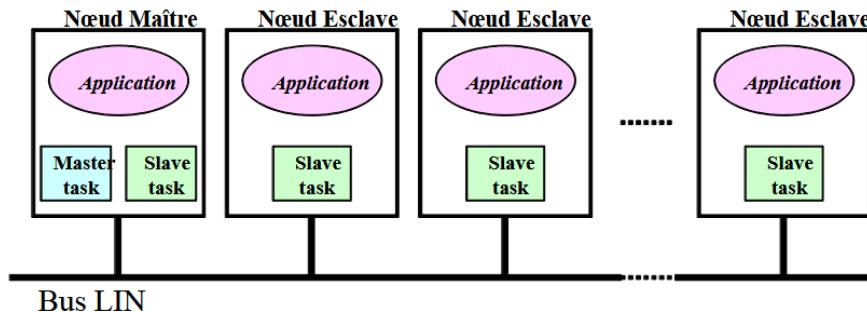


FIGURE 2 – Exemple d'architecture LIN

### Connexion

Le bus est constitué d'une *seule ligne* reliée à chaque noeud par une sortie à collecteur ouvert. Le maître utilise une résistance de tirage de  $1\text{ k}\Omega$ , tandis que chaque esclave utilise  $30\text{ k}\Omega$ . La ligne est au niveau *récessif (1)* lorsqu'aucun noeud ne force l'état, et au niveau *dominant (0)* dès qu'au moins un noeud impose ce niveau.

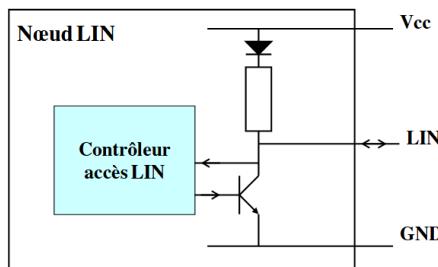


FIGURE 3 – Connexion physique d'un noeud à la ligne LIN

### Vitesse de transmission

Le débit varie de 1 kbit/s à 20 kbit/s, fixé pour une architecture donnée. Trois vitesses sont recommandées :

- **Lente** : 2400 bit/s,
- **Moyenne** : 9600 bit/s,
- **Rapide** : 19200 bit/s.

### Communications et trames

Les messages LIN sont composés de plusieurs champs :

- *Synchronisation Break* : marque le début du message,

- *Synchronisation Field* : alignement des horloges (valeur 0x55),
- *Identification Field* : contenu et longueur des données, avec contrôle de parité,
- *Data Field* : octets d'information transmis du LSB vers le MSB,
- *Checksum Field* : somme de contrôle des données (modulo 256 inversée).

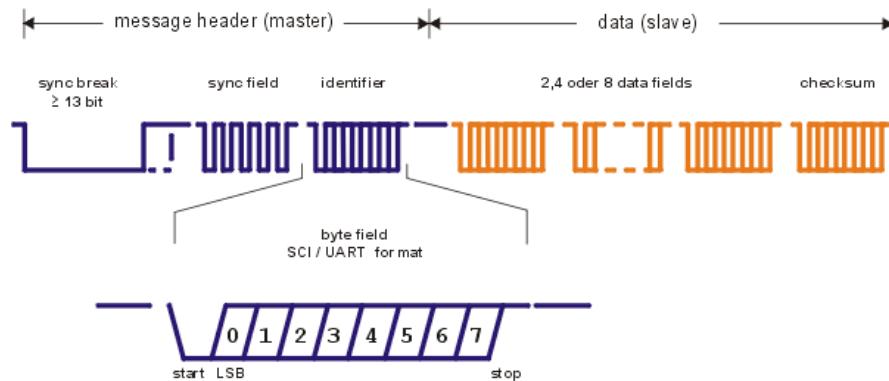


FIGURE 4 – Type de Trame Protocol LIN

Une communication peut être de deux types :

- **Écriture** : le maître envoie l'intégralité du message,
- **Lecture** : le maître envoie seulement l'entête, puis reçoit la réponse de l'esclave.

### 3 Cahier des Charges

Le projet se concentre sur une partie restreinte du récepteur LIN, uniquement pour la réception de trames de type « **écriture** », avec une **entrée LIN unique** et une vitesse fixée à 19 200 bit/s. La distinction maître/esclave et la connexion physique complète ne sont pas traitées.

#### Limitations et simplifications :

- Pas de gestion de perte d'octets,
- Vérification des bits start/stop par un seul échantillon,
- Pas de contrôle de parité ni de vérification du checksum.

#### Fonctionnalités attendues :

- Conversion série → parallèle (données de 8 bits) pour un microprocesseur,
- Possibilité de **filtrer les messages** grâce à un registre de comparaison **SelAddr** (8 bits),
- Signalisation de fin de réception (**M\_Received**) uniquement si l'identifiant reçu correspond à **SelAddr**,
- Réinitialisation des compteurs et effacement des messages non valides.

#### Gestion des messages :

- Un seul message peut être stocké à la fois (FIFO),
- Les octets doivent être accessibles dans leur ordre d'arrivée, même si le message est encore en cours de réception,
- Tous les octets doivent être mémorisés, indépendamment du filtrage,
- Le récepteur doit déterminer la fin du message et l'indiquer au microprocesseur.

#### État du récepteur :

Accessible par registre ( ETAT ) à tout moment, il doit indiquer :

- si un message a été reçu (après filtrage),
- le nombre d'octets reçus,
- les erreurs simples de réception (bits START/STOP, durée du *synchro break*).

Après lecture du registre d'état, les champs sont réinitialisés (sauf le compteur d'octets reçus).

#### Contraintes supplémentaires :

- Interface physique avec le microprocesseur imposée,
- Caractéristiques fonctionnelles, physiques et temporelles définies,
- Temps d'échanges précisés pour assurer la compatibilité avec l'environnement.

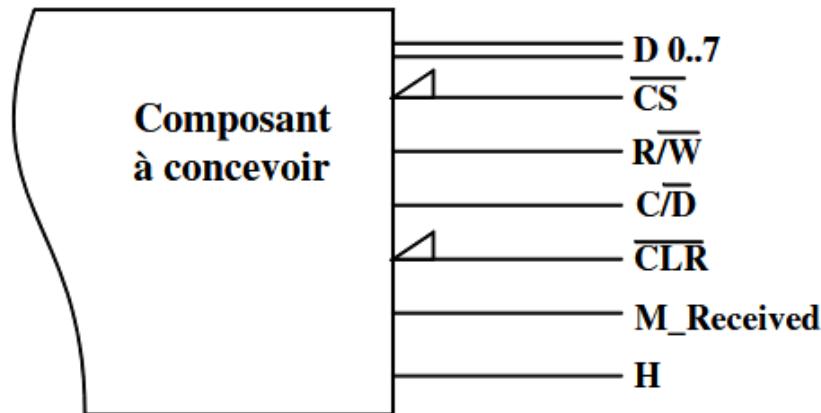


FIGURE 5 – Interface microprocesseur associée au circuit à concevoir

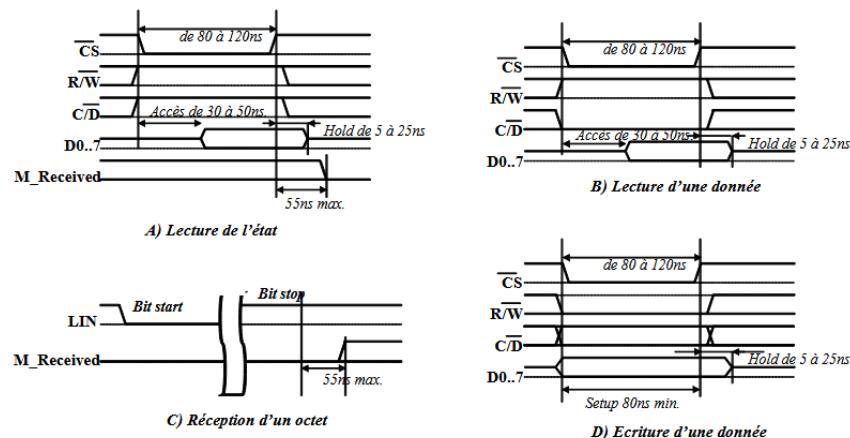


FIGURE 6 – Chronogrammes des échanges entre le circuit et son environnement

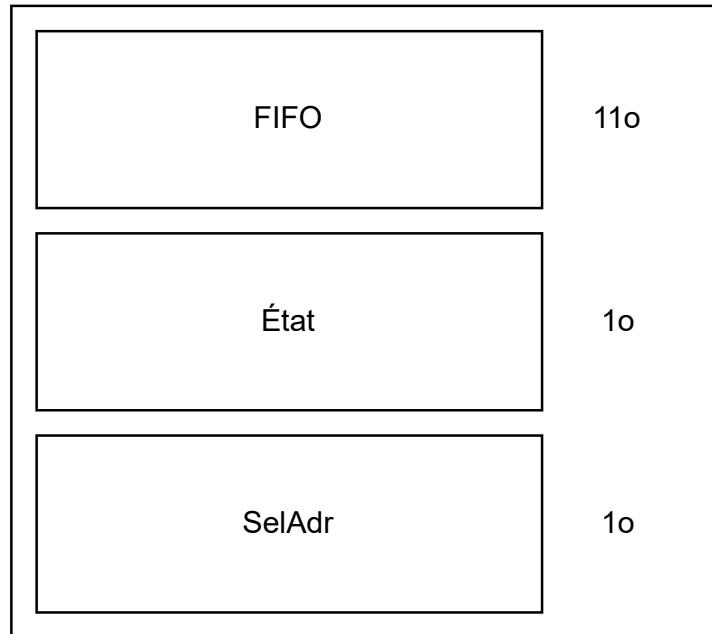


FIGURE 7 – Schema Conception Registre interne Système

## 4 Description des différentes spécifications définies en travaux dirigés

### Objectif

Les spécifications ont pour objectif de définir le comportement attendu du système, autrement dit, ce qui doit être réalisé en réponse au cahier des charges. Elles constituent l'expression formelle des besoins fonctionnels, en se plaçant du point de vue de l'environnement du système, c'est-à-dire, tout ce qui est externe au circuit mais interagit avec lui.

La phase de spécification représente la première étape de la conception d'un circuit. Elle adopte une approche boîte noire : on s'intéresse uniquement à ce que le système doit faire, sans se préoccuper des solutions techniques internes ni du comment il fonctionnera. Les spécifications sont donc indépendantes de la technologie utilisée.

Au départ, nous avions pour indication de créer un bloc capable de communiquer à la fois avec le système de trame LIN et avec le microcontrôleur. Afin de simplifier notre étude et nos spécifications, nous avons choisi de diviser le système en deux parties : la réception de trame, qui se chargera de gérer la réception des données octet par octet, et un autre bloc dédié à la communication avec le microcontrôleur.

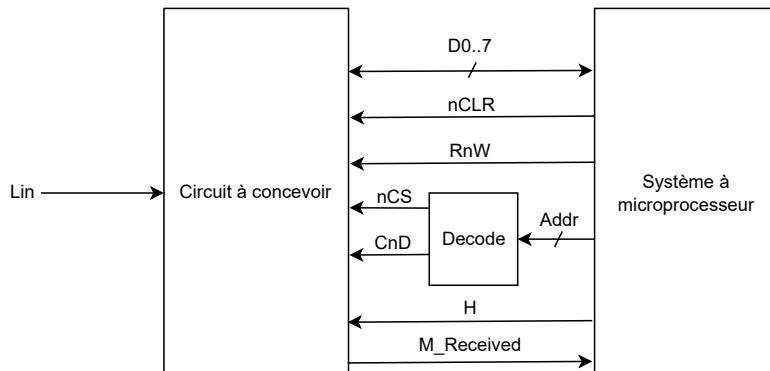


FIGURE 8 – Interface microprocesseur associée au circuit à concevoir

### 4.1 Interface Microprocesseur

D'après les données du cahier des charges, nous pouvons définir tous les signaux nécessaires pour le système réception trame :

Signaux	Mode	Type	Description
D07	INOUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données
nCS	IN	STD_LOGIC	Chip Select
RnW	IN	STD_LOGIC	Opération Lecture / Ecriture
CnD	IN	STD_LOGIC	Opération Contrôle / Données
nCLR	IN	STD_LOGIC	Réinitialisation
M_Received	OUT	STD_LOGIC	Fin de réception de la trame
H	IN	STD_LOGIC	Horloge

## 4.2 Reception Trame

D'après les données du cahier des charges, nous pouvons définir tous les signaux nécessaires pour le système interface Reception Trame :

Signaux	Mode	Type	Description
LIN	IN	STD_LOGIC	Reception de la trame LIN

## 4.3 FIFO

Pour le moment, nous savons qu'il s'agit d'un registre de stockage fonctionnant en mode FIFO, destiné à mémoriser les données de réception d'une trame LIN :

Signaux	Mode	Type	Description
OctetRecu	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Signal Entrée
OctetLu	OUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Signal Sortie

## 4.4 ETAT

Ce registre peut être considéré comme le registre de log de la trame. Grâce au cahier des charges, nous connaissons précisément ses fonctionnalités :

Signaux	Mode	Type	Description
Erreur_Start	IN	STD_LOGIC	Bit d'erreur de Start
Erreur_Stop	IN	STD_LOGIC	Bit d'erreur de Stop
Erreur_SynchroBreak	IN	STD_LOGIC	Bit d'erreur de Synchro Break
NbOctetReceived	IN	STD_LOGIC_VECTOR(3 DOWNTO 0)	Nombre d'octets reçus
MessageReceived_SET	IN	STD_LOGIC	Indicateur de trame reçue
EtatLu	OUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Octet d'information de la trame

## 5 Description et justification de la structure fonctionnelle

### Objectifs

Cette section présente l'organisation du système en blocs fonctionnels et décrit leurs interactions. Chaque bloc (réception de trame, FIFO, registre d'état, interface microprocesseur) est expliqué dans son rôle et sa contribution au fonctionnement global. L'objectif est de montrer comment les fonctionnalités spécifiées sont réparties de manière logique pour répondre au cahier des charges.

Dans cette section, nous établissons une structure fonctionnelle indépendante de la technologie.

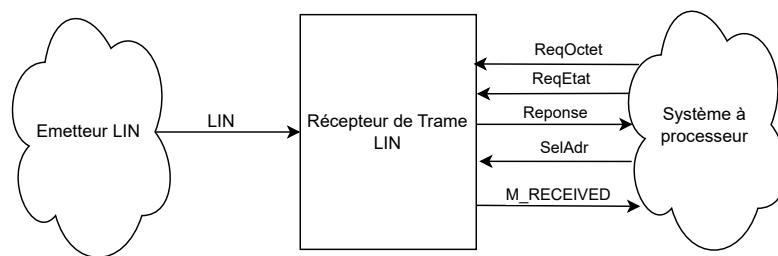


FIGURE 9 – Structure fonctionnelle du système

Pour le moment, nous nous sommes limités à deux blocs principaux : l'interface microprocesseur et la réception des trames. Ces deux blocs sont connectés à un bloc d'échange, qui permet la communication entre eux. Ce bloc assure l'interprétation des échanges avec le microprocesseur ainsi que la gestion des deux registres de données.

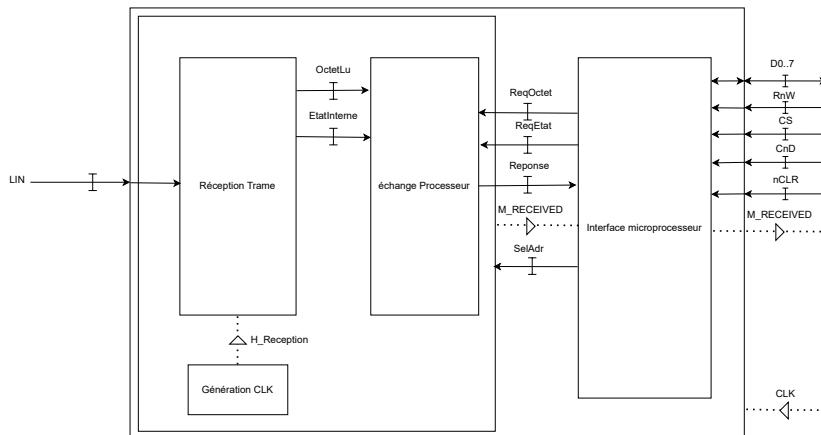


FIGURE 10 – Architecture du système de réception de trame LIN V1

Nous pouvons également représenter l'automate du système de communication avec le processeur :

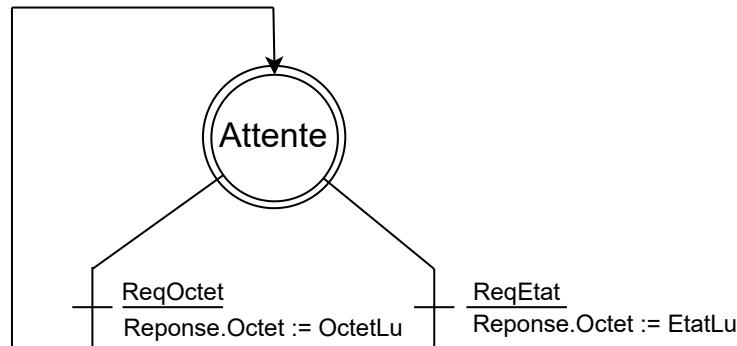


FIGURE 11 – Échanges avec le processeur

Dans notre démarche d'optimisation du système, nous nous sommes rendu compte que le bloc d'échange microprocesseur pouvait être intégré au bloc d'interface. Cela permettra de supprimer des signaux supplémentaires, inutiles et encombrants.

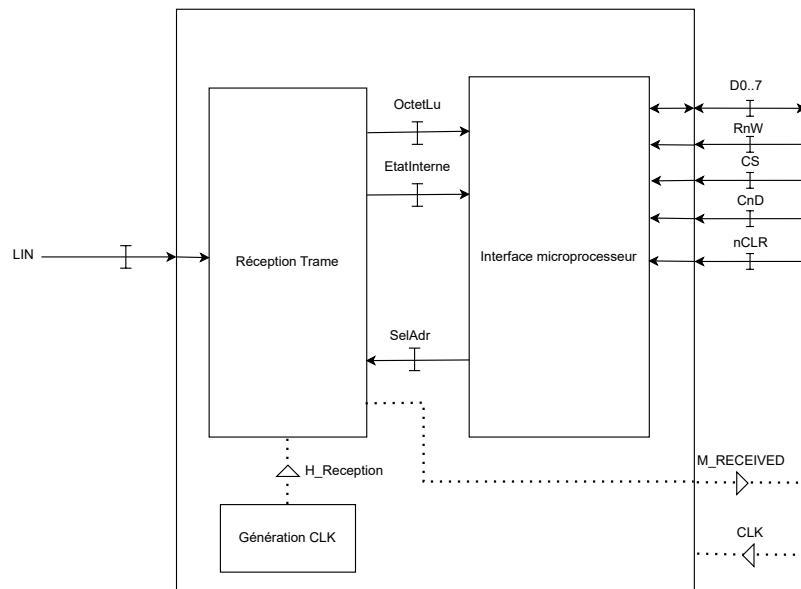


FIGURE 12 – Architecture du système de réception de trame LIN V2

Par la suite, nous avons décidé d'ajouter deux blocs de registres internes : le registre de stockage des données de la trame LIN (FIFO) et le registre d'état interne (ETAT), qui permet de connaître les informations sur l'état de la réception.

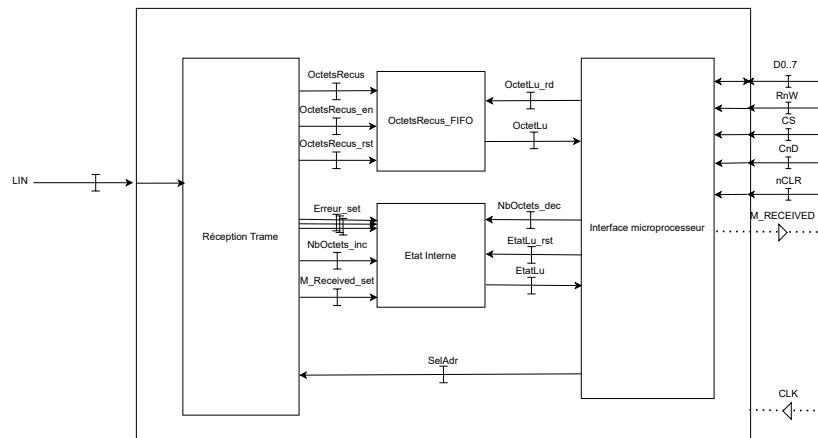


FIGURE 13 – Description finale du circuit

Comme indiqué ci-dessus, dans le schéma de notre système global, nous avons décidé d'ajouter certains signaux internes entre les blocs de registres et les interfaces.

## FIFO

Signaux	Mode	Type	Description
OctetRecu_WR	IN	STD_LOGIC	Read / Write opération
OctetRecu_RST	IN	STD_LOGIC	Réinitialisation des données reçues
OctetLu_RD	IN	STD_LOGIC	Sélection de la mémoire (Control / Data)

La définition de ces nouveaux signaux permet de gérer de manière précise le fonctionnement de la FIFO. OctetRecu\_WR est un signal d'écriture qui déclenche l'enregistrement d'un octet reçu dans la mémoire FIFO, garantissant que chaque donnée entrante est correctement stockée. OctetRecu\_RST est un signal de réinitialisation qui vide complètement la FIFO et remet à zéro tous les compteurs internes, permettant ainsi de reprendre la réception de données sans risque de corruption ou de chevauchement. OctetLu\_RD est un signal de lecture qui active l'accès aux données stockées dans la FIFO et permet leur transfert vers le microprocesseur ou d'autres blocs du système. Ensemble, ces signaux assurent une communication fiable, synchronisée et organisée entre l'interface microprocesseur et le registre de réception, tout en facilitant la gestion des flux de données entrants.

## ETAT

Signaux	Mode	Type	Description
NbOctetRecu_RST	IN	STD_LOGIC	Réinitialisation du compteur d'octets
DecNbOctet	IN	STD_LOGIC	Flag de lecture pour FIFO
EtatLu_RST	IN	STD_LOGIC	Reset de l'état lu

La définition de ces signaux du bloc ETAT permet de contrôler et de suivre l'état interne de la réception des données. NbOctetRecu\_RST réinitialise le compteur d'octets reçus, assurant un suivi précis du nombre de données traitées. DecNbOctet agit comme un indicateur de lecture pour la FIFO, signalant quand un octet peut être lu et transféré, ce qui permet une gestion correcte des flux de données. EtatLu\_RST permet de réinitialiser les informations d'état lues,

garantissant que le système dispose toujours d'une représentation exacte de l'état actuel de la réception. Ensemble, ces signaux assurent une supervision fiable et synchronisée des opérations internes, facilitant le contrôle et la gestion du flux de données dans le système.

## 6 Description et justification de la solution architectureale obtenue pour le circuit

### Objectifs

Une fois les algorithmes fonctionnels définis, la description fonctionnelle interne du circuit est considérée comme complète. Cette description reste indépendante de la technologie utilisée et ne prend pas nécessairement en compte les réalités physiques, telles que les interférences entre signaux et entités.

La phase de conception architecturale consiste alors à intégrer les interfaces physiques, à analyser les opportunités de simplification des algorithmes, et à définir la stratégie d'implantation du circuit.

Dans cette partie, nous nous intéressons à l'architecture matérielle à partir de la solution fonctionnelle et à la description du circuit au niveau RT (Register Transfer).

- Introduction des interfaces physiques
- Identification des ressources logiques de stockage
- Description structurelle du circuit au niveau RT
- Écriture du comportement du circuit au niveau RT

### 6.1 Horloge

Nous commençons par l'une des parties les plus simples du système : la gestion de l'horloge. La vitesse de l'horloge est déterminée à partir du pas d'échantillonnage  $N$ , défini par :

$$N = \frac{T_{bit}}{T_{processeur}}$$

Dans notre cas, le cycle de lecture/écriture spécifié dans le cahier des charges est en moyenne de 100 ns, avec une vitesse de transmission de 19 200 bit/s. Cela donne :

$$N = \frac{52 \text{ } \mu\text{s}}{100 \text{ } ns} = 520$$

Pour notre implémentation, nous choisissons  $N = 2048$ , une valeur arbitraire qui facilite la synchronisation et le traitement interne.

### 6.2 Architecture Reception Trame

Pour rappel voici tous les signaux associés à ce bloc :

Signaux	Mode	Type	Description
LIN	IN	STD_LOGIC	Bus de données d'entrée
SelAddr	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Sélection Addrress Composant
OctetReçu	OUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données de sortie
OctetReçu_WR	OUT	STD_LOGIC	Read / Write opération
OctetReçu_RST	OUT	STD_LOGIC	Réinitialisation des données reçues
Erreur_Start	OUT	STD_LOGIC	Bit d'erreur de Start
Erreur_Stop	OUT	STD_LOGIC	Bit d'erreur de Stop
Erreur_SynchroBreak	OUT	STD_LOGIC	Bit d'erreur de Synchro Break
IncNbOctet	OUT	STD_LOGIC	Flag de reception pour lecture
MessageReceived_SET	OUT	STD_LOGIC	Indicateur de trame reçue
NbOctetReçu_RST	OUT	STD_LOGIC	Réinitialisation du compteur d'octets

Pour le bloc de réception de trames, nous implémentons une machine séquentielle qui permet de distinguer une partie opérative et une unité de commande, assurant ainsi la gestion efficace de la réception de ce type de trame.

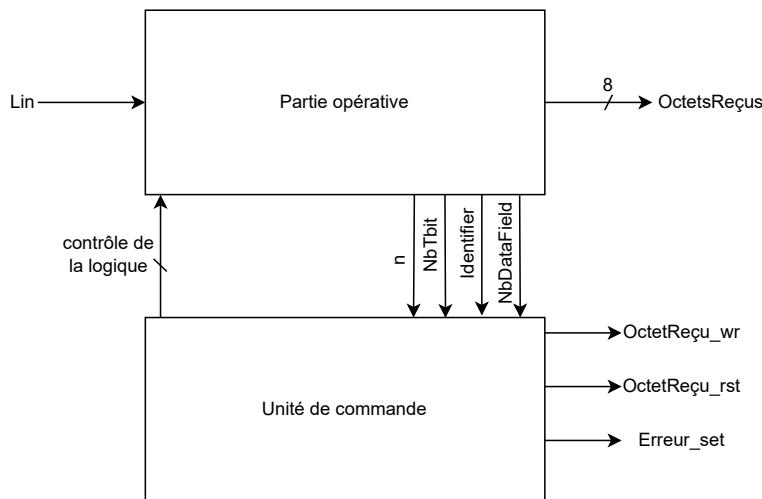


FIGURE 14 – Machine Séquentielle Reception Trame

Nous ajoutons des variable interne pour la communication entre les deux bloc de la machine afin d'obtenir un sytème correcte :

Variable	Taille (bit)	Opération	Opérateur	Signaux de contrôle
n	$\log_2(N)$	décrémentation, initialisation à $N - 1$ ou $N/2$	décompteur, Mux	n_Load, n_En, n_Select
NbTbit	4	décrémentation, initialisation à 13 ou 8	décompteur, Mux	NBTbit_Load, NBTbit_en, NBTbit_select
Identifier	8	sauvegarde	registre 8 bits	Identifier_en
OctetsReçus	8	décalage bit à bit	registre à décalage	OctetReçu_en
NbDataField	3	décrémentation, initialisation à 1, 3 ou 7	décompteur, décodeur	NBdatafield_en, NBdatafield_load

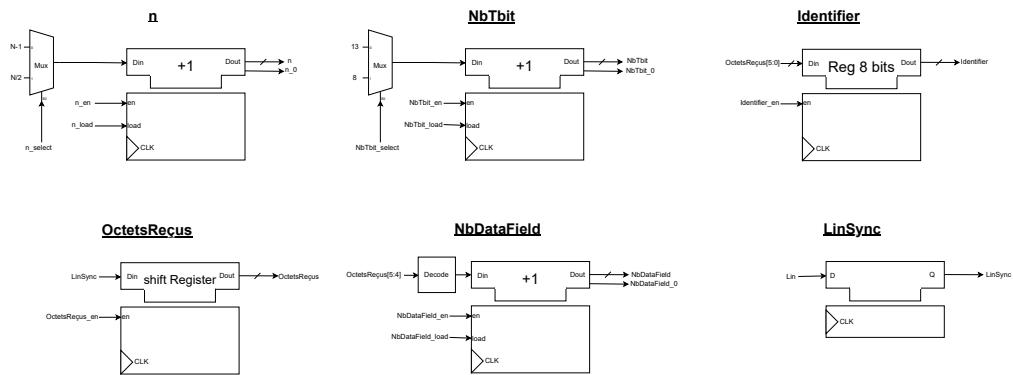


FIGURE 15 – Structure partie Opérative Reception Trame

Dans cette partie nous retrouvons les différents compteurs et registres nécessaires au fonctionnement de la réception de trame. Pour prendre un exemple le premier en haut à gauche représente le décompteur *n* qui permet de décompter jusqu'à *N-1* ou *N/2* pour l'échantillonnage. Le choix de *N-1* ou *N/2* dépend de l'utilisation, soit *N/2* pour la détection de l'état au bout de un demi Tbit ou *N-1* pour la détection de l'état à la fin de un Tbit ( Potentiel Front ).

Pour la partie commande, nous avons conçu et implémenté un automate afin de gérer les différentes séquences de contrôle :

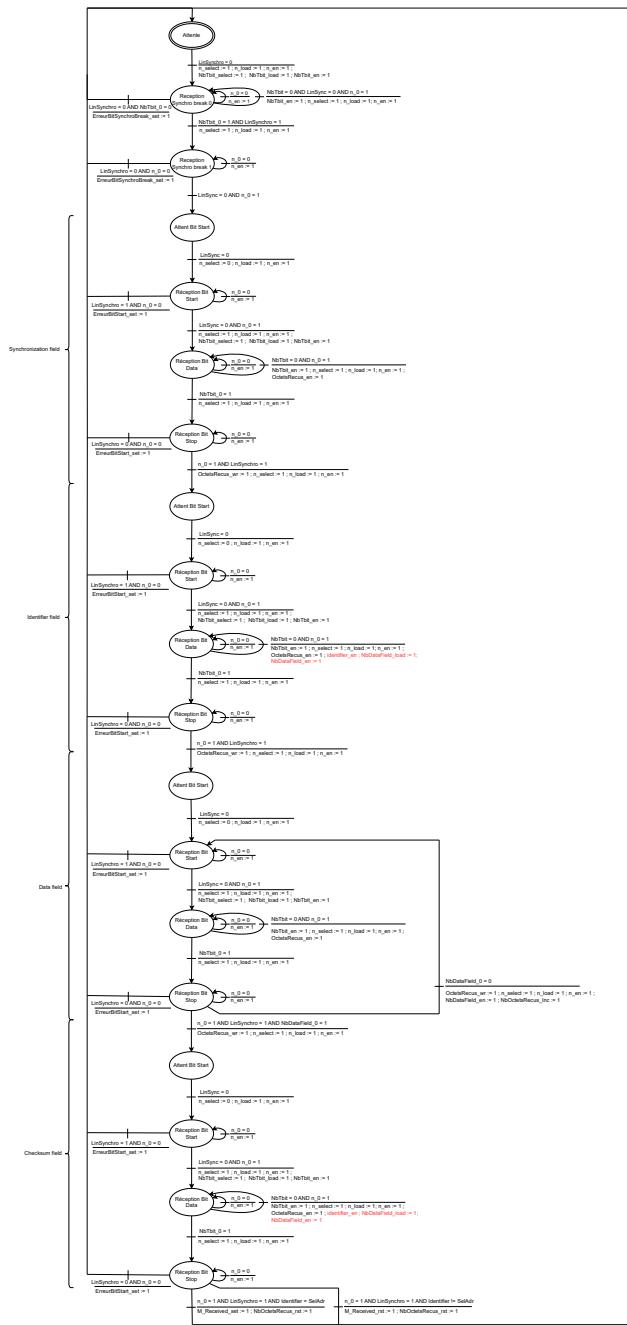


FIGURE 16 – Structure partie Commande Reception Trame

## Description de l'automate

Cet automate décrit un processus de réception de trame LIN (Local Interconnect Network), organisé en états et transitions conditionnées par des signaux de synchronisation, des compteurs et des champs de données. Il commence par une phase d'**attente**, suivie d'une **synchronisation sur break** où la ligne (LinSynchro) est surveillée pour détecter le début de la

trame. Des erreurs de synchro ou de bit de start peuvent être signalées via des flags comme `ErreurBitSynchroBreak_set` ou `ErreurBitStart_set`.

Ensuite, l'automate passe à la réception des différents champs de la trame :

- **Synchro** (octet de synchronisation),
- **Identifier** (champ d'identifiant),
- **Data field** (données, avec gestion du compteur `NbDataField`),
- **Checksum** (vérification de l'intégrité).

À chaque octet, le système contrôle le bit de start, les données, et le bit de stop, en décrémentant les compteurs comme `NbTbit` ou `NbOctetsRecus`. Selon que l'identifiant reçu correspond ou non à l'adresse sélectionnée (`SelAddr`), l'automate valide la trame (`M_Received_set`) ou la rejette. Des réinitialisations (`rst`) ont lieu après réception complète ou en cas d'erreur.

Ce processus assure une réception séquentielle et robuste, typique des protocoles série, avec gestion d'erreurs et validation conditionnelle des trames.

Cet automate peut être représenté sous forme de machine de Mealy, ce qui simplifie l'écriture du code VHDL :

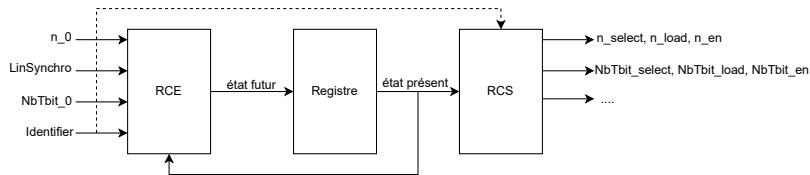


FIGURE 17 – Machine de MEALY Unité de Commande Reception Trame

### 6.3 Architecture Interface MicroProcesseur

Pour rappel voici tous les signaux associés à ce bloc :

Signaux	Mode	Type	Description
D07	INOUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données
nCS	IN	STD_LOGIC	Chip Select
RnW	IN	STD_LOGIC	Opération Lecture / Écriture
CnD	IN	STD_LOGIC	Opération Contrôle / Données
nCLR	IN	STD_LOGIC	Réinitialisation
M_Received	OUT	STD_LOGIC	Fin de réception de la trame
H	IN	STD_LOGIC	Horloge
EtatLu	IN	STD_LOGIC_VECTOR(3 DOWNTO 0)	Octet d'information de la Trame
DecNbOctet	OUT	STD_LOGIC	Flag de lecture pour FIFO
EtatLu_RST	OUT	STD_LOGIC	Reset de l'état lu
OctetLu	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données de sortie
OctetLu_RD	OUT	STD_LOGIC	Sélection de la mémoire (Control / Data)

Pour le bloc d'interface Microprocesseur, nous implémentons une machine séquentielle qui permet de distinguer une partie opérative et une partie commande, assurant ainsi la gestion efficace du contrôle du système et la lecture des différentes informations.

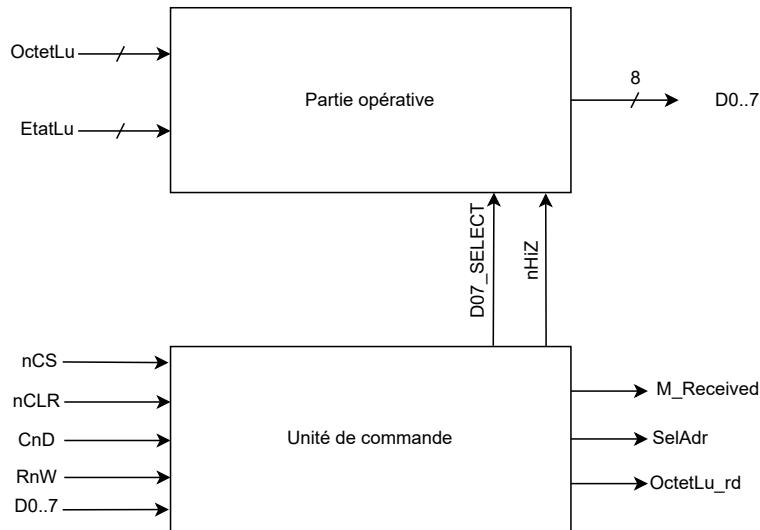


FIGURE 18 – Machine Séquentielle Reception Trame

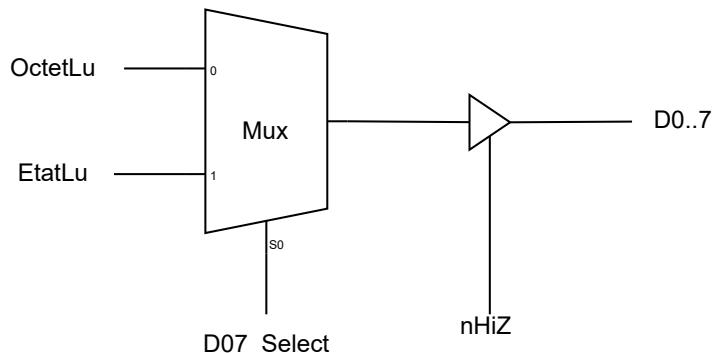


FIGURE 19 – Structure partie Opérative Interface Microprocesseur

Pour la partie opérative de l'interface microprocesseur, nous obtenons un schéma simple de multiplexeur permettant de choisir quelles données mettre dans le vecteur D07 en sortie, soit celles provenant de la FIFO, soit celles de l'État interne.

Pour la partie commande, nous avons conçu et implémenté un automate afin de gérer les différentes séquences de contrôle :

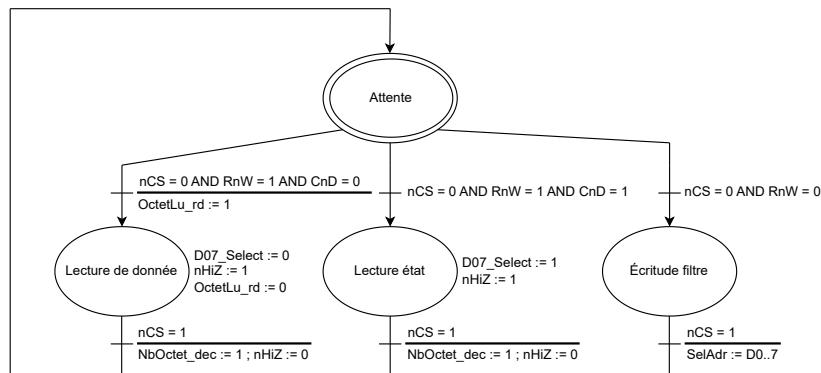


FIGURE 20 – Structure partie Commande Interface Microprocesseur

Cet automate peut être représenté sous forme de machine de Mealy, ce qui simplifie l'écriture du code VHDL :

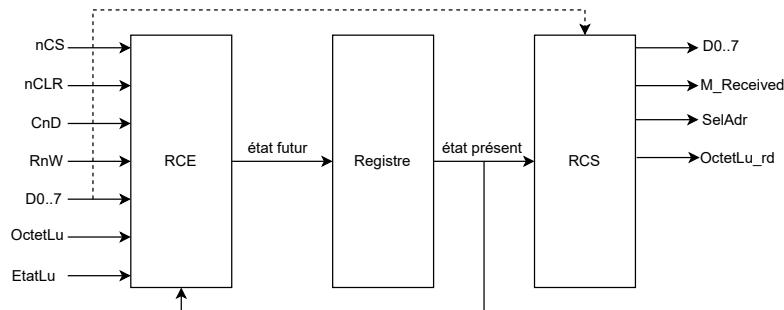


FIGURE 21 – Machine de MEALY Unité de Commande Interface Micro

#### 6.4 Architecture FIFO

Pour rappel voici tous les signaux associés à ce bloc :

Signaux	Mode	Type	Description
OctetRecu	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données d'entrée
OctetRecu_WR	IN	STD_LOGIC	Read / Write opération
OctetRecu_RST	IN	STD_LOGIC	Réinitialisation des données reçues
OctetLu	OUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données de sortie
OctetLu_RD	IN	STD_LOGIC	Sélection de la mémoire (Control / Data)

Étant donné que ce système est moins complexe que les deux interfaces précédentes, nous avons choisi de ne pas le réaliser sous forme de machine séquentielle et de le représenter directement comme un bloc, comme montré ci-dessous :

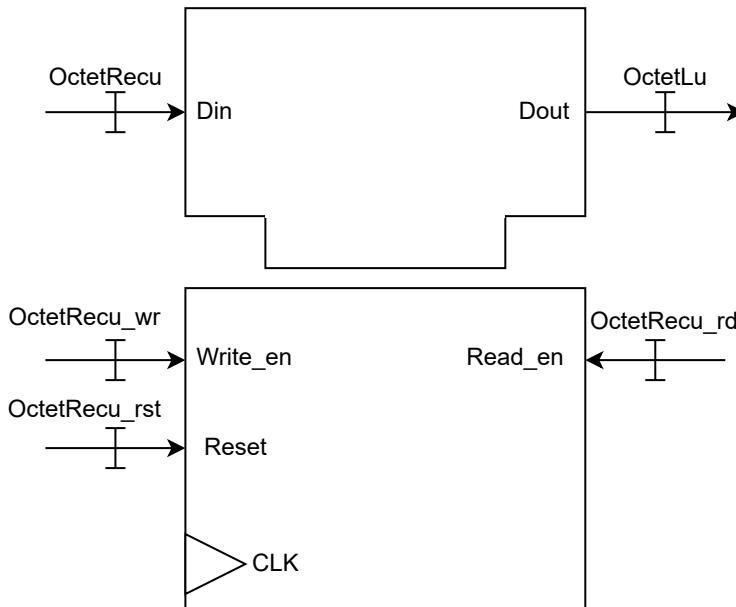


FIGURE 22 – Implémentation de la FIFO

## 6.5 Implémentation de l'État Interne

Pour rappel voici tous les signaux associés à ce bloc :

Signaux	Mode	Type	Description
Erreur_Start	IN	STD_LOGIC	Bit d'erreur de Start
Erreur_Stop	IN	STD_LOGIC	Bit d'erreur de Stop
Erreur_SynchroBreak	IN	STD_LOGIC	Bit d'erreur de Synchro Break
IncNbOctet	IN	STD_LOGIC	Flag de reception pour lecture
MessageReceived_SET	IN	STD_LOGIC	Indicateur de trame reçue
NbOctetRecu_RST	IN	STD_LOGIC	Réinitialisation du compteur d'octets
EtatLu	OUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Octet d'information de la Trame
DecNbOctet	IN	STD_LOGIC	Flag de lecture pour FIFO
EtatLu_RST	IN	STD_LOGIC	Reset de l'état lu

Étant donné que ce système présente une complexité similaire et une facilité d'implémentation comparable, nous le conservons également sous forme de bloc, comme montré ci-dessous :

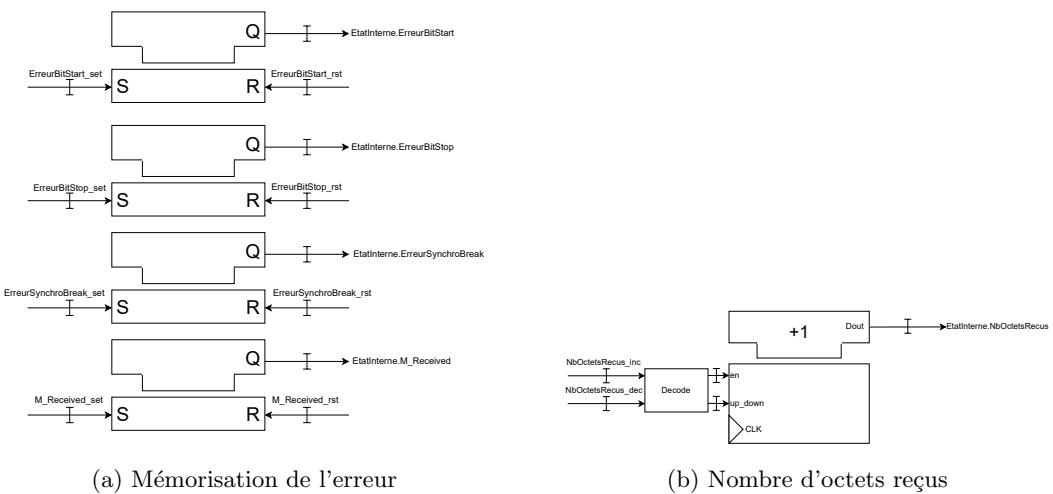


FIGURE 23 – implémentation de l'état interne en représentation structurelle au niveau RT

## 7 Présentation du fonctionnement des fonctions

### 7.1 Interface MicroProcesseur

Dans cette partie, nous avons initié une séance de travaux pratiques pour nous familiariser avec le logiciel HDL Designer. Le programme «Interface Microprocesseur», préalablement implémenté par les enseignants, respecte strictement les données présentées dans le TD et développées dans les sections précédentes du rapport. Nous allons l'étudier en détail afin de démontrer sa correspondance avec le modèle théorique.

Pour rappel, l'interface Microprocesseur a été conçue selon une machine séquentielle, tandis que la partie commande a été développée sur le modèle d'une machine de Mealy. Le code présenté respecte rigoureusement la structure des blocs : réseau combinatoire d'entrée, réseau combinatoire de sortie et registres correspondant à la machine à états.

#### 7.1.1 Réseau Combinatoire d'Entrée

```

1 InputProc_Synchro :  PROCESS(H, nRST)
2 BEGIN
3   IF (nRST='0') THEN
4     nCS_Synchro <= '1';
5     RnW_Synchro <= '1';
6     CnD_Synchro <= '1';
7     D07_Synchro <= (others => '0');
8   ELSIF (H'EVENT AND H='1') THEN
9     nCS_Synchro <= nCS;
10    RnW_Synchro <= RnW;
11    CnD_Synchro <= CnD;
12    D07_Synchro <= D07;
13  END IF;
14 END PROCESS InputProc_Synchro;

```

Listing 1 – Reseau Cominatoire d'entrée

Ce bloc VHDL gère la synchronisation des signaux provenant du microprocesseur. Le processus InputProc\_Synchro lit les signaux d'entrée à chaque front montant de l'horloge H et les initialise lors de la mise à zéro nRST. Les signaux synchronisés (nCS\_Synchro, RnW\_Synchro, CnD\_Synchro, D07\_Synchro) sont ensuite utilisés par le reste de l'interface.

#### 7.1.2 Réseau Combinatoire de Sortie

```

1 OutputProc_Comb : PROCESS(nCS_Synchro, CnD_Synchro, RnW_Synchro,
2   EtatCourant, OctetLu, EtatLu)
3 BEGIN
4   D07 <= (others => 'Z');
5   OctetLu_RD <= '0';
6   EtatLu_RST <= '0';
7   DecNbOctet <= '0';
8   CASE EtatCourant IS
9     WHEN Attente =>
10       IF (nCS_Synchro='0' AND CnD_Synchro='0' AND RnW_Synchro='1') THEN
11         OctetLu_RD <= '1';
12       END IF;
13     WHEN LectureData =>

```

```

13      D07 <= OctetLu;
14      IF (nCS_Synchro='1') THEN
15          DecNbOctet <= '1';
16      END IF;
17      WHEN LectureEtat =>
18          D07 <= EtatLu;
19          IF (nCS_Synchro='1') THEN
20              EtatLu_RST <= '1';
21          END IF;
22      WHEN EcritureFiltre =>
23      END CASE;
24  END PROCESS OutputProc_Comb;

```

Listing 2 – Réseau Combinatoire de Sortie

Le processus `OutputProc_Comb` contrôle la sortie des données et des états vers le microprocesseur. Il met à jour les signaux `D07`, `OctetLu_RD`, `EstatLu_RST`, `DecNbOctet` en fonction de l'état courant de la machine et des signaux synchronisés d'entrée. La logique combinatoire assure la correspondance entre les actions de lecture/écriture et l'état de la machine.

### 7.1.3 Registres et Machine à États

```

1 ClockedProc : PROCESS(H, nRST)
2 BEGIN
3     IF (nRST='0') THEN
4         EtatCourant <= Attente;
5     ELSIF (H'EVENT AND H='1') THEN
6         EtatCourant <= EtatSuivant;
7     END IF;
8 END PROCESS ClockedProc;
9
10 NextStateProc : PROCESS(nCS_Synchro, CnD_Synchro, RnW_Synchro,
11                         EtatCourant)
12 BEGIN
13     EtatSuivant <= EtatCourant;
14     CASE EtatCourant IS
15     WHEN Attente =>
16         IF (nCS_Synchro='0' AND CnD_Synchro='0' AND RnW_Synchro='1') THEN
17             EtatSuivant <= LectureData;
18         ELSIF (nCS_Synchro='0' AND CnD_Synchro='1' AND RnW_Synchro='1') THEN
19             EtatSuivant <= LectureEtat;
20         ELSIF (nCS_Synchro='0' AND CnD_Synchro='0' AND RnW_Synchro='0') THEN
21             EtatSuivant <= EcritureFiltre;
22         ELSE
23             EtatSuivant <= Attente;
24         END IF;
25     WHEN LectureData =>
26         IF (nCS_Synchro='1') THEN
27             EtatSuivant <= Attente;
28         ELSE
29             EtatSuivant <= LectureData;
30         END IF;
31     WHEN LectureEtat =>
32         IF (nCS_Synchro='1') THEN
33             EtatSuivant <= Attente;
34         ELSE
35             EtatSuivant <= LectureEtat;

```

```

35     END IF;
36 WHEN EcritureFiltre =>
37     IF (nCS_Synchro='1') THEN
38         EtatSuivant <= Attente;
39     ELSE
40         EtatSuivant <= EcritureFiltre;
41     END IF;
42 END CASE;
43 END PROCESS NextStateProc;

```

Listing 3 – Registres

Les processus `ClockedProc` et `NextStateProc` implémentent la machine séquentielle. `ClockedProc` met à jour l'état courant à chaque front montant de l'horloge et réinitialise l'état au démarrage. `NextStateProc` définit l'état suivant selon les conditions des signaux d'entrée et l'état courant, en suivant la logique de la machine de Mealy.

#### 7.1.4 Réseau Synchronisé de Sortie

```

1 OutputProc_Synchro : PROCESS(H, nCLR)
2 BEGIN
3     IF (nCLR='0') THEN
4         SelAdr <= (others => '0');
5     ELSIF (H'EVENT AND H='1') THEN
6         CASE EtatCourant IS
7             WHEN EcritureFiltre =>
8                 IF (nCS_Synchro='1') THEN
9                     SelAdr <= D07_Synchro;
10                END IF;
11            WHEN OTHERS =>
12                END CASE;
13            END IF;
14        END PROCESS OutputProc_Synchro;
15
16 M_Received <= EtatLu(4);

```

Listing 4 – Réseau Synchronisé de Sortie

Le processus `OutputProc_Synchro` synchronise la sélection d'adresse `SelAdr` avec l'horloge `H`. Il est actif principalement pendant l'état `EcritureFiltre`, assurant que les données de l'entrée `D07_Synchro` sont correctement mémorisées. Le signal `M_Received` est également mis à jour pour refléter l'état du bit correspondant.

## 8 Simulation des fonctions

### 8.1 Interface Microprocesseur

Dans cette section, nous présentons la simulation du bloc *Interface Microprocesseur* et l'analyse des chronogrammes obtenus. La simulation a été réalisée à l'aide d'un *testbench*, implémenté sous la forme d'un bloc nommé **EnvTest\_InterfaceMicroprocesseur**, connecté au composant **InterfaceMicroprocesseur**. L'objectif est de vérifier la conformité du fonctionnement par rapport à l'automate décrit dans la section *Architecture*.

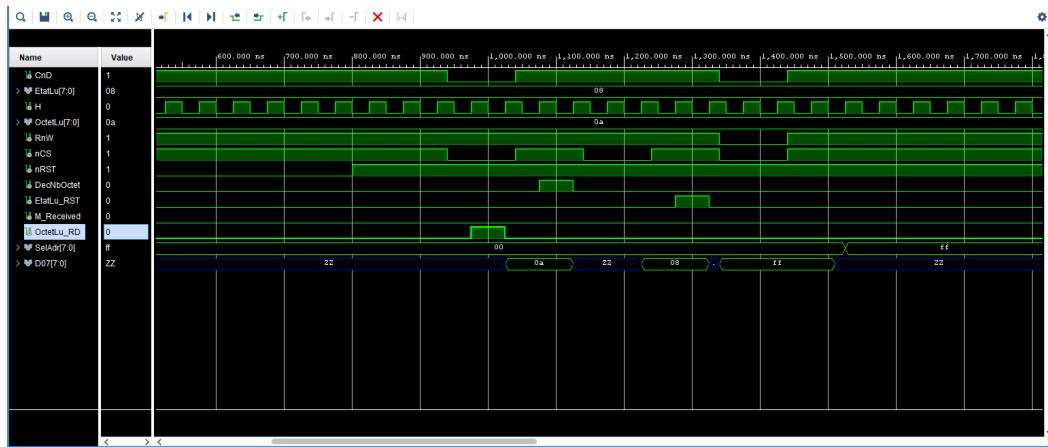


FIGURE 24 – Chronogramme de simulation de l'Interface Microprocesseur

Le testbench (fourni en annexe) a pour rôle de reproduire l'environnement dans lequel le composant est amené à fonctionner. Il émule le comportement d'un microprocesseur en générant automatiquement les stimuli nécessaires à la validation du bloc testé.

#### 8.1.1 Déclarations et signaux

Le testbench commence par la déclaration des librairies IEEE, nécessaires à la manipulation des types logiques et des vecteurs binaires. Les principaux signaux utilisés sont :

- CnD, RnW, nCS, nRST, H : lignes de contrôle classiques d'une interface microprocesseur (commande/données, lecture/écriture, sélection du composant, reset, horloge),
- OctetLu, EtatLu, SelAdr, D07 : bus de données et d'adresses sur 8 bits,
- DecNbOctet, EtatLu\_RST, M\_Received, OctetLu\_RD : signaux internes utilisés pour la communication avec le composant testé.

#### 8.1.2 Instanciation du composant testé

Le composant **InterfaceMicroprocesseur** est instancié dans l'architecture de simulation. Il est relié à l'ensemble des signaux déclarés, permettant ainsi l'observation de son comportement face aux stimuli générés.

#### 8.1.3 Environnement de test

Le composant **EnvTest\_InterfaceMicroprocesseur** simule le rôle du microprocesseur en générant automatiquement les signaux nécessaires :

- génération de l'horloge (H),
- gestion du reset global (nRST),

- activation des commandes de lecture/écriture (RnW, CnD, nCS),
  - pilotage du bus de données (D07).
- Cet environnement est donné par plusieurs paramètres génériques :
- **CLOCK\_PERIOD** : période d'horloge (50 ns),
  - **RESET\_OFFSET** et **RESET\_DURATION** : moment et durée du reset (500 ns et 300 ns),
  - **ACCESS\_TIME** et **HOLD\_TIME** : contraintes temporelles d'accès et de maintien (40 ns et 70 ns).

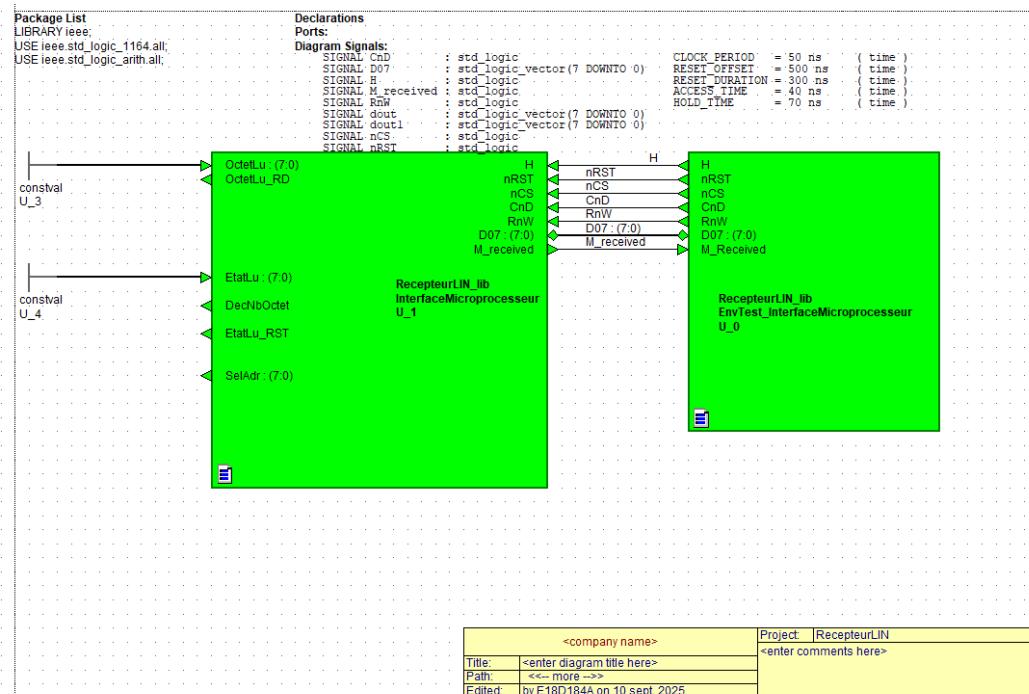


FIGURE 25 – Block Diagramme de test de l'Interface Microprocesseur

#### 8.1.4 Stimuli supplémentaires

Un processus spécifique (StimProc) complète la génération des signaux. Après la fin du reset, il impose des valeurs constantes sur certaines lignes :

- **OctetLu**  $\leftarrow$  10 (codé sur 8 bits),
- **EstatLu**  $\leftarrow$  8 (codé sur 8 bits).

Ces valeurs permettent de vérifier la gestion correcte des données reçues par l'interface. La simulation est ensuite maintenue en attente infinie.

#### 8.1.5 Analyse du chronogramme de simulation

L'analyse du chronogramme met en évidence le comportement attendu du composant :

- lorsque les signaux de contrôle actifs à l'état bas (nRST, nCS) sont à l'état haut, aucune action n'est effectuée,
- lorsque ces signaux sont activés (passage à l'état bas), le composant réagit conformément à l'automate interne,
- les signaux RnW et CnD permettent de sélectionner respectivement les opérations de lecture/écriture et le type d'accès (commande ou données),
- les valeurs imposées sur OctetLu et EtatLu sont correctement lues via le bus de données D07.

Ce chronogramme confirme ainsi le bon fonctionnement du composant **InterfaceMicroprocesseur** : après la levée du reset, l'environnement de test génère des cycles de lecture et d'écriture auxquels le composant répond correctement, en échangeant les données prévues et en activant les signaux de contrôle appropriés.

## 9 Synthèse des fonctions

Une fois la validation en simulation des différents blocs effectuée, il est nécessaire de réaliser la synthèse sur FPGA afin d'observer les ressources logiques attribuées à notre système. Dans le cadre de ce projet, nous avons utilisé un FPGA *AMD Xilinx Artix-7*, plus précisément le modèle **7A35TCPG236**. L'objectif de cette étape est d'analyser les ressources logiques mobilisées, ainsi que les éléments matériels effectivement utilisés par notre conception.

### 9.1 Interface Microprocesseur

Le schéma RTL (*Register Transfer Level*) représente une implémentation synthétisée d'un module matériel décrit en VHDL ou Verilog. Il illustre les registres, les multiplexeurs, les portes logiques, ainsi que la logique séquentielle et combinatoire du circuit.

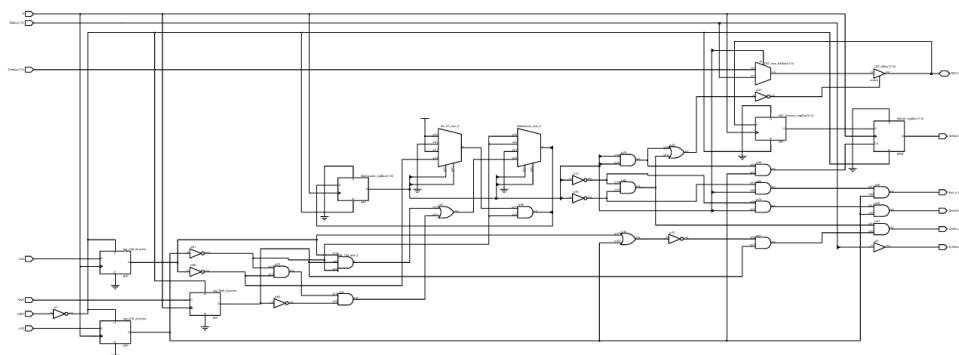


FIGURE 26 – Schéma RTL InterfaceMicroprocesseur

### Structure générale

Le schéma peut être décomposé en plusieurs parties :

- **Entrées principales** : signaux tels que H, CnD, RnW, nRST, nCS, etc.
- **Registres (Flip-Flops D)** : éléments synchronisés par l'horloge, servant à mémoriser l'état interne du circuit.
- **Multiplexeurs (MUX)** : permettent de sélectionner une donnée parmi plusieurs, selon les conditions de contrôle.
- **Logique combinatoire** : réalisée par des portes AND, OR, NOT et XOR, afin de générer les conditions de transition et les sorties.
- **Sorties** : plusieurs signaux dérivés de l'état interne, comme **State\_XX**, **Output\_XX**, etc.

### Fonctionnement global

Le circuit implémente une **machine à états finis** :

- Les **registres** contiennent l'état courant.
- La **logique combinatoire** calcule l'état suivant en fonction de l'état courant et des entrées.
- Les **multiplexeurs** dirigent les transitions entre états.
- Les **sorties** sont activées ou désactivées selon l'état courant et certaines combinaisons d'entrées.

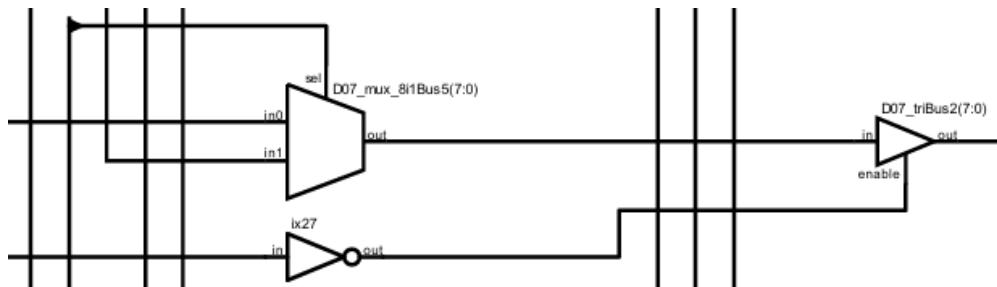


FIGURE 27 – Partie opérative avec multiplexeur et Tristate : InterfaceMicroprocesseur

De plus, nous pouvons retrouver la partie opérative dessinée en classe, lors de nos TD, qui permet, grâce à un multiplexeur, de sélectionner **Etallu** ou **OctetLu** pour l'envoyer vers une porte **Tristate**. Cela démontre la cohérence entre la réalisation théorique et la mise en œuvre pratique.

À la suite de la synthèse logique nous pouvons avoir la synthèse matériel qui transforme la logique en ressource matérielle.

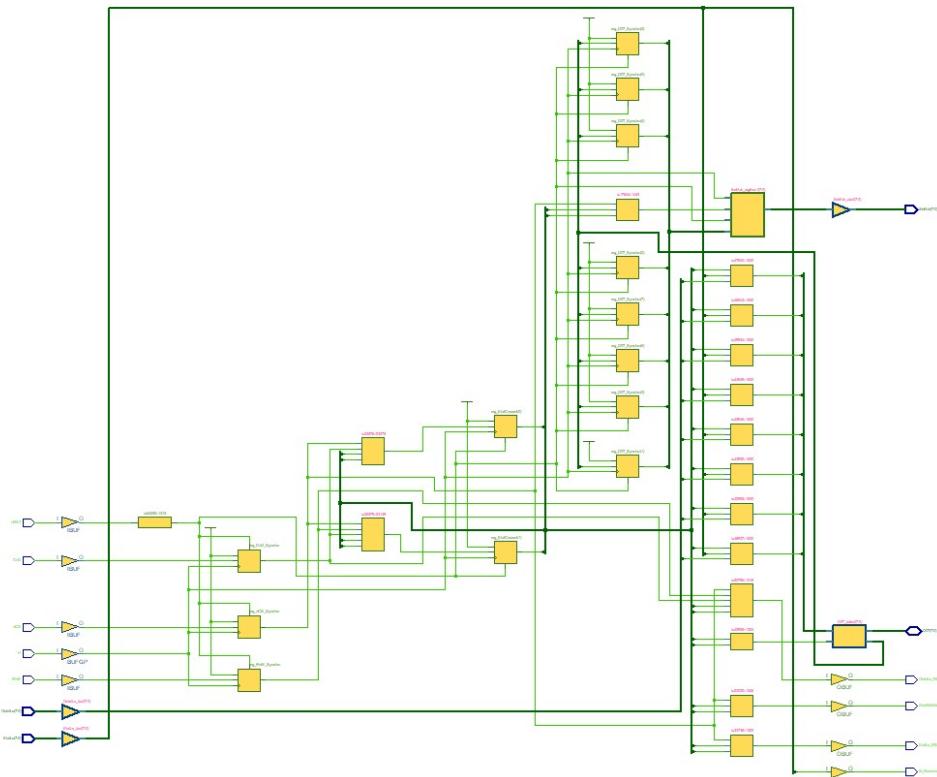


FIGURE 28 – Synthèse matérielle InterfaceMicroprocesseur

Cette transformation consiste à mapper les éléments logiques du schéma RTL, tels que les portes **AND**, **OR**, **NOT** ou les **multiplexeurs**, sur les **ressources matérielles physiques** disponibles dans le FPGA, notamment :

- **LUT (Look-Up Tables)** : les fonctions combinatoires, comme les portes logiques ou les multiplexeurs, sont réalisées à l'aide de LUT. Chaque LUT peut implémenter n'importe

---

quelle fonction booléenne sur un nombre limité d'entrées, ce qui permet de reproduire fidèlement la logique définie dans le HDL.

- **Flip-flops** : les éléments séquentiels tels que les registres ou les bascules sont mappés sur des flip-flops pour stocker les bits et synchroniser les signaux dans le temps.
- **Buffers** : certains chemins logiques nécessitent des buffers pour renforcer les signaux ou adapter les niveaux électriques, garantissant ainsi la stabilité et l'intégrité du routage sur la puce.

Grâce à cette conversion, le design passe d'une **représentation abstraite de la logique** à une **implémentation matérielle concrète**, optimisée pour le FPGA cible. Cela permet non seulement de visualiser l'organisation physique des composants.

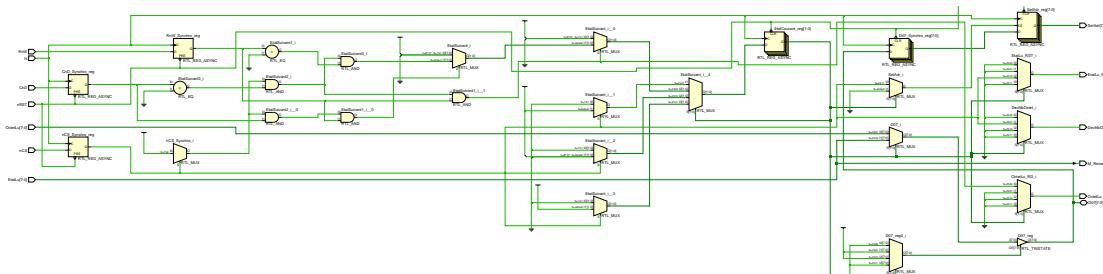
## 10 Routages des Fonctions

### 10.1 Interface Microprocesseur

Après l'étape de **synthèse**, nous pouvons nous intéresser à l'assignation des ressources matérielles du système.

Pour cela, nous utilisons le logiciel **Vivado** (étant donné que les outils de HDL n'étaient pas disponibles le jour du TP), qui permet de générer un schéma RTL ainsi qu'une vue du routage associé à l'interface microprocesseur.

Dans un premier temps, Nous allons resynthétiser le design afin d'obtenir le schéma RTL. Ce schéma, présenté ci-dessous, illustre les ressources logiques utilisées pour implémenter l'interface microprocesseur.



Par la suite, nous pouvons également observer la synthèse matérielle réalisée sur Vivado, qui transforme les ressources logiques en ressources matérielles pour le FPGA.

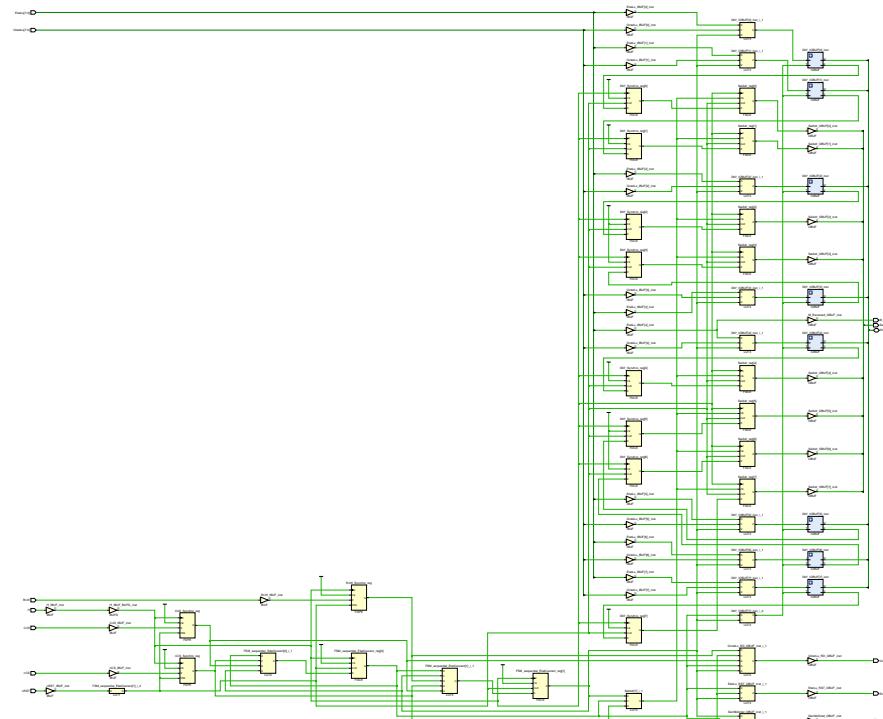
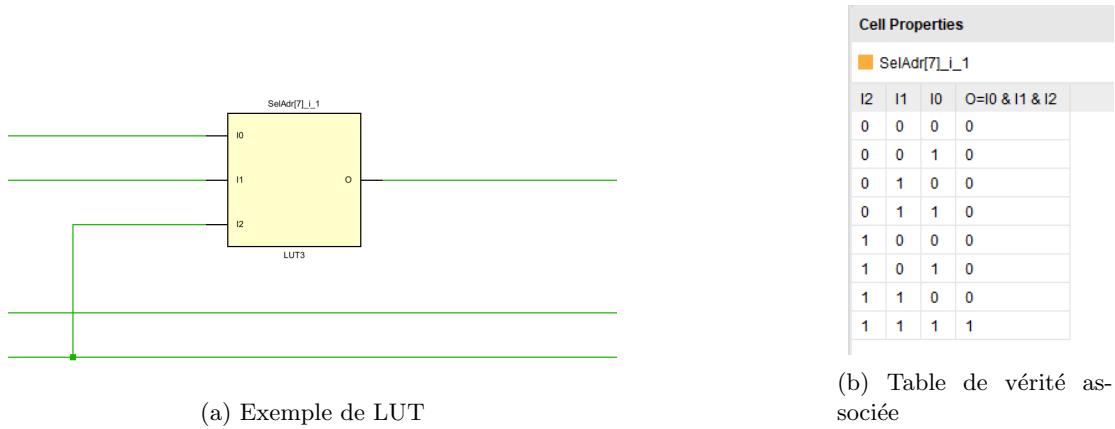


FIGURE 29 – Synthèse matérielle Vivado

Cette synthèse nous montre que la logique est transformée en ressource matérielle, notamment des LUT (Look-Up Tables) et des Flip-Flops, qui sont les éléments de base pour implémenter la logique dans un FPGA.

Les **LUT** (Look-Up Tables), éléments fondamentaux d'un FPGA, peuvent être considérées comme des portes logiques programmables capables de réaliser toute fonction combinatoire. Elles constituent la base de l'implémentation matérielle et offrent une vision schématique complète du système.

Prenons l'exemple d'une **LUT3** :



(a) Exemple de LUT

(b) Table de vérité associée

FIGURE 30 – Illustration d'une LUT et de sa table de vérité

Grace à la table de vérité de la LUT nous remarquons que cette LUT3 implémente une fonction logique ET à trois entrées.

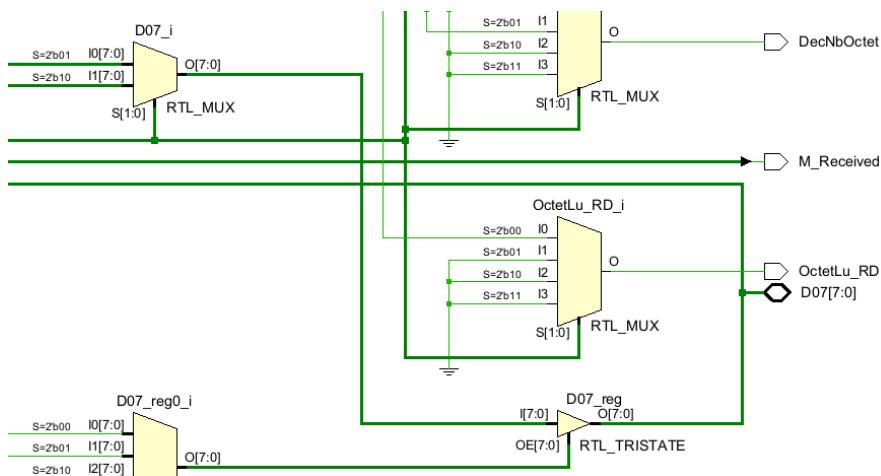


FIGURE 31 – Synthèse Logique Vivado

Nous observons également la présence de la partie opérative de l'interface Microprocesseur.

La suite du flot de conception consiste à lancer l'**implémentation** du système afin d'obtenir

le routage complet. Vivado propose alors une vue générale du FPGA, mettant en évidence ses différentes zones fonctionnelles :

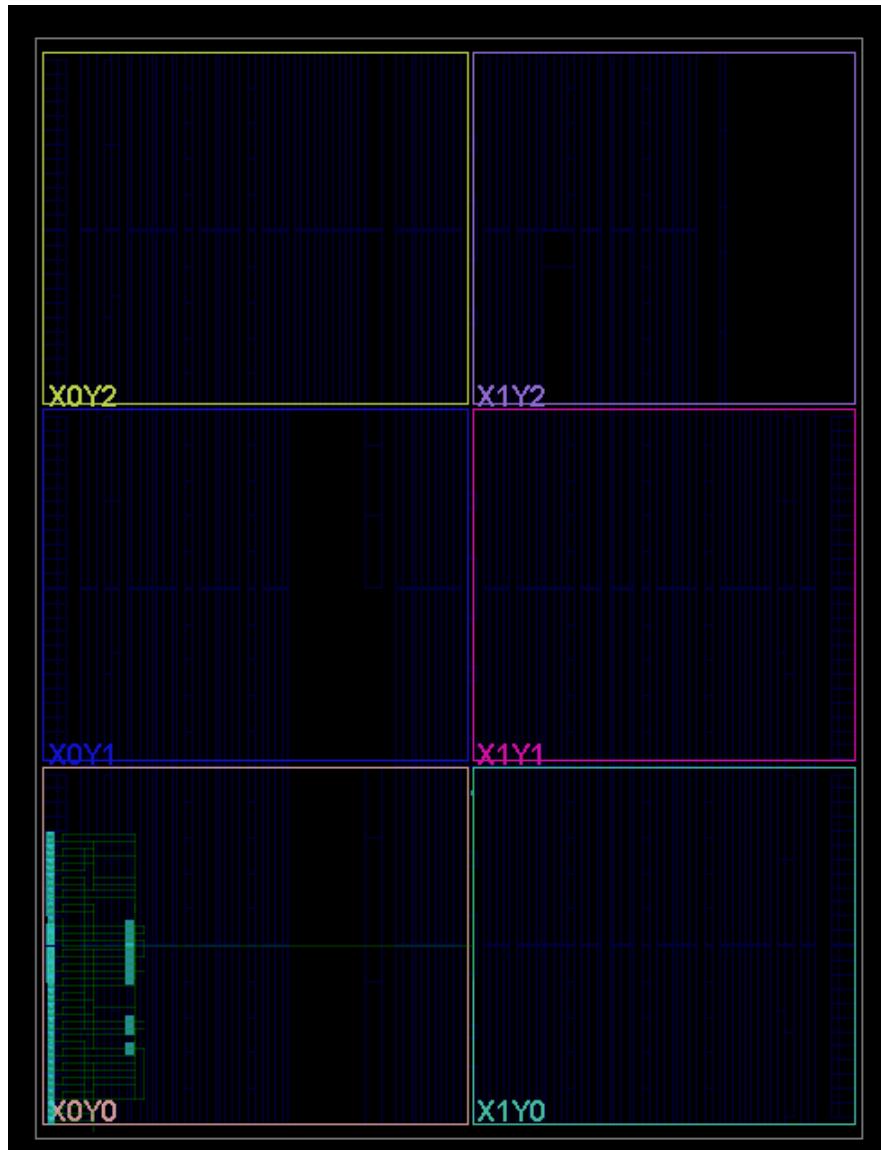


FIGURE 32 – Slice du FPGA après routage

En effectuant un zoom, il est possible de constater que le système a été implémenté dans la zone **0** du FPGA. Nous observons que la zone située à gauche correspond aux entrées de chaque variable, celles-ci étant toutes reliées à des **buffers**.

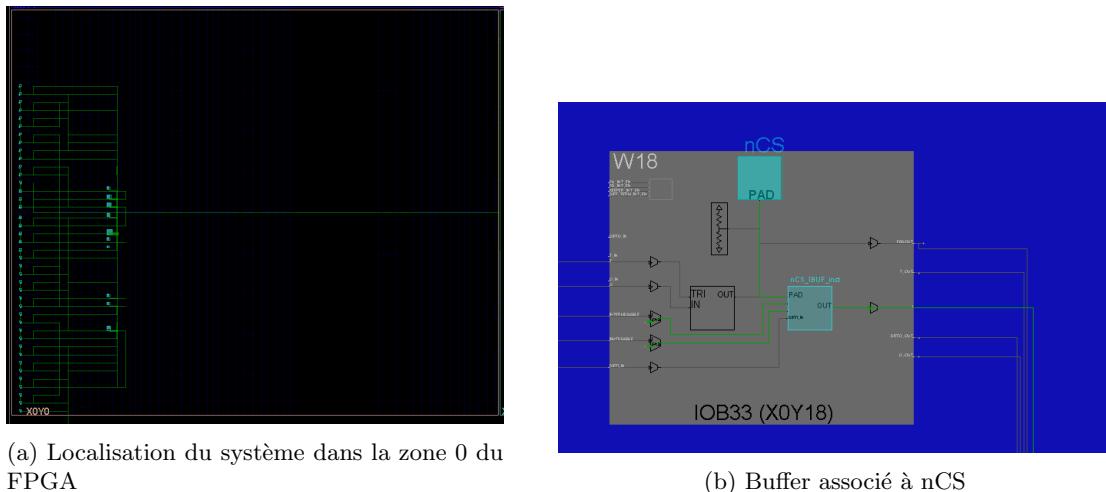


FIGURE 33 – Vue du système routé et des buffers associés sur le FPGA

Un zoom encore plus détaillé permet d'observer le câblage interne des ressources identifiées lors de la synthèse. Par exemple, une **LUT3** est câblée de la manière suivante :

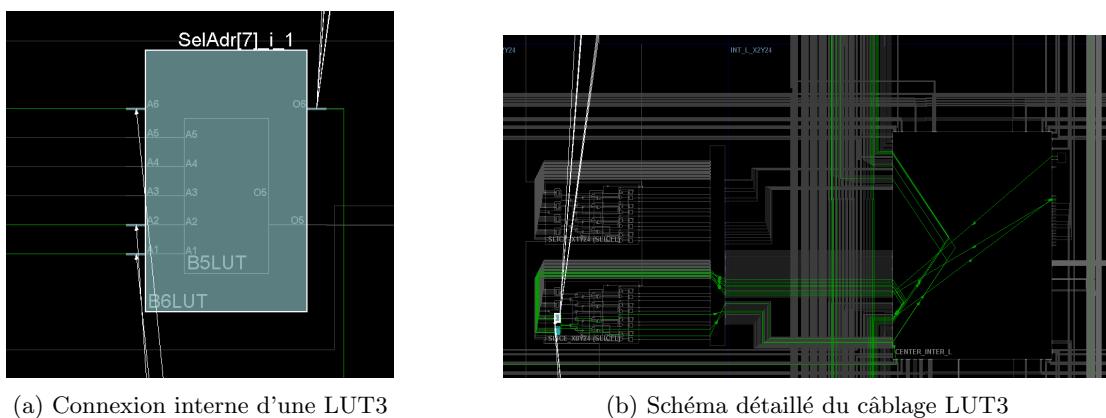


FIGURE 34 – Exemple de routage d'une LUT3 dans le FPGA

Le résultat final est une représentation complète et hiérarchisée du système, directement mappée sur le FPGA. **Vivado** offre ainsi la possibilité de visualiser l'ensemble du flot, depuis la description logique RTL jusqu'au routage physique détaillé.

En résumé, la conception suit une progression en trois étapes :

- la **synthèse** génère une description logique optimisée du système (LUT, registres, blocs fonctionnels) ;
- le **placement** attribue ces ressources aux cellules physiques du FPGA ;
- le **routage** établit les interconnexions nécessaires au bon fonctionnement du circuit.

Cette approche permet de passer d'une description abstraite en langage HDL à une implémentation matérielle concrète, où chaque fonction logique est traduite en ressources physiques. Vivado fournit alors une vision globale et détaillée du FPGA, allant de la logique combinatoire jusqu'au câblage interne des composants.

## 11 Conclusion

Pour ce premier rapport concernant le projet **Réception LIN** de conception de circuit numérique, nous avons suivi plusieurs étapes successives afin de concevoir ce système.

Dans un premier temps, lors des séances de TD, nous avons décomposé notre étude en plusieurs parties afin de répondre au cahier des charges. L'utilisation du **diagramme en Y** nous a permis d'analyser séparément chacune de ces étapes et de structurer notre démarche.

La phase de **spécification fonctionnelle** nous a permis de définir l'ensemble des ressources nécessaires, en lien direct avec le cahier des charges, ainsi que le nombre de blocs et les signaux de base.

Ensuite, la **solution architecturale** a permis de préciser chaque partie en les reliant à des modèles connus (machines séquentielles, machines de Moore ou machines de Mealy). Cette étape a été essentielle pour découper notre système en une partie opérative et une partie commande :

- la partie opérative a été conçue à partir de blocs logiques simples (multiplexeurs, bascules D, etc.) ;
- la partie commande a été décrite sous forme d'automates, associés à des machines de Moore ou de Mealy, afin d'obtenir une description claire et structurée.

Cette méthodologie nous a permis d'écrire un code plus simple, lisible et cohérent.

La phase de **simulation** a ensuite validé le fonctionnement du système en stimulant les différents ports d'entrée et en observant les sorties.

La phase de **synthèse** nous a permis de vérifier la logique interne du système à travers les schémas RTL générés. Ces schémas ont ensuite été exploités dans Vivado, qui a associé les différentes fonctions logiques aux **LUT**.

Enfin, l'**implémentation** nous a donné accès au routage sur FPGA. Cette étape a permis de vérifier concrètement l'affectation des ressources matérielles et le câblage interne du système.

En conclusion, ce premier travail nous a permis de valider l'**interface microprocesseur**. La suite du projet consistera à finaliser la conception complète du récepteur LIN afin de répondre à l'intégralité du cahier des charges.

## 12 Annexes

### 12.1 Testbench InterfaceMicroprocesseur

```

1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
3 USE ieee.std_logic_arith.all;
4
5 ENTITY EnvTest_InterfaceMicroprocesseur IS
6     GENERIC(
7         CLOCK_PERIOD      : time := 50 ns;
8         RESET_OFFSET     : time := 500 ns;
9         RESET_DURATION   : time := 300 ns;
10        ACCESS_TIME     : time := 40 ns;
11        HOLD_TIME       : time := 70 ns
12    );
13    PORT(
14        M_Received : IN      std_logic;
15        CnD        : OUT     std_logic;
16        H          : OUT     std_logic;
17        RnW        : OUT     std_logic;
18        nCS        : OUT     std_logic;
19        nRST       : OUT     std_logic;
20        D07        : INOUT   std_logic_vector (7 DOWNTO 0)
21    );
22    -- Declarations
23
24 END EnvTest_InterfaceMicroprocesseur ;
25
26 --
27 ARCHITECTURE arch OF EnvTest_InterfaceMicroprocesseur IS
28     TYPE DefState IS (Waiting, DataReading, StateReading, FilterWriting);
29
30     SIGNAL ProcessorState : DefState;
31
32 BEGIN
33
34     ClockGeneratorProc : PROCESS
35     BEGIN
36         H <= '0';
37         WAIT FOR CLOCK_PERIOD/2;
38         H <= '1';
39         WAIT FOR CLOCK_PERIOD/2;
40     END PROCESS ClockGeneratorProc;
41
42     ResetGeneratorProc : PROCESS
43     BEGIN
44         nRST <= '1';
45         WAIT FOR RESET_OFFSET;
46         nRST <= '0';
47         WAIT FOR RESET_DURATION;
48         nRST <= '1';
49         WAIT;
50     END PROCESS ResetGeneratorProc;
51
52     ProcessorBehaviorProc : PROCESS

```

```

53 BEGIN
54     D07 <= (others => 'Z');
55     --Waiting cycle--
56     ProcessorState <= Waiting;
57     nCS <= '1';
58     CnD <= '1';
59     RnW <= '1';
60     WAIT FOR RESET_OFFSET+RESET_DURATION+2*CLOCK_PERIOD;
61     --Reading data cycle--
62     ProcessorState <= DataReading;
63     WAIT FOR ACCESS_TIME;
64     nCS <= '0';
65     CnD <= '0';
66     RnW <= '1';
67     WAIT FOR 2*CLOCK_PERIOD;
68     --Waiting cycle--
69     ProcessorState <= Waiting;
70     nCS <= '1';
71     CnD <= '1';
72     RnW <= '1';
73     WAIT FOR 2*CLOCK_PERIOD-ACCESS_TIME;
74     --Reading state cycle--
75     ProcessorState <= StateReading;
76     WAIT FOR ACCESS_TIME;
77     nCS <= '0';
78     CnD <= '1';
79     RnW <= '1';
80     WAIT FOR 2*CLOCK_PERIOD;
81     --Waiting cycle--
82     ProcessorState <= Waiting;
83     nCS <= '1';
84     CnD <= '1';
85     RnW <= '1';
86     WAIT FOR 2*CLOCK_PERIOD-ACCESS_TIME;
87     --Writing cycle--
88     ProcessorState <= FilterWriting;
89     WAIT FOR ACCESS_TIME;
90     nCS <= '0';
91     CnD <= '0';
92     RnW <= '0';
93     D07 <= (others => '1');
94     WAIT FOR 2*CLOCK_PERIOD;
95     --Waiting cycle--
96     ProcessorState <= Waiting;
97     nCS <= '1';
98     CnD <= '1';
99     RnW <= '1';
100    WAIT FOR HOLD_TIME;
101    D07 <= (others => 'Z');
102    WAIT;
103 END PROCESS ProcessorBehaviorProc;
104
105 END ARCHITECTURE arch;

```

Listing 5 – Testbench InterfaceMicroprocesseur

## Table des figures

1	Exemple d'architecture d'un réseau dans un véhicule . . . . .	3
2	Exemple d'architecture LIN . . . . .	4
3	Connexion physique d'un noeud à la ligne LIN . . . . .	4
4	Type de Trame Protocol LIN . . . . .	5
5	Interface microprocesseur associée au circuit à concevoir . . . . .	7
6	Chronogrammes des échanges entre le circuit et son environnement . . . . .	7
7	Schema Conception Registre interne Système . . . . .	8
8	Interface microprocesseur associée au circuit à concevoir . . . . .	9
9	Structure fonctionnelle du système . . . . .	11
10	Architecture du système de réception de trame LIN V1 . . . . .	11
11	Échanges avec le processeur . . . . .	12
12	Architecture du système de réception de trame LIN V2 . . . . .	12
13	Description finale du circuit . . . . .	13
14	Machine Sequentielle Reception Trame . . . . .	16
15	Structure partie Opérative Reception Trame . . . . .	17
16	Structure partie Commande Reception Trame . . . . .	18
17	Machine de MEALY Unité de Commande Reception Trame . . . . .	19
18	Machine Sequentielle Reception Trame . . . . .	20
19	Structure partie Opérative Interface Microprocesseur . . . . .	20
20	Structure partie Commande Interface Microprocesseur . . . . .	21
21	Machine de MEALY Unité de Commande Interface Micro . . . . .	21
22	Implémentation de la FIFO . . . . .	22
23	implémentation de l'état interne en représentation structurelle au niveau RT . . . . .	23
24	Chronogramme de simulation de l'Interface Microprocesseur . . . . .	27
25	Block Diagramme de test de l'Interface Microprocesseur . . . . .	28
26	Schéma RTL InterfaceMicroprocesseur . . . . .	30
27	Partie opérative avec multiplexeur et Tristate : InterfaceMicroprocesseur . . . . .	31
28	Synthèse matérielle InterfaceMicroprocesseur . . . . .	31
29	Synthèse matérielle Vivado . . . . .	33
30	Illustration d'une LUT et de sa table de vérité . . . . .	34
31	Synthèse Logique Vivado . . . . .	34
32	Slice du FPGA après routage . . . . .	35
33	Vue du système routé et des buffers associés sur le FPGA . . . . .	36
34	Exemple de routage d'une LUT3 dans le FPGA . . . . .	36