Rapport Final - Conception Circuit Numérique

Tony PEAULT & Nolan BUCHET

Novembre 2025

POLYTECH NANTES | Pôle Sciences et technologie Nantes Université

Recepteur de Transmission LIN

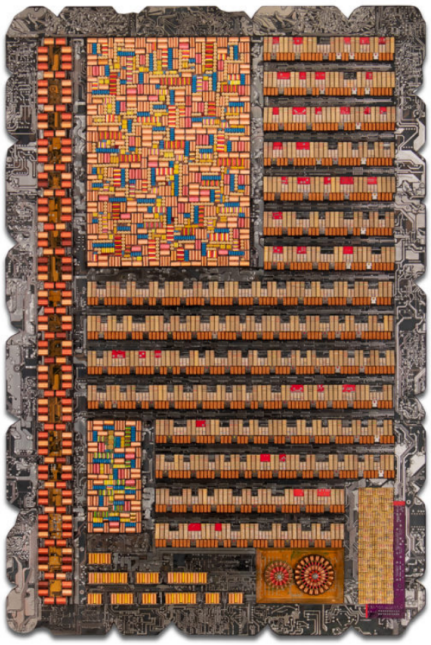


Table des matières

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.

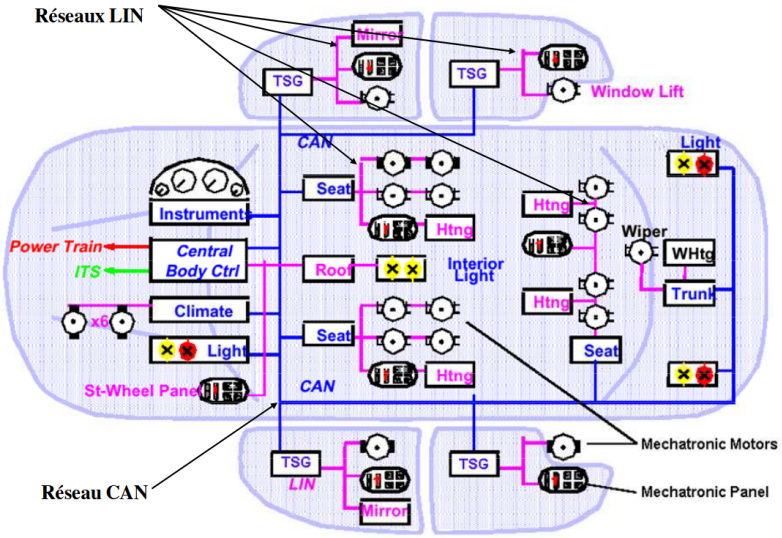


FIGURE 1 – Exemple d'architecture d'un réseau dans un véhicule

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 nœuds esclaves sans modifier les nœuds existants.

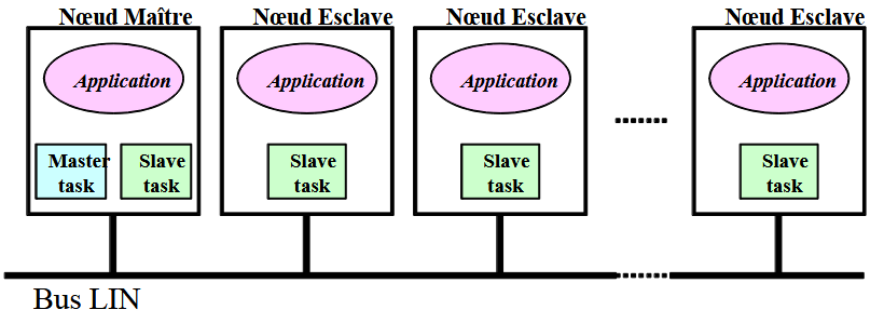


FIGURE 2 – Exemple d'architecture LIN

Connexion

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

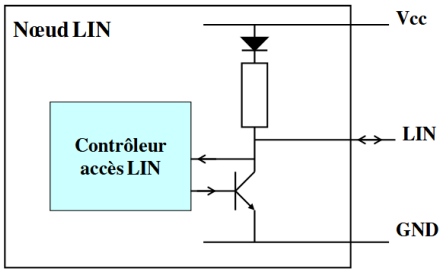


FIGURE 3 – Connexion physique d'un noeud à la ligne LIN

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).

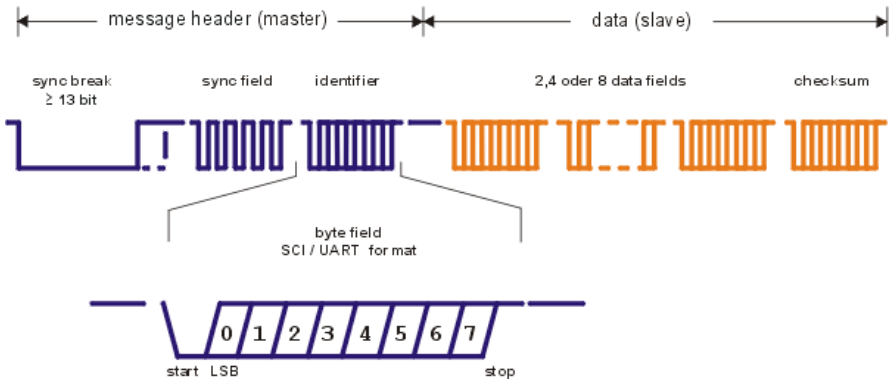


FIGURE 4 – Type de Trame Protocol LIN

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 `SelAdr` (8 bits),
- Signalisation de fin de réception (`M_Received`) uniquement si l'identifiant reçu correspond à `SelAdr`,
- 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.

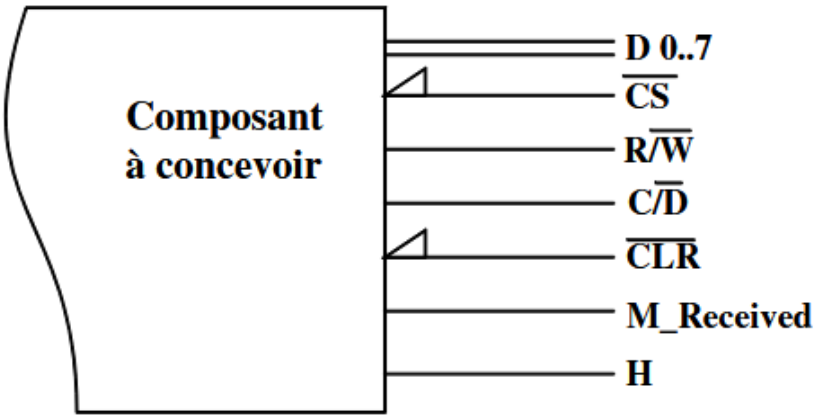


FIGURE 5 – Interface microprocesseur associée au circuit à concevoir

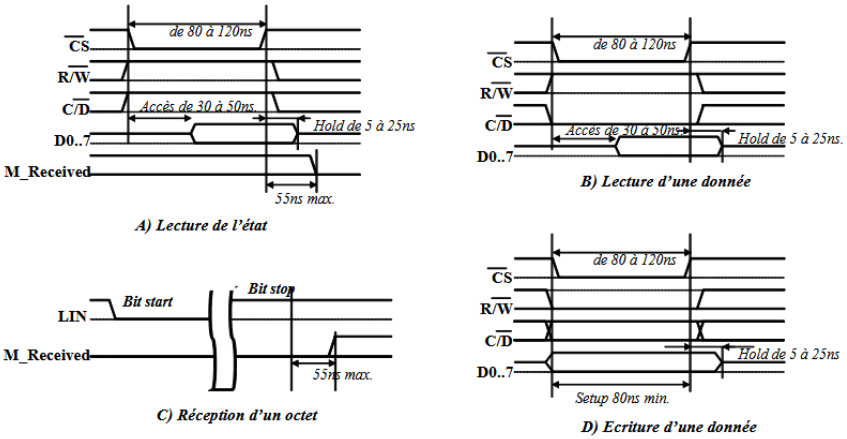


FIGURE 6 – Chronogrammes des échanges entre le circuit et son environnement

Ces figures illustrent les interfaces et les chronogrammes des échanges entre le circuit à concevoir et son environnement, mettant en évidence les interactions avec le microprocesseur et les timings associés.

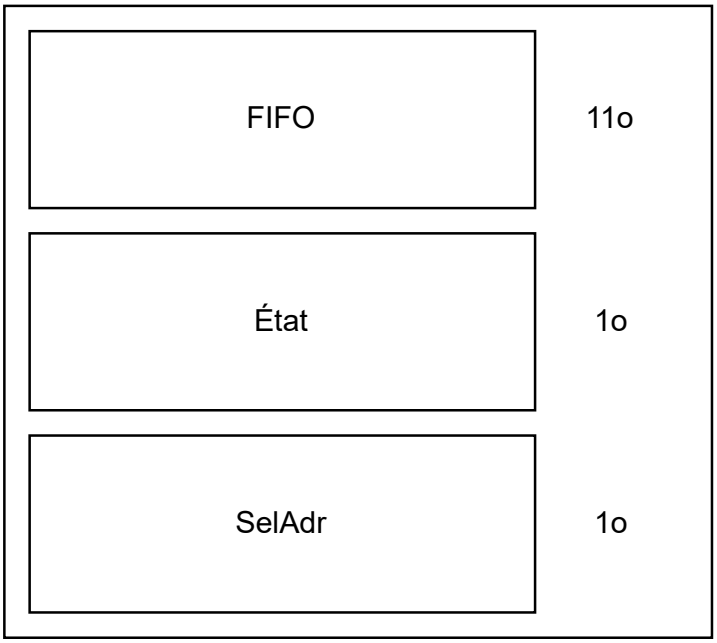


FIGURE 7 – Schema Conception Registre interne Système

Ce schéma détaille la conception des registres internes du système, incluant les registres de données, d'état et de sélection d'adresse.

4 Description des différentes spécifications définies en travaux dirigés

Objectif

Les spécifications ont pour objectif de définir le comportement attendu du système, c'est-à-dire **ce que le circuit doit faire** en réponse au cahier des charges. Elles constituent une description **fonctionnelle** du système, exprimée du point de vue de son **environnement** — c'est-à-dire de tout ce qui interagit avec lui, sans se soucier de son implémentation interne.

Cette phase correspond au **niveau de spécification fonctionnelle** dans le diagramme en Y. Elle adopte une approche **boîte noire**, centrée sur les entrées et sorties observables, indépendamment de toute considération technologique (langage, type logique, fréquence, etc.).

Le cahier des charges indique que le circuit doit pouvoir **communiquer à la fois avec le système de trame LIN et avec un microcontrôleur**. Afin de clarifier les fonctions du système, nous avons choisi de le **décomposer en deux sous-blocs principaux** :

- un bloc de **réception de trame LIN**, chargé de décoder et de stocker les données reçues ;
- un bloc d'**interface microprocesseur**, permettant l'échange de données et de signaux de contrôle avec le microcontrôleur.

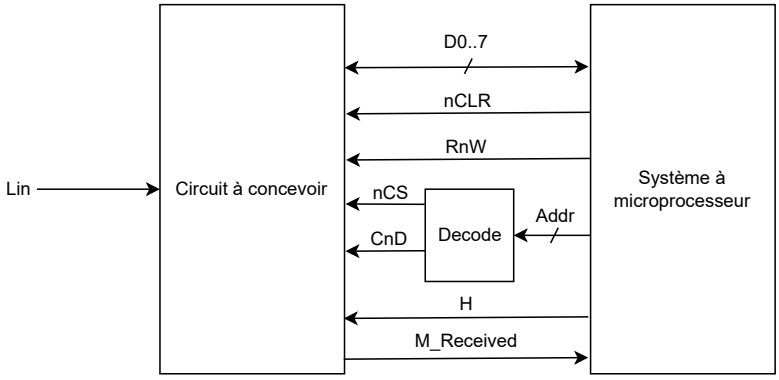


FIGURE 8 – Description fonctionnelle du circuit à concevoir

4.1 Interface Microprocesseur

Ce sous-système permet la communication entre le circuit et le microprocesseur. Les signaux décrits ici représentent les **flux d'informations échangés** (données, commandes, synchronisation, validation), sans spécifier leur codage logique ni leur type de signal électrique.

Signal	Sens	Nature	Rôle fonctionnel
D_BUS	Bidirectionnel	Données	Bus de transfert de données entre le microprocesseur et le circuit
CS	Entrée	Commande	Sélection du circuit (validation de la communication)
RW	Entrée	Commande	Indique une opération de lecture ou d'écriture
CD	Entrée	Commande	Sélectionne entre registre de commande et registre de données
RESET	Entrée	Commande	Réinitialisation du système
MSG_RECEIVED	Sortie	Indicateur	Signal indiquant la fin de réception d'une trame
CLK	Entrée	Synchronisation	Signal d'horloge du système

4.2 Bloc Réception de Trame LIN

Ce bloc assure le **décodage séquentiel** des trames LIN reçues. Il analyse le flux série provenant du bus LIN et extrait les octets de données en respectant la structure du protocole.

Signal	Sens	Nature	Rôle fonctionnel
LIN_RX	Entrée	Données	Flux série reçu depuis le bus LIN
DATA_OUT	Sortie	Données	Octet de données extrait et validé
VALID	Sortie	Indicateur	Indique la disponibilité d'un nouvel octet reçu

4.3 Mémoire FIFO

Ce bloc a pour rôle de **stocker temporairement les octets reçus** avant leur transfert vers le microprocesseur. Il fonctionne selon le principe « premier entré, premier sorti ».

Signal	Sens	Nature	Rôle fonctionnel
DATA_IN	Entrée	Données	Octet à mémoriser dans la file FIFO
DATA_OUT	Sortie	Données	Octet extrait de la file FIFO
WRITE_REQ	Entrée	Commande	Requête d'écriture (nouvelle donnée reçue)
READ_REQ	Entrée	Commande	Requête de lecture (demande du microprocesseur)
EMPTY	Sortie	Indicateur	Indique que la FIFO est vide
FULL	Sortie	Indicateur	Indique que la FIFO est pleine

4.4 Registre d'État

Le registre d'état fournit une **synthèse du déroulement de la réception**. Il conserve les informations nécessaires à la supervision ou au diagnostic (erreurs détectées, nombre d'octets reçus, trame complète, etc.).

Signal	Sens	Nature	Rôle fonctionnel
ERR_START	Entrée	Indicateur	Erreur sur le bit de début de trame
ERR_STOP	Entrée	Indicateur	Erreur sur le bit de fin de trame
ERR_SYNC	Entrée	Indicateur	Erreur de synchronisation
BYTE_COUNT	Entrée	Données	Nombre d'octets reçus dans la trame
FRAME_VALID	Entrée	Indicateur	Validation de la réception complète
STATE_OUT	Sortie	Données	Octet d'état global de la réception

5 Description et justification de la structure fonctionnelle

Objectifs

Cette section présente l'organisation fonctionnelle du système et la répartition des rôles entre les différents sous-ensembles. Chaque bloc (réception de trame, mémoire FIFO, registre d'état, interface microprocesseur) est décrit dans sa fonction et ses interactions avec les autres. L'objectif est de montrer comment les fonctionnalités définies lors de la spécification sont structurées logiquement pour répondre au cahier des charges, tout en restant indépendantes de toute technologie d'implémentation.

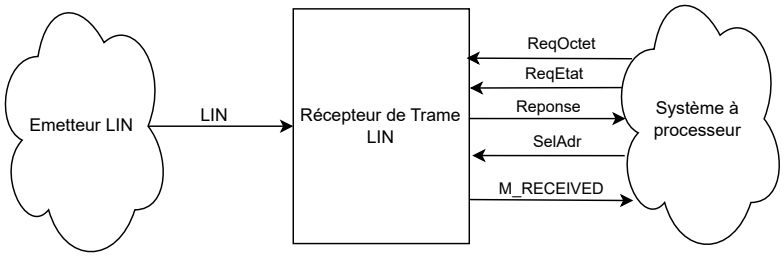


FIGURE 9 – Représentation fonctionnelle des échanges entre l'émetteur LIN et le système à processeur

À ce stade, le système est structuré autour de deux blocs principaux : l'interface microprocesseur et la réception des trames LIN. Ces deux blocs communiquent via un bloc d'échange chargé d'assurer la cohérence des transferts d'informations et la coordination entre les différents registres internes.

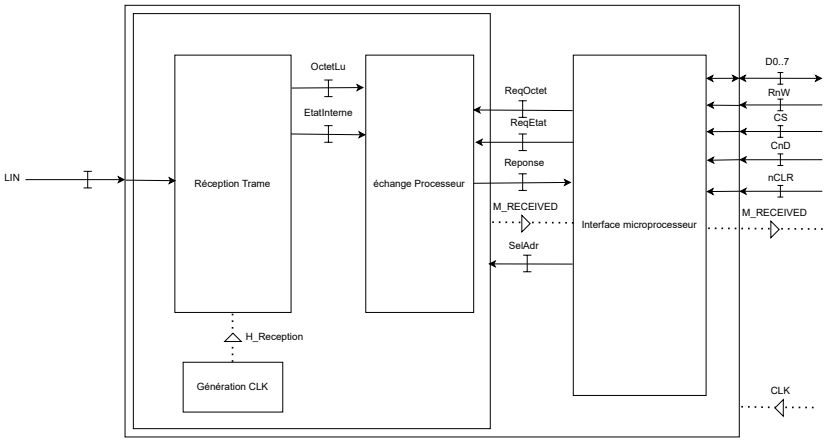


FIGURE 10 – Structure fonctionnelle initiale du circuit après introduction des interfaces

Le comportement général peut être représenté par un automate de communication illustrant les échanges avec le processeur :

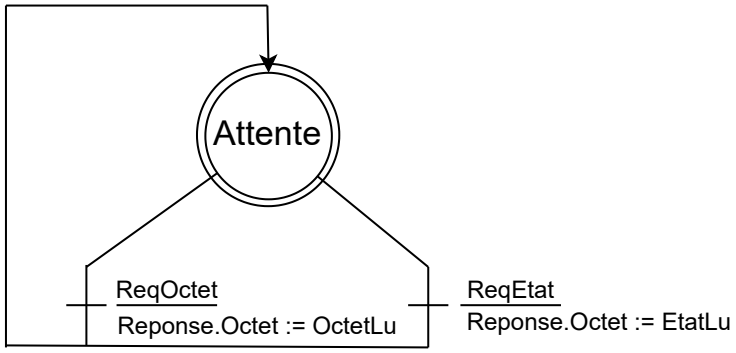


FIGURE 11 – Échanges fonctionnels entre le système et le processeur

Lors de la phase d'analyse, il est apparu que le bloc d'échange microprocesseur pouvait être intégré directement à l'interface microprocesseur. Cette simplification permet de réduire le nombre de signaux intermédiaires et d'améliorer la clarté fonctionnelle du système.

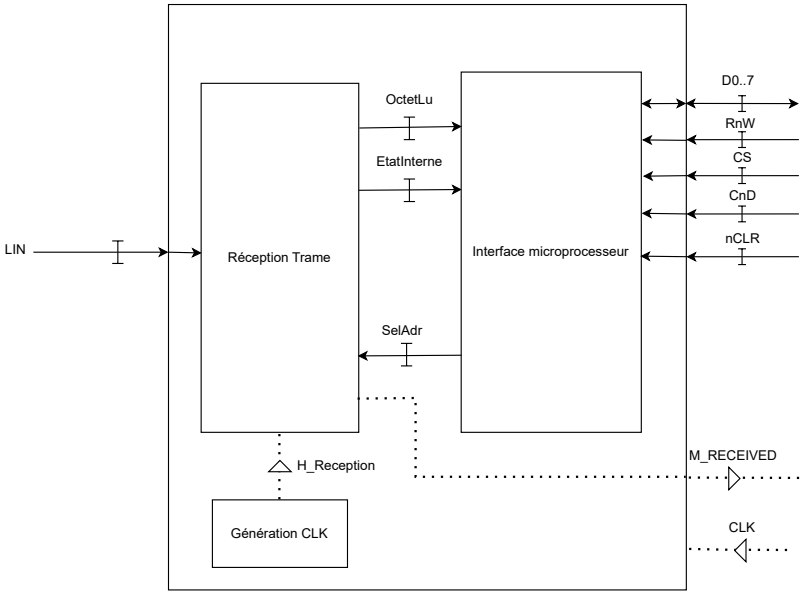


FIGURE 12 – Architecture fonctionnelle optimisée du système de réception de trame LIN

Enfin, deux registres internes ont été ajoutés :
— un registre de stockage des données de trame (FIFO) ;
— un registre d'état interne (ETAT), contenant les informations de suivi et d'erreur.

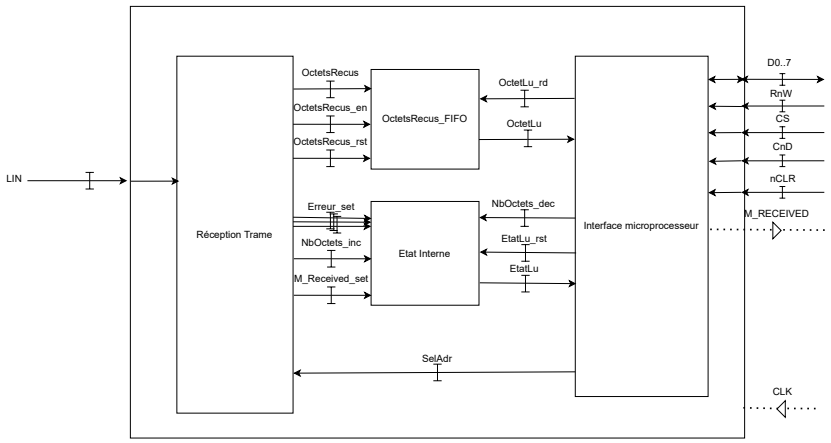


FIGURE 13 – Description fonctionnelle finale du circuit complet

Le schéma global ci-dessus illustre les interactions fonctionnelles entre les différents blocs du système. Les échanges sont exprimés en termes de flux d'informations (données, ordres, signaux de contrôle), sans référence à la nature physique ou logique de ces signaux.

Bloc FIFO

Signal	Sens	Rôle fonctionnel
Écriture_Octet	Entrée	Déclenche l'enregistrement d'un octet dans la mémoire FIFO.
Réinitialisation_FIFO	Entrée	Vide la mémoire FIFO et remet à zéro les compteurs internes.
Lecture_Octet	Entrée	Permet l'accès séquentiel aux données stockées dans la mémoire FIFO.

Ces signaux assurent la gestion du flux d'informations entre la réception de trame et le microprocesseur. Ils garantissent la synchronisation et la fiabilité du stockage des données reçues, en évitant toute perte ou chevauchement.

Bloc ÉTAT

Signal	Sens	Rôle fonctionnel
Réinitialisation_Compteur	Entrée	Réinitialise le nombre d'octets reçus.
Décrémentation_Compteur	Entrée	Indique qu'un octet a été lu depuis la FIFO.
Réinitialisation_État	Entrée	Remet à zéro les indicateurs d'état et d'erreur après lecture.

Ces signaux permettent le suivi interne du processus de réception et la gestion des informations d'erreur. Ils facilitent la communication avec le microprocesseur tout en assurant une supervision fiable et indépendante de toute implémentation matérielle.

6 Description et justification de la solution architecturale obtenue pour le circuit

Objectifs

Une fois la description fonctionnelle définie, la conception passe au **niveau architectural** du diagramme en Y. Cette phase vise à transformer la description fonctionnelle en une organisation interne du système : elle introduit les interfaces physiques, identifie les ressources de stockage et de traitement, et établit les principes de commande et de transfert des données.

Cette description reste indépendante de la technologie d'implémentation (langage HDL, logique, FPGA, etc.) mais tient compte des contraintes structurelles et temporelles du système. Elle correspond au **niveau Registre-Transfert (RT)**.

- Introduction et définition des interfaces physiques
- Identification des ressources logiques (registres, compteurs, opérateurs)
- Organisation structurelle du circuit au niveau RT
- Description du comportement séquentiel et des signaux de commande

6.1 Horloge

La gestion de l'horloge constitue un élément fondamental de la synchronisation interne du système.

Le rapport entre la période du bit LIN et celle du processeur est défini par :

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

Dans notre cas, le cahier des charges spécifie un cycle de lecture/écriture moyen de 100 ns et une vitesse de transmission de 19 200 bit/s, soit :

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

Pour notre implémentation, nous choisissons $N = 2048$, une valeur supérieure qui facilite la synchronisation interne et la gestion des transitions logiques.

6.2 Architecture de la Réception de Trame

Ce bloc correspond à la partie du système chargée de la réception et du décodage des trames LIN.

Signal	Sens	Nature	Rôle fonctionnel
LIN	Entrée	Données	Signal série reçu depuis le bus LIN
SEL_ADR	Entrée	Données	Sélection de l'adresse du composant
DATA_OUT	Sortie	Données	Octet de données reçu
DATA_WR	Sortie	Commande	Validation d'écriture vers la mémoire FIFO
DATA_RST	Sortie	Commande	Réinitialisation des données reçues
ERR_START	Sortie	Indicateur	Erreur sur bit de start
ERR_STOP	Sortie	Indicateur	Erreur sur bit de stop
ERR_SYNC	Sortie	Indicateur	Erreur de synchronisation (Synchro Break)
INC_COUNT	Sortie	Commande	Incrémentation du compteur d'octets reçus
FRAME_VALID	Sortie	Indicateur	Trame reçue et validée
COUNT_RST	Sortie	Commande	Réinitialisation du compteur d'octets

Le bloc de réception repose sur une **machine séquentielle** structurée en deux sous-parties :
— une **unité opérative** regroupant les registres, compteurs et multiplexeurs ;
— une **unité de commande** gérant la séquence d'opérations et les signaux de contrôle.

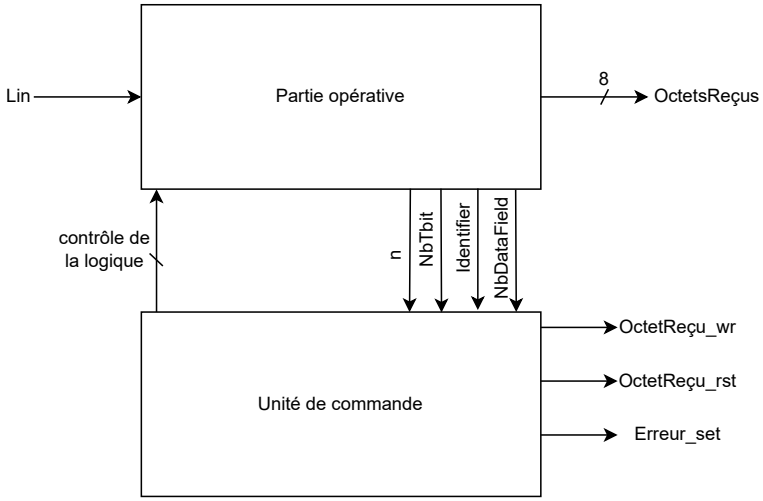


FIGURE 14 – Organisation séquentielle du bloc de réception de trame

Les principales variables internes assurent la gestion du comptage, du stockage et du décalage des bits reçus :

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

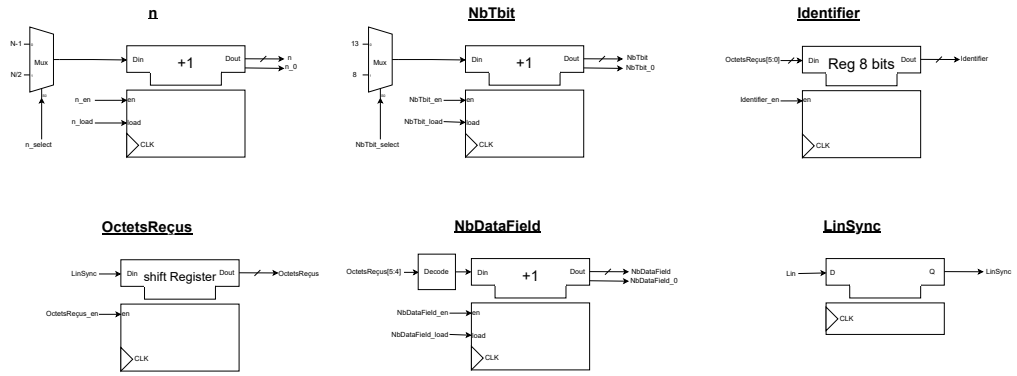


FIGURE 15 – Structure opérative du bloc de réception de trame

La partie commande est implémentée sous forme d'un **automate séquentiel**, représentant les différents états de réception d'une trame LIN.

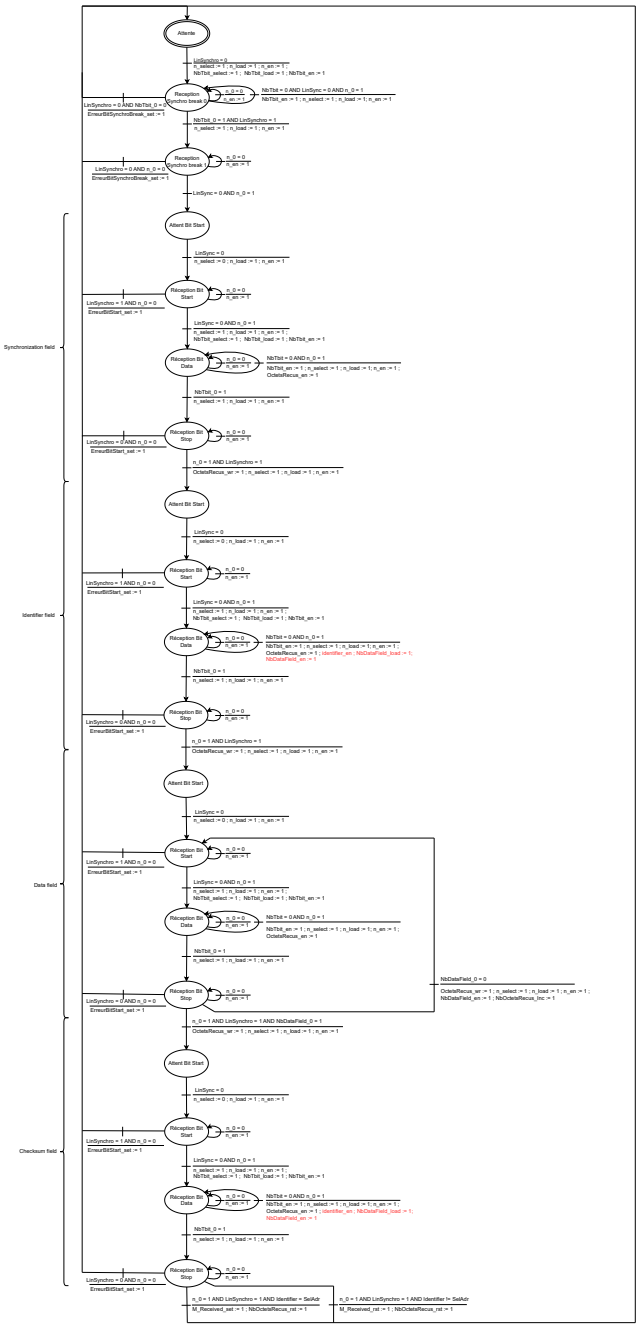


FIGURE 16 – Automate de réception de trame LIN

Description de l'automate

L'automate décrit la séquence d'opérations depuis l'attente du signal de début jusqu'à la validation de la trame complète. Il gère successivement les phases suivantes :

- **Attente et détection de break de synchronisation**
- **Réception du champ de synchronisation**

— **Réception de l'identifiant et des données**
— **Vérification du checksum et validation de la trame**

Les erreurs de synchronisation ou de bits sont détectées via des indicateurs spécifiques (erreurs de start, stop, ou synchro). Ce fonctionnement correspond à une **machine de Mealy**, dans laquelle les sorties dépendent à la fois des états internes et des entrées instantanées.

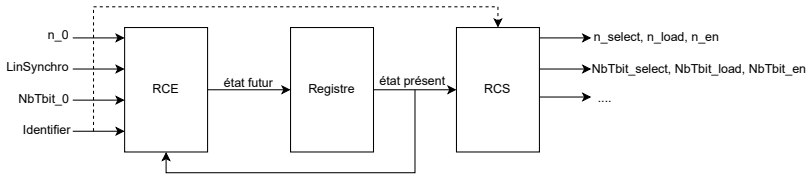


FIGURE 17 – Machine de Mealy – Unité de commande de réception de trame

6.3 Architecture de l'Interface Microprocesseur

Ce bloc gère les échanges entre le microprocesseur et les registres internes du système. Il coordonne la lecture et l'écriture des données, ainsi que la signalisation de fin de réception.

Signal	Sens	Nature	Rôle fonctionnel
D_BUS	Bidirectionnel	Données	Bus de communication principal
CS	Entrée	Commande	Sélection du circuit
RW	Entrée	Commande	Lecture ou écriture
CD	Entrée	Commande	Sélection entre commande et données
RESET	Entrée	Commande	Réinitialisation du système
FRAME_RECEIVED	Sortie	Indicateur	Signal de fin de réception
CLK	Entrée	Synchronisation	Horloge du système
STATE_IN	Entrée	Données	État interne du système
DEC_COUNT	Sortie	Commande	Décrémentation du compteur FIFO
STATE_RST	Sortie	Commande	Réinitialisation de l'état
DATA_IN	Entrée	Données	Donnée issue de la FIFO
DATA_SEL	Sortie	Commande	Sélection du type de donnée affichée

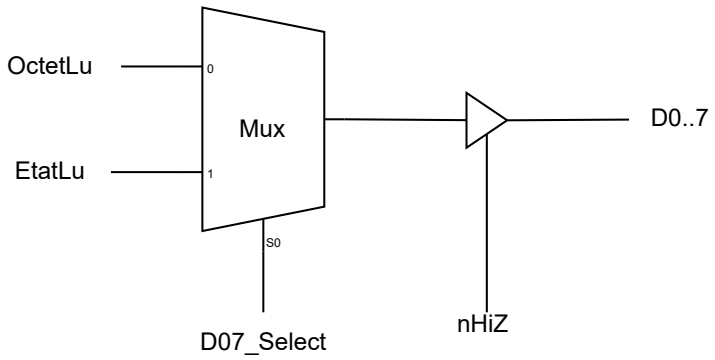


FIGURE 18 – Structure opérative de l'interface microprocesseur

L'unité de commande correspondante est également décrite par un automate de type Mealy :

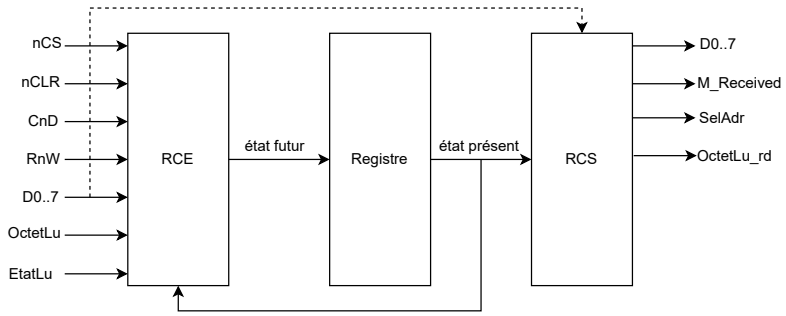


FIGURE 19 – Machine de Mealy – Interface microprocesseur

6.4 Architecture de la Mémoire FIFO

La FIFO assure le stockage temporaire des octets reçus. Étant de complexité limitée, elle est décrite directement sous forme structurelle.

Signal	Sens	Nature	Rôle fonctionnel
DATA_IN	Entrée	Données	Données reçues à stocker
WRITE	Entrée	Commande	Validation d'écriture
RESET	Entrée	Commande	Réinitialisation du contenu
DATA_OUT	Sortie	Données	Données lues
READ	Entrée	Commande	Validation de lecture

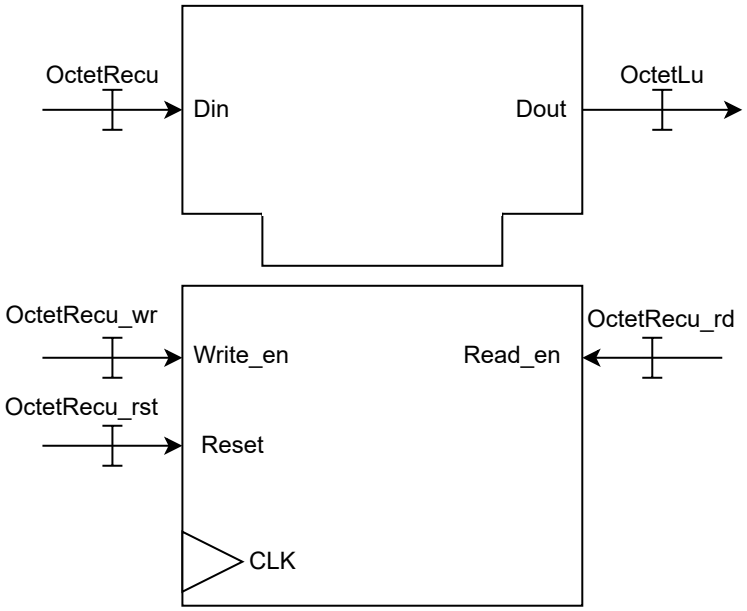
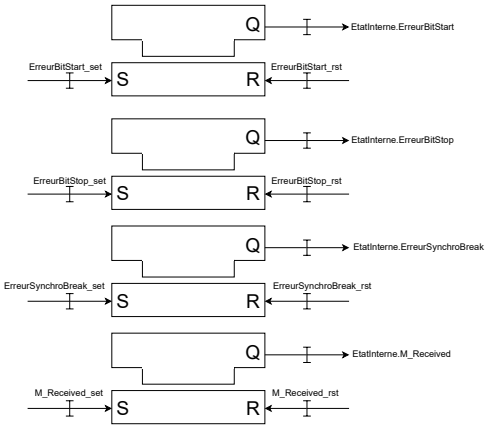


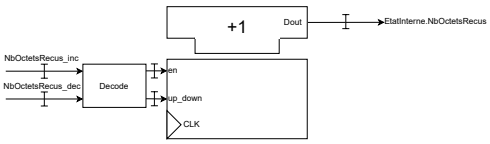
FIGURE 20 – Implémentation structurelle de la mémoire FIFO

6.5 Implémentation du Registre d'État

Le registre d'état regroupe les informations relatives aux erreurs, au nombre d'octets reçus et à la validation des trames.



(a) Mémorisation des erreurs détectées



(b) Comptage des octets reçus

FIGURE 21 – Implémentation structurelle du registre d'état 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 Synchronisation des Entrées

```
InputProc_Synchro :  PROCESS(H, nRST)
BEGIN
  IF (nRST='0') THEN
    nCS_Synchro <= '1';
    RnW_Synchro <= '1';
    CnD_Synchro <= '1';
    D07_Synchro <= (others => '0');
  ELSIF (H'EVENT AND H='1') THEN
    nCS_Synchro <= nCS;
    RnW_Synchro <= RnW;
    CnD_Synchro <= CnD;
    D07_Synchro <= D07;
  END IF;
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

```
OutputProc_Comb : PROCESS(nCS_Synchro, CnD_Synchro, RnW_Synchro,
    EtatCourant, OctetLu, EtatLu)
BEGIN
  D07 <= (others => 'Z');
  OctetLu_RD <= '0';
  EtatLu_RST <= '0';
  DecNbOctet <= '0';
  CASE EtatCourant IS
    WHEN Attente =>
      IF (nCS_Synchro='0' AND CnD_Synchro='0' AND RnW_Synchro='1') THEN
        OctetLu_RD <= '1';
      END IF;
    WHEN LectureData =>
```

```
      D07 <= OctetLu;
      IF (nCS_Synchro='1') THEN
        DecNbOctet <= '1';
      END IF;
    WHEN LectureEtat =>
      D07 <= EtatLu;
      IF (nCS_Synchro='1') THEN
        EtatLu_RST <= '1';
      END IF;
    WHEN EcritureFiltre =>
    END CASE;
END PROCESS OutputProc_Comb;
```

Listing 2 – Reseau Cominatoire 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`, `EtatLu_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 Réseau Combinatoire d'Entrée

```
ClockedProc : PROCESS(H, nRST)
BEGIN
  IF (nRST='0') THEN
    EtatCourant <= Attente;
  ELSIF (H'EVENT AND H='1') THEN
    EtatCourant <= EtatSuivant;
  END IF;
END PROCESS ClockedProc;

NextStateProc : PROCESS(nCS_Synchro, CnD_Synchro, RnW_Synchro,
    EtatCourant)
BEGIN
  EtatSuivant <= EtatCourant;
  CASE EtatCourant IS
  WHEN Attente =>
    IF (nCS_Synchro='0' AND CnD_Synchro='0' AND RnW_Synchro='1') THEN
      EtatSuivant <= LectureData;
    ELSIF (nCS_Synchro='0' AND CnD_Synchro='1' AND RnW_Synchro='1') THEN
      EtatSuivant <= LectureEtat;
    ELSIF (nCS_Synchro='0' AND CnD_Synchro='0' AND RnW_Synchro='0') THEN
      EtatSuivant <= EcritureFiltre;
    ELSE
      EtatSuivant <= Attente;
    END IF;
    WHEN LectureData =>
      IF (nCS_Synchro='1') THEN
        EtatSuivant <= Attente;
      ELSE
        EtatSuivant <= LectureData;
      END IF;
    WHEN LectureEtat =>
      IF (nCS_Synchro='1') THEN
        EtatSuivant <= Attente;
      ELSE
        EtatSuivant <= LectureEtat;
```

```
      END IF;
    WHEN EcritureFiltre =>
      IF (nCS_Synchro='1') THEN
        EtatSuivant <= Attente;
      ELSE
        EtatSuivant <= EcritureFiltre;
      END IF;
  END CASE;
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

```
OutputProc_Synchro : PROCESS(H, nCLR)
BEGIN
  IF (nCLR='0') THEN
    SelAdr <= (others => '0');
  ELSIF (H'EVENT AND H='1') THEN
    CASE EtatCourant IS
    WHEN EcritureFiltre =>
      IF (nCS_Synchro='1') THEN
        SelAdr <= D07_Synchro;
      END IF;
    WHEN OTHERS =>
    END CASE;
  END IF;
END PROCESS OutputProc_Synchro;

M_Received <= EtatLu(4);
```

Listing 4 – Reseau 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.

7.2 Interface de Réception LIN

L'interface de réception LIN a été étudiée sous la forme d'une machine séquentielle, composée d'une partie opérative et d'une partie commande. La partie opérative est développée sous la forme d'un schéma fonctionnel comprenant différents blocs tels que des multiplexeurs et des compteurs/décompteurs. La partie commande, quant à elle, a été modélisée sous la forme d'un automate, traduit en machine de Moore, puis implémenté en VHDL.

Voici un tableau recapitulant les entrées et les sorties du bloc Repetion LIN complet :

Signaux	Mode	Type	Description
LIN	IN	STD_LOGIC	Bus de données d'entrée
SelAdr	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Sélection Adrress Composant
OctetRecu	OUT	STD_LOGIC_VECTOR(7 DOWNTO 0)	Bus de données de sortie
OctetRecu_WR	OUT	STD_LOGIC	Read / Write opération
OctetRecu_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
NbOctetRecu_RST	OUT	STD_LOGIC	Réinitialisation du compteur d'octets

7.2.1 Partie opérative

La partie opérative, déjà définie dans la section Description de la solution architecturale, a été reprise sous forme de blocs fonctionnels. Il a simplement été nécessaire de reproduire le schéma global dans HDL Designer, afin d'assurer la cohérence entre la conception théorique et la modélisation pratique.

Le schéma correspondant est présenté ci-dessous :

Voici aussi le tableau des entrées et sorties de la partie opérative :

Signaux	Mode	Type	Description
H	IN	STD_LOGIC	Horloge principale du système.
Identifier	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Identifiant du message reçu à comparer avec l'adresse sélectionnée.
LinSynchro	IN	STD_LOGIC	Signal de synchronisation de trame LIN.
NbTbit_0	IN	STD_LOGIC	Bit de configuration du nombre de bits de trame.
NbDataField_0	IN	STD_LOGIC	Bit de configuration du nombre d'octets de données dans le champ Data.
SelAdr	IN	STD_LOGIC_VECTOR(7 DOWNTO 0)	Sélection de l'adresse du composant (comparée à Identifier).
nCLR	IN	STD_LOGIC	Signal de réinitialisation asynchrone active à l'état bas.
n_0	IN	STD_LOGIC	Signal de contrôle interne (sélection ou validation).
Error_Start	OUT	STD_LOGIC	Bit d'erreur sur le champ Start.
Error_Stop	OUT	STD_LOGIC	Bit d'erreur sur le champ Stop.
Error_Synchro	OUT	STD_LOGIC	Bit d'erreur de synchronisation (Synchro Break).
Identifieur_en	OUT	STD_LOGIC	Validation de l'identifiant reçu.
IncNbOctet	OUT	STD_LOGIC	Incrémentation du compteur d'octets reçus.
MessageReceiveSet	OUT	STD_LOGIC	Indique qu'une trame complète a été reçue.
NbDataField_EN	OUT	STD_LOGIC	Activation du champ de données (Data Field).
NbDataField_load	OUT	STD_LOGIC	Chargement du nombre d'octets de données.
NbOctetRecu_RST	OUT	STD_LOGIC	Réinitialisation du compteur d'octets reçus.
OctetRecu_RST	OUT	STD_LOGIC	Réinitialisation du registre d'octet reçu.
OctetRecu_WR	OUT	STD_LOGIC	Signal d'écriture de l'octet reçu.
OctetRecu_en	OUT	STD_LOGIC	Validation du registre d'octet reçu.
n_Tbit_Load	OUT	STD_LOGIC	Chargement du registre associé au Tbit.
n_Tbit_en	OUT	STD_LOGIC	Activation du registre Tbit.
n_Tbit_select	OUT	STD_LOGIC	Sélection de la source ou mode du Tbit.
n_en	OUT	STD_LOGIC	Activation du signal ou compteur "n".
n_load	OUT	STD_LOGIC	Chargement de la valeur "n".
n_select	OUT	STD_LOGIC	Sélection de la source du signal "n".

7.2.2 Partie Commande

La partie commande consiste principalement à **traduire l'automate** présenté en figure **??** en **code VHDL**. Cette étape reste relativement simple, car elle repose sur la création de **trois processus principaux** :

1. **Le réseau combinatoire d'entrée**, chargé de déterminer les *états futurs* de l'automate en fonction des *états présents* et des *signaux d'entrée*.
2. **Le réseau de registres**, synchronisé sur l'horloge, permettant de *mémoriser les états* et d'assurer la transition entre les *états présents* et les *états futurs*.
3. **Le réseau combinatoire de sortie**, qui met à jour les *signaux de sortie* en fonction de l'état courant de l'automate.

Voici le code suivant qui permet de traduire l'automate :

```
-- Etat de la machine
type CFM is (
    R_BRK_0, R_BRK_1,                   -- Reception Break
    SYN_A, SYN_RST, SYN_RD, SYN_RSP,    -- Synchro
    IDN_A, IDN_RST, IDN_RD, IDN_RSP,    -- Identifier
    DAT_A, DAT_RST, DAT_RD, DAT_RSP,    -- Datafield
    CHK_A, CHK_RST, CHK_RD, CHK_RSP,    -- Checksum
    REPOS                               -- Repos
);
```

Listing 5 – Declaration des états

Dans une première étape, nous déclarons les différents états de l'automate sous la forme d'un type énuméré nommé `CFM`. Chaque état correspond à une étape spécifique du processus de réception LIN, facilitant ainsi la gestion des transitions et des actions associées à chaque état.

```
-- Register
CFM_Register : process(H, nCLR)
begin
    if nCLR = '0' then
        P_CFM <= REPOS;
    elsif rising_edge(H) then
        P_CFM <= N_CFM;
    end if;
end process CFM_Register;
```

Listing 6 – Registres Reception Trame

Dans ce code nous retrouvons la clock qui permet de synchroniser les états de l'automate avec le signal d'horloge H. Le changement d'état se fait au front montant de l'horloge. Le reset asynchrone nCLR permet de remettre l'automate dans son état initial

```
-- Reseau Combinatoire d'Entree
CFM_RCE : process(P_CFM, H, Identifier, LinSynchro, NbTbit_0,
    NbDataField_0, SelAdr, nCLR, n_0)
begin
    -- Next State <= Present State
    N_CFM <= P_CFM;

    case P_CFM is
        when REPOS => -- Attente
```

```
            if LinSynchro = '0' then
                N_CFM <= R_BRK_0;
            end if;
        when R_BRK_0 => -- Synchro Break 0
            if NbTbit_0 = '1' AND LinSynchro = '1' then
                N_CFM <= R_BRK_1;
            elsif NbTbit_0 = '0' AND LinSynchro = '1' then
                N_CFM <= REPOS;
            elsif n_0 = '0' AND LinSynchro = '0' AND NbTbit_0 = '0' then
                N_CFM <= R_BRK_0;
            elsif NbTbit_0 = '0' AND LinSynchro = '0' AND n_0 = '1' then
                N_CFM <= R_BRK_0;
            end if;
        when R_BRK_1 => -- Synchro Break 1
            if n_0 = '1' AND LinSynchro = '1' then
                N_CFM <= SYN_A;
            elsif n_0 = '0' AND LinSynchro = '0' then
                N_CFM <= REPOS;
            elsif n_0 = '0' then
                N_CFM <= R_BRK_1;
            end if;
        when SYN_A => -- Attente bit Start Synchronisation
            if LinSynchro = '0' then
                N_CFM <= SYN_RST;
            end if;
        when SYN_RST => -- Reception Start Synchronisation
            if n_0 = '0' AND LinSynchro = '0' then
                N_CFM <= SYN_RD;
            elsif n_0 = '0' AND LinSynchro = '1' then
                N_CFM <= REPOS;
            elsif n_0 = '0' then
                N_CFM <= SYN_RST;
            end if;
        when SYN_RD => -- Reception Data Synchronisation
            if NbTbit_0 = '1' then
                N_CFM <= SYN_RSP;
            elsif n_0 = '1' AND NbTbit_0 = '0' then
                N_CFM <= SYN_RD;
            elsif n_0 = '0' then
                N_CFM <= SYN_RD;
            end if;
        when SYN_RSP => -- Reception bit Stop Synchronisation
            if n_0 = '1' AND LinSynchro = '1' then
                N_CFM <= IDN_A;
            elsif n_0 = '0' AND NbTbit_0 = '0' then
                N_CFM <= REPOS;
            elsif n_0 = '0' then
                N_CFM <= SYN_RSP;
            end if;
        when IDN_A => -- Attente bit Start Identifier
            if LinSynchro = '0' then
                N_CFM <= IDN_RST;
            end if;
        when IDN_RST => -- Reception bit Start Identifier
            if n_0 = '1' AND LinSynchro = '0' then
                N_CFM <= IDN_RD;
            elsif n_0 = '1' AND LinSynchro = '1' then
```

```
66                  N_CFM <= REPOS;
67              elsif n_0 = '0' AND LinSynchro = '0' then
68                  N_CFM <= IDN_RST;
69              end if;
70          when IDN_RD => -- Reception Data Identifier
71              if NbTbit_0 = '1' AND n_0 = '1' then
72                  N_CFM <= IDN_RSP;
73              elsif n_0 = '0' then
74                  N_CFM <= IDN_RD;
75              elsif n_0 = '1' AND NbTbit_0 = '0' then
76                  N_CFM <= IDN_RD;
77              end if;
78          when IDN_RSP => -- Reception bit Stop Identifier
79              if n_0 = '1' AND LinSynchro = '1' then
80                  N_CFM <= DAT_A;
81              elsif n_0 = '1' AND LinSynchro = '0' then
82                  N_CFM <= REPOS;
83              elsif n_0 = '0' AND LinSynchro = '1' then
84                  N_CFM <= IDN_RSP;
85              end if;
86          when DAT_A => -- Attente bit Start Datafield
87              if LinSynchro = '0' then
88                  N_CFM <= DAT_RST;
89              end if;
90          when DAT_RST => -- Reception bit Start Datafield
91              if n_0 = '1' AND LinSynchro = '0' then
92                  N_CFM <= DAT_RD;
93              elsif n_0 = '1' AND LinSynchro = '1' then
94                  N_CFM <= REPOS;
95              elsif n_0 = '0' AND LinSynchro = '0' then
96                  N_CFM <= DAT_RST;
97              end if;
98          when DAT_RD => -- Reception bit Data Datafield
99              if NbTbit_0 = '1' AND LinSynchro = '1' then
100                 N_CFM <= DAT_RSP;
101             elsif n_0 = '1' AND NbTbit_0 = '0' then
102                 N_CFM <= DAT_RD;
103             elsif n_0 = '0' then
104                 N_CFM <= DAT_RD;
105             end if;
106         when DAT_RSP => -- Reception bit Stop Datafield
107             if n_0 = '0' AND NbDataField_0 = '1' AND LinSynchro = '1'
                    then
108                 N_CFM <= CHK_A;
109             elsif NbDataField_0 = '0' AND n_0 = '1'AND LinSynchro = '1'
                    then
110                 N_CFM <= DAT_A;
111             elsif n_0 = '1' AND LinSynchro = '0' then
112                 N_CFM <= REPOS;
113             elsif n_0 = '0' AND LinSynchro = '1'then
114                 N_CFM <= DAT_RSP;
115             end if;
116         when CHK_A => -- Attente bit Start Checksum
117             if LinSynchro = '0' then
118                 N_CFM <= CHK_RST;
119             end if;
120         when CHK_RST => -- Reception bit Start Checksum
```

```
            if n_0 = '1' AND LinSynchro = '0' then
                N_CFM <= CHK_RD;
            elsif n_0 = '1' AND LinSynchro = '1' then
                N_CFM <= REPOS;
        elsif n_0 = '0' AND LinSynchro = '0' then
                N_CFM <= CHK_RST;
            end if;
          when CHK_RD => -- Reception bit Data Checksum
            if NbTbit_0 = '1' AND LinSynchro = '1' then
                N_CFM <= CHK_RSP;
            elsif n_0 = '1' AND NbTbit_0 = '0' then
                N_CFM <= CHK_RD;
            elsif n_0 = '0' then
                N_CFM <= CHK_RD;
            end if;
          when CHK_RSP => -- Reception bit Stop Checksum
            if n_0 = '1' AND LinSynchro = '1' AND Identifier = SelAdr
               then
                N_CFM <= REPOS; -- Fin de Trame Complete
            elsif n_0 = '1' AND LinSynchro = '1' AND Identifier /=
               SelAdr then
                N_CFM <= REPOS;
            elsif n_0 = '0' AND LinSynchro = '0' then
                N_CFM <= REPOS;
            elsif n_0 = '0' AND LinSynchro = '1' then
                N_CFM <= CHK_RSP;
            end if;
      end case;

end process CFM_RCE;
```

Listing 7 – Réseau Combinatoire d'Entrée Reception Trame

Ce code VHDL traduit l'automate de réception LIN en utilisant un processus combinatoire nommé `CFM_RCE`. Il détermine l'état suivant (`N_CFM`) en fonction de l'état actuel (`P_CFM`) et des signaux d'entrée tels que `LinSynchro`, `NbTbit_0`, `Identifier`, etc. Chaque état de l'automate est représenté par une branche dans la structure `CASE`, avec des conditions spécifiques pour les transitions entre états.

```
  -- Reseau Combinatoire de Sortie
CFM_RES : process(P_CFM, H, Identifier, LinSynchro, NbTbit_0,
    NbDataField_0, SelAdr, nCLR, n_0 )
begin
  -- Valeurs par defaut
  Error_Start       <= '0';
  Error_Stop        <= '0';
  Error_Synchro     <= '0';
  Identifieur_en    <= '0';
  IncNbOctet        <= '0';
  MessageReceiveSet <= '0';
  NbDataField_EN    <= '0';
  NbDataField_load  <= '0';
  NbOcyeyRecu_RST   <= '0';
  OctetRecu_RST     <= '0';
  OctetRecu_WR      <= '0';
  OctetRecu_en      <= '0';
  n_Tbit_Load       <= '0';
```

```
18    n_Tbit_en          <= '0';
19    n_Tbit_select      <= '0';
20    n_en               <= '0';
21    n_load             <= '0';
22    n_select           <= '0';
23
24    case P_CFM is
25      when REPOS => -- Attente
26        if LinSynchro = '0' then
27          n_select        <= '0';
28          n_load          <= '1';
29          n_en            <= '1';
30          n_Tbit_select   <= '0';
31          n_Tbit_en       <= '1';
32          n_Tbit_Load     <= '1';
33        end if;
34
35      when R_BRK_0 => -- Synchro Break 0
36        if NbTbit_0 = '1' and LinSynchro = '1' then
37          n_select <= '0';
38          n_en     <= '1';
39          n_load   <= '1';
40        elsif NbTbit_0 = '0' and LinSynchro = '1' then
41          Error_Synchro <= '1';
42        elsif NbTbit_0 = '0' and LinSynchro = '0' and n_0 = '1' then
43          n_Tbit_en <= '1';
44          n_select  <= '0';
45          n_load    <= '1';
46          n_en      <= '1';
47        elsif n_0 = '0' AND LinSynchro = '0' AND NbTbit_0 = '0' then
48           n_en <= '1';
49        end if;
50
51      when R_BRK_1 => -- Synchro Break 1
52        if n_0 = '1' and LinSynchro = '0' then
53          -- Nothing
54        elsif n_0 = '0' and LinSynchro = '0' then
55          Error_Synchro <= '1';
56        elsif n_0 = '0' then
57          n_en <= '1';
58        end if;
59
60      when SYN_A => -- Attente bit Start Synchronisation
61          if LinSynchro = '0' then
62              n_select  <= '0';
63              n_load    <= '1';
64              n_en      <= '1';
65          end if;
66
67      when SYN_RST => -- Reception Start Synchronisation
68          if n_0 = '0' AND LinSynchro = '0' then
69              n_select       <= '1'; -- passage de fin de front a milieu
                    bit
70              n_load         <= '1';
71              n_en           <= '1';
72              n_Tbit_select  <= '1';
73              n_Tbit_en      <= '1';
```

```
74              n_Tbit_Load     <= '1';
75          elsif n_0 = '0' AND LinSynchro = '1' then
76              Error_Start     <= '1';
77          elsif n_0 = '0' then
78              n_en            <= '1';
79          end if;
80
81      when SYN_RD => -- Reception Data Synchronisation
82          if NbTbit_0 = '1' then
83              n_select        <= '0';
84              n_load          <= '1';
85              n_en            <= '1';
86          elsif n_0 = '1' AND NbTbit_0 = '0' then
87              n_select        <= '0';
88              n_load          <= '1';
89              n_en            <= '1';
90              n_Tbit_en       <= '1';
91              OctetRecu_en    <= '1';
92          elsif n_0 = '0' then
93              n_en            <= '1';
94          end if;
95
96      when SYN_RSP => -- Reception bit Stop Synchronisation
97          if n_0 = '1' AND LinSynchro = '1' then
98              OctetRecu_WR    <= '1';
99              n_select        <= '1';
100             n_load          <= '1';
101             n_en            <= '1';
102         elsif n_0 = '0' AND NbTbit_0 = '0' then
103             Error_Stop      <= '1';
104         elsif n_0 = '0' then
105             n_en            <= '1';
106         end if;
107
108     when IDN_A => -- Attente bit Start Identifier
109         if LinSynchro = '0' then
110             n_select        <= '0';
111             n_load          <= '1';
112             n_en            <= '1';
113         end if;
114
115     when IDN_RST => -- Reception bit Start Identifier
116         if n_0 = '1' AND LinSynchro = '0' then
117             n_select        <= '0';
118             n_load          <= '1';
119             n_en            <= '1';
120             n_Tbit_select   <= '1';
121             n_Tbit_Load     <= '1';
122             n_Tbit_en       <= '1';
123         elsif n_0 = '1' AND LinSynchro = '1' then
124             Error_Start     <= '1';
125         elsif n_0 = '0' AND LinSynchro = '0' then
126             n_en            <= '1';
127         end if;
128
129     when IDN_RD => -- Reception Data Identifier
130         if NbTbit_0 = '1' AND n_0 = '1' then
```

```
131             n_select        <= '0';
132             n_load          <= '1';
133             n_en            <= '1';
134       OctetRecu_en    <= '1';
135         elsif n_0 = '0' then
136             n_en            <= '1';
137         elsif n_0 = '1' AND NbTbit_0 = '0' then
138       n_select        <= '0';
139       n_load          <= '1';
140             n_en            <= '1';
141       n_Tbit_en       <= '1';
142             OctetRecu_en    <= '1';
143         end if;
144
145     when IDN_RSP => -- Reception bit Stop Identifier
146         if n_0 = '1' AND LinSynchro = '1' then
147             OctetRecu_en    <= '1';
148             NbDataField_EN  <= '1';
149             Identifieur_en  <= '1';
150             NbDataField_load <= '1';
151         elsif n_0 = '0' AND LinSynchro = '0' then
152             Error_Stop      <= '1';
153         elsif n_0 = '0' AND LinSynchro = '1' then
154             n_select        <= '0';
155             n_load          <= '0';
156             n_en            <= '1';
157         end if;
158
159     when DAT_A => -- Attente bit Start Datafield
160         if LinSynchro = '0' then
161             n_select        <= '0';
162             n_load          <= '1';
163             n_en            <= '1';
164         end if;
165
166     when DAT_RST => -- Reception bit Start Datafield
167         if n_0 = '1' AND LinSynchro = '0' then
168             n_select        <= '0';
169             n_load          <= '1';
170             n_en            <= '1';
171             n_Tbit_select   <= '1';
172             n_Tbit_Load     <= '1';
173             n_Tbit_en       <= '1';
174         elsif n_0 = '1' AND LinSynchro = '1' then
175             Error_Start     <= '1';
176         elsif n_0 = '0' AND LinSynchro = '0' then
177             n_en            <= '1';
178         end if;
179
180     when DAT_RD => -- Reception bit Data Datafield
181         if NbTbit_0 = '1' AND LinSynchro = '1' then
182             n_select        <= '0';
183             n_load          <= '1';
184             n_en            <= '1';
185             OctetRecu_en    <= '1';
186         elsif n_0 = '0' then
187             n_en            <= '1';
```

```
188         elsif n_0 = '1' AND NbTbit_0 = '0' then
189             n_select        <= '0';
190             n_load          <= '1';
191             n_en            <= '1';
192             n_Tbit_en       <= '1';
193             OctetRecu_en    <= '1';
194         end if;
195
196     when DAT_RSP => -- Reception bit Stop Datafield
197         if n_0 = '0' AND NbDataField_0 = '1' AND LinSynchro = '1' then
198              OctetRecu_WR    <= '1';
199              n_select        <= '0';
200              n_load          <= '1';
201              n_en            <= '1';
202              IncNbOctet      <= '1';
203           elsif NbDataField_0 = '0' AND n_0 = '1'AND LinSynchro = '1'
                  then
204              OctetRecu_WR    <= '1';
205              NbDataField_EN  <= '1';
206              n_select        <= '0';
207              n_load          <= '1';
208              n_en            <= '1';
209              IncNbOctet      <= '1';
210           elsif n_0 = '1' AND LinSynchro = '0' then
211             Error_Stop      <= '1';
212         elsif n_0 = '0' AND LinSynchro = '1'then
213              n_en            <= '1';
214         end if;
215
216     when CHK_A => -- Attente bit Start Checksum
217         if LinSynchro = '0' then
218             n_select        <= '0';
219             n_load          <= '1';
220             n_en            <= '1';
221         end if;
222
223     when CHK_RST => -- Reception bit Start Checksum
224         if n_0 = '1' AND LinSynchro = '0' then
225                 n_select        <= '0';
226                 n_load          <= '1';
227                 n_en            <= '1';
228                 n_Tbit_select   <= '1';
229         elsif n_0 = '1' AND LinSynchro = '1' then
230                 Error_Start     <= '1';
231         elsif n_0 = '0' AND LinSynchro = '0' then
232                 n_en            <= '1';
233       end if;
234
235     when CHK_RD => -- Reception bit Data Checksum
236         if NbTbit_0 = '1' AND LinSynchro = '1' then
237                 n_select        <= '0';
238                 n_load          <= '1';
239                 n_en            <= '1';
240         elsif n_0 = '1' AND NbTbit_0 = '0' then
241                 n_en            <= '1';
242         elsif n_0 = '0' then
243                 n_Tbit_en       <= '1';
```

```
244                 n_select        <= '0';
245                 n_load          <= '1';
246                 n_en            <= '1';
247                 OctetRecu_en    <= '1';
248                 Identifieur_en     <= '1';
249                 NbDataField_load   <= '1';
250                 NbDataField_EN     <= '1';
251       end if;
252
253     when CHK_RSP => -- Reception bit Stop Checksum
254         if n_0 = '0' AND LinSynchro = '1' AND Identifier = SelAdr then
255             MessageReceiveSet <= '1';
256             NbOcyeyRecu_RST   <= '1';
257         elsif n_0 = '0' AND LinSynchro = '1' AND Identifier /= SelAdr
                then
258             MessageReceiveSet <= '1';
259             NbOcyeyRecu_RST   <= '1';
260         elsif n_0 = '0' AND LinSynchro = '0' then
261             Error_Stop      <= '1';
262         elsif n_0 = '0' then
263             n_en            <= '1';
264         end if;
265
266     when others =>
267       -- Sec
268       null;
269   end case;
270 end process CFM_RES;
```

Listing 8 – Réseau Combinatoire de Sortie Reception Trame

Ce code VHDL implémente le réseau combinatoire de sortie (`CFM_RES`) pour l'automate de réception LIN. Il met à jour les signaux de sortie tels que `Error_Start`, `Error_Stop`, `OctetRecu_WR`, etc., en fonction de l'état actuel (`P_CFM`) et des signaux d'entrée. Chaque état de l'automate est géré dans une structure `CASE`, avec des conditions spécifiques pour définir les actions à entreprendre dans chaque état. Des valeurs par défaut sont également définies au début du processus pour éviter des comportements indésirlés.

Une fois ces processus implémentés, il nous reste qu'a les assemblés pour ne formé qu'une machine Complete.

Le schéma complet de l'interface de réception LIN est présenté ci-dessus, illustrant l'intégration des différentes parties opératives et commandes (Répresentant la machine séquentielle).

7.3 FIFO

La mémoire FIFO (First In, First Out) a été conçue pour stocker temporairement les données reçues via l'interface LIN avant leur traitement ultérieur. Elle est structurée autour d'un composant clef, la FIFO. Cette section à été réaliser sur un schéma blocs, en reprenant les concepts vus en cours.

7.4 Etat Interne

Etat interne est un composant VHDL qui gère les états internes du système de réception LIN. Le systeme à été conçu pour suivre et contrôler les différentes étapes du processus de réception des trames LIN. L'Objectifs étant d'additionner des fonctionnalités de contrôle et de gestion des états internes pour ne formé qu'une seul trame.

Ce systeme à lui aussi été conçu sous la forme d'un schéma bloc, reprenant les concepts vus en cours. Avec des bascules D, des compteurs et l'ajout de OR pour gérer les RESET.

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*.

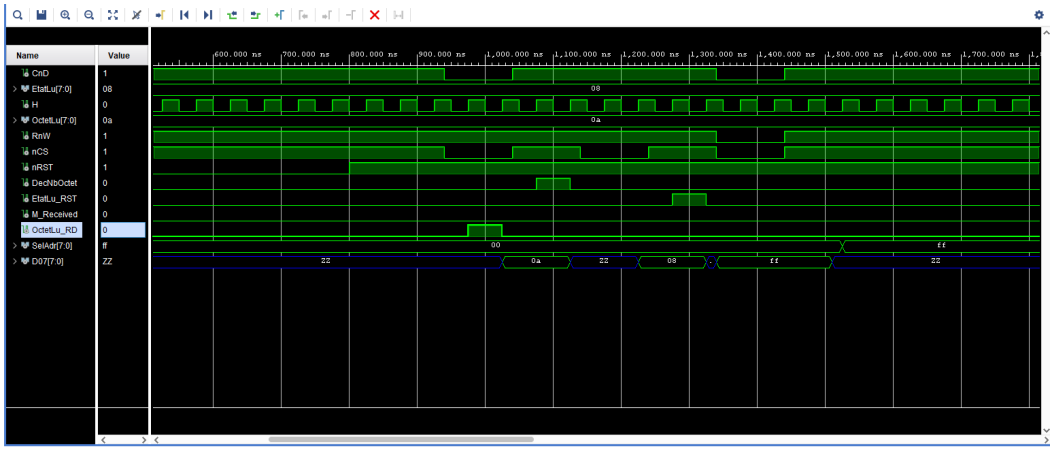


FIGURE 22 – Chronogramme de simulation de l'Interface Microprocesseur

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).

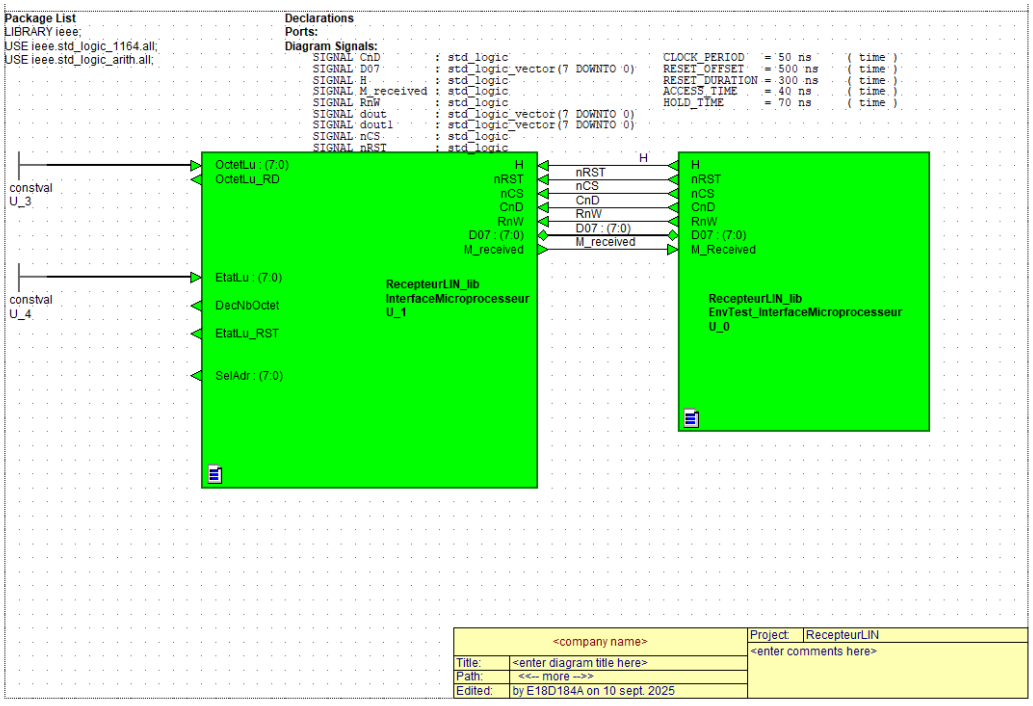


FIGURE 23 – Block Diagramme de test de l'Interface Microprocesseur

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` ← 10 (codé sur 8 bits),
— `EtatLu` ← 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.

8.2 Interface Reception LIN

8.2.1 Déclarations et signaux

8.2.2 Instanciation du composant testé

8.2.3 Environnement de test

8.2.4 Stimuli supplémentaires

8.2.5 Analyse du chronogramme de simulation

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.

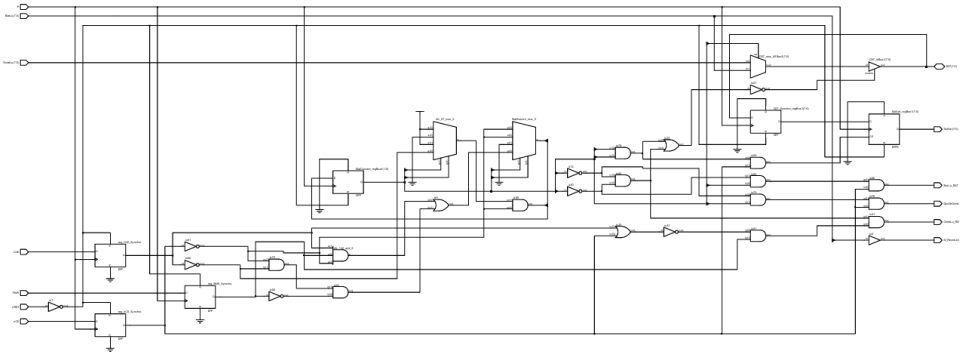


FIGURE 24 – Schéma RTL InterfaceMicroprocesseur

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.

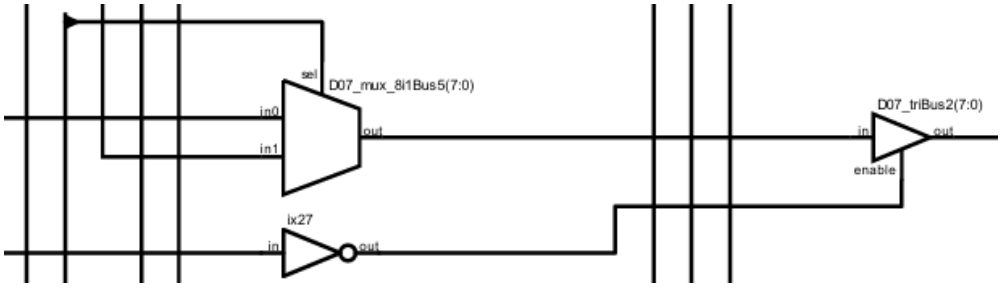


FIGURE 25 – Partie opérative avec multiplexeur et Tristate : InterfaceMicroprocesseur

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 la synthèse matériel qui transforme la logique en ressource matérielle.

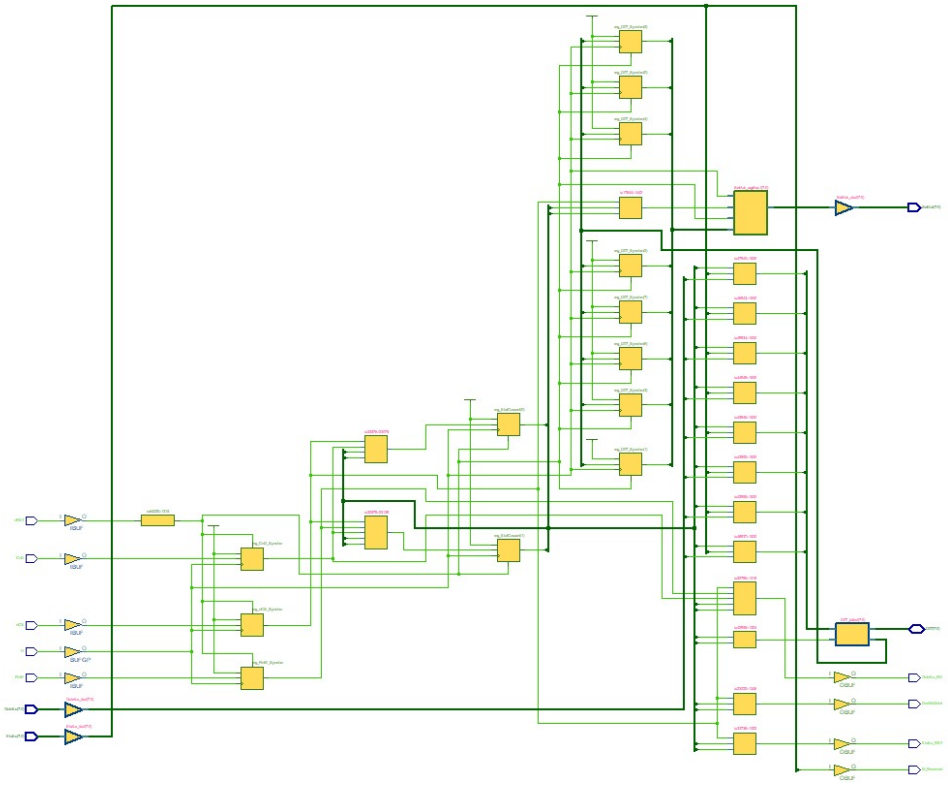


FIGURE 26 – Synthese matérielle InterfaceMicroprocesseur

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.

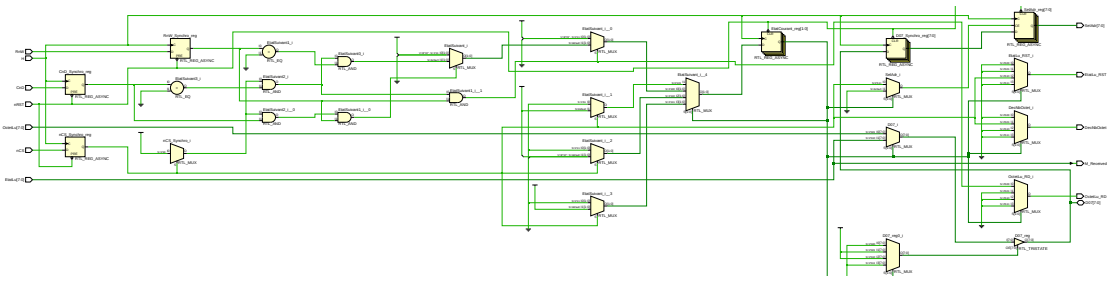
Grâce à cette conversion, le design passe d'une **représentation abstraite de la logique** à une **implémentation matérielle concrète**, optimisée pour le FPGA cible. Cela permet non seulement de visualiser l'organisation physique des composants.

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.

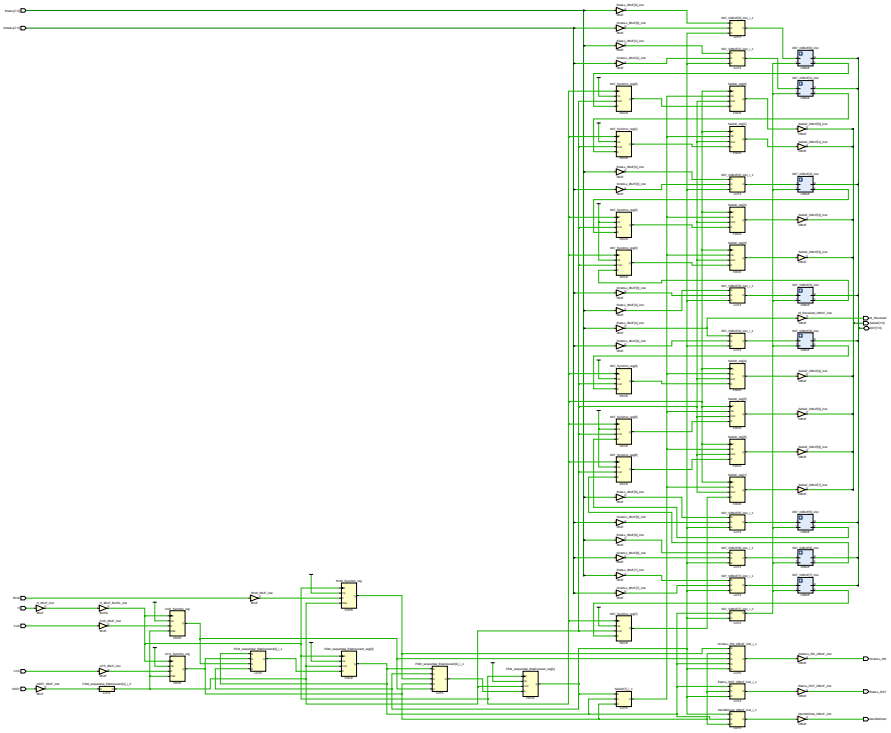
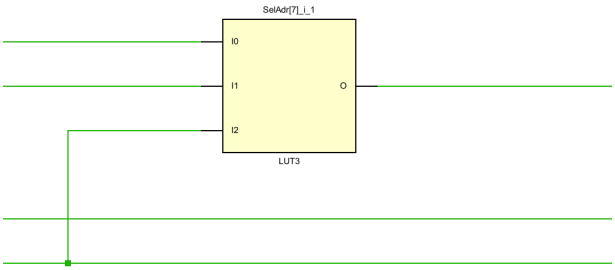


FIGURE 27 – Synthèse matérielle Vivado

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

Cell Properties

SelAdr[7]_i_1

I2	I1	I0	O=I0 & I1 & I2
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

(b) Table de vérité associée

FIGURE 28 – 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.

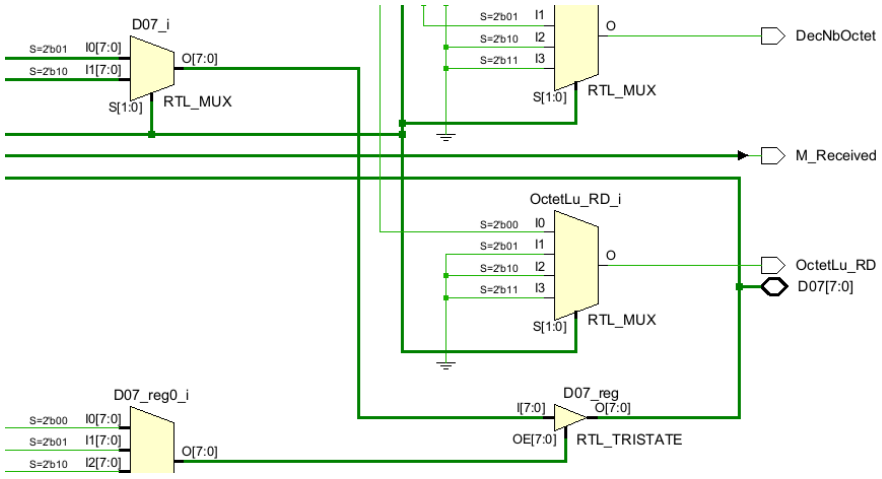


FIGURE 29 – Synthèse Logique Vivado

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 :

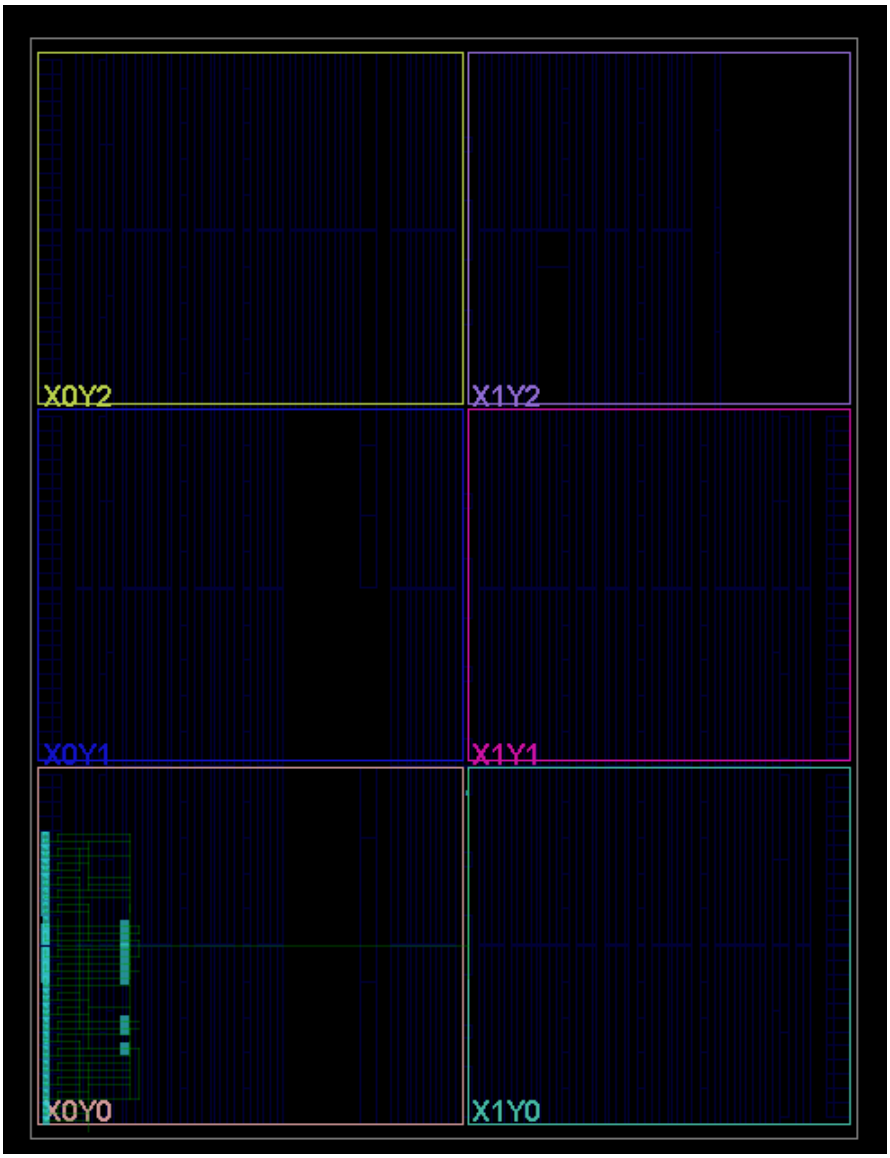
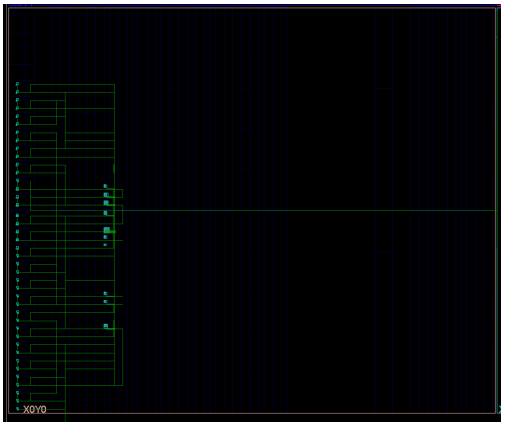
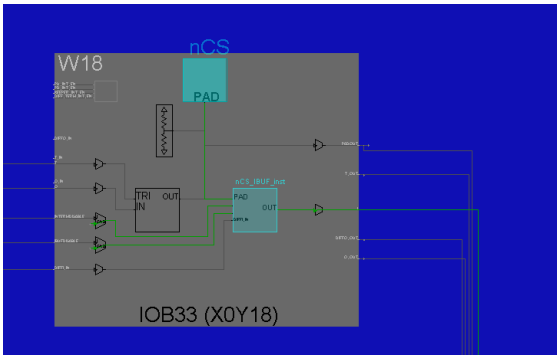


FIGURE 30 – Slice du FPGA après routage

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**.



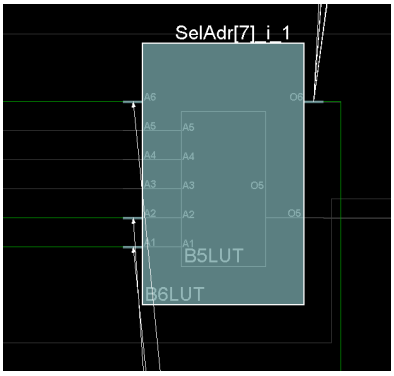
(a) Localisation du système dans la zone 0 du FPGA



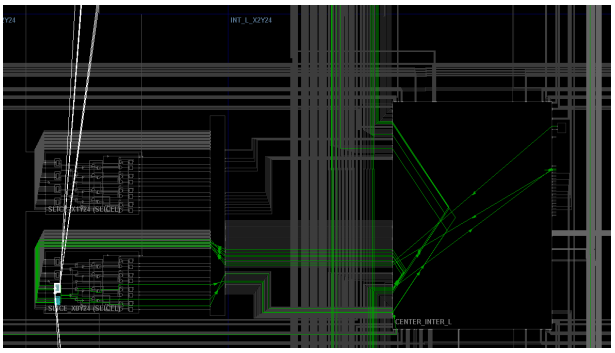
(b) Buffer associé à nCS

FIGURE 31 – 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 :



(a) Connexion interne d'une LUT3



(b) Schéma détaillé du câblage LUT3

FIGURE 32 – 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

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY EnvTest_InterfaceMicroprocesseur IS
   GENERIC(
      CLOCK_PERIOD    : time := 50 ns;
      RESET_OFFSET    : time := 500 ns;
      RESET_DURATION : time := 300 ns;
      ACCESS_TIME     : time := 40 ns;
      HOLD_TIME       : time := 70 ns
   );
   PORT(
      M_Received : IN     std_logic;
      CnD        : OUT    std_logic;
      H          : OUT    std_logic;
      RnW        : OUT    std_logic;
      nCS        : OUT    std_logic;
      nRST       : OUT    std_logic;
      D07        : INOUT  std_logic_vector (7 DOWNTO 0)
   );
-- Declarations

END EnvTest_InterfaceMicroprocesseur ;

--
ARCHITECTURE arch OF EnvTest_InterfaceMicroprocesseur IS
TYPE DefState IS (Waiting, DataReading, StateReading, FilterWriting);

SIGNAL ProcessorState : DefState;

BEGIN

ClockGeneratorProc : PROCESS
BEGIN
  H <= '0';
  WAIT FOR CLOCK_PERIOD/2;
  H <= '1';
  WAIT FOR CLOCK_PERIOD/2;
END PROCESS ClockGeneratorProc;

ResetGeneratorProc : PROCESS
BEGIN
  nRST <= '1';
  WAIT FOR RESET_OFFSET;
  nRST <= '0';
  WAIT FOR RESET_DURATION;
  nRST <= '1';
  WAIT;
END PROCESS ResetGeneratorProc;

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 9 – Testbench InterfaceMicroprocesseur

Table des figures