# Compte Rendu FPGA1_2----Centrale DCC sur FPGA

**XU Weiqin 3410681**
**HUANG Xingyun 3602284**

## Présentation du projet :

Le but de ce projet est d'envoyer les commandes à un modélisme ferroviaire.

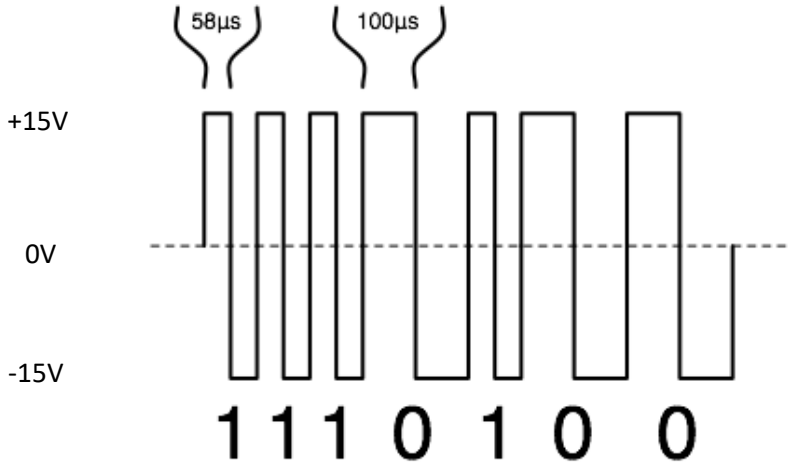
La réalisation : On implémente une **Centrale DCC** dans le **FPGA** de la carte **Nexys4-DDR**. Après avoir envoyé les consignes via les boutons de la carte **Nexys4-DDR**, des commandes vont être amplifié par une carte **Booster** (amplificateur de signal) afin que le signal soit suffisamment puissant d'être envoyé sur les rails puis d'être décodé par les locomatives.

## I) Protocole DCC

Le protocol **DCC**(Digital Command Control) est un standard défini par le **NMRA** (National Model Railroad Association), qui permet le contrôl individuel des locomotives ou des accessoires en modulant la tension d'alimentation de la voie.

### 1. Le bit :

La tension de la voie est un signal continu bipolaire qui donne une forme de courant alternatif. Les inversions de tensions sont instantanées ce qui donne un signal pulsé. La durée de l'impulsion dans chaque sens fournit le codage du signal :Pour définir un bit de '1', la durée est courte (58µs pour un demi-cycle) alors qu'un bit '0' est représenté par une période longue (100µs pour un demi-cycle).



## 2. La trame :

Les locomotives et leurs accessoires (feux, effets sonores des engins moteurs) ainsi que les accessoires du réseau (aiguillages, itinéraires) possèdent chacun une adresse unique et décodent les instructions qui leur sont envoyées, et seules les instructions des paquets avec l'adresse correspondante seront activées par le décodeur. Le signal codé envoyé sur la voie donne des ordres aux équipements tout en fournissant la puissance.

Les informations envoyées consistent en une séquence de bits (un paquet / une trame ) qui est utilisé pour coder l'un d'un ensemble d'instructions que le décodeur numérique fonctionnera sur. Les paquets doivent être précisément définis pour assurer le bon encodage et le décodage des instructions.

Le paquet de contrôle de commande consiste en trois octets de données précédés d'un préambule et d'un bit de démarrage, comme indiqué dans le tableau ci-dessous:

| Composition de la Trame Détail | Détail |
|---|---|
| Préambule | Suite de 14 bits à 1 |
| Start Bit | 1 bit à 0 |
| Octet d'adresse | Adresse de la locomotive à commander |
| Start Bit | 1 bit à 0 |
| Octet de donnée | Commande envoyée au train |
| Start Bit | 1 bit à 0 |
| Octet de contrôle | XOR entre les deux octets précédents, pour détecter d'éventuelles erreurs de transmission |
| Stop Bit | 1 bit à 1 |

## 3. La commande (une séquence de 4 trames):

Pour donner les consignes (8 bits), on a distribué 3 bits pour indiquer l'adresse, 2 bits pour indiquer la VITESSE en proposant 4 choixs et 2 bits pour indiquer la FONCTION en représentant 4 foncitions différentes :

```
go <= out_go;
consigne(7) <= '0';
consigne(6 downto 4) <= adresse_t;
consigne(3 downto 2) <= vitesse_t;
consigne(1 downto 0) <= fonction_t;
```

La commande d'un train s'opère en envoyant périodiquement une séquence de quatre trames : IDLE - VITESSE- FONCTION - IDLE . Chaque trame est séparée par un temps de 400 µs, pendant lequel la sortie est maintenue à 0.

L'octet de donnée de la trame de VITESSE est de la forme suivante : 01DXXXXX. Le bit D fixe la direction du train : le bit 0 pour une marche arrière et le bit 1 pour une marche avant. Les 5 bits de poids faible indiquent la vitesse de la locomotive. On a choisi 4 mode de vitesse, comme indiqué dans le tableau ci-dessous:

| Consigne(3 downto 2) | L'octet de donnée de VITESSE | Vitesse choisie (Speed) |
|---|---|---|
| 00 | 01100000 | Stop (Arrêt) |
| 01 | 01110110 | Speed= Step 10 (lente/avant) |
| 10 | 01111010 | Speed= Step 20 (vite/avant) |
| 11 | 01010110 | Speed= Step 10 (lente/arrière) |

```
if consigne(3 downto 2) = "00" then
      byte <= "01100000";
elsif consigne(3 downto 2) = "01" then
      byte <= "01110110";
elsif consigne(3 downto 2) = "10" then
      byte <= "01111011";
elsif consigne(3 downto 2) = "11" then
      byte <= "01010110";
end if;
```

L'octet de donnée de la trame de FONCTION permet d'activer ou de désactiver certaines fonctionnalités des trains, selon le tableau ci-dessous. Chaque fonction est activée en mettant el bit correspondant à 1, et est désactivée en mettant ce même bit à 0. Comme dans le tableau ci-dessous, on a distribué 2 bits pour représenter 4 fonctions :

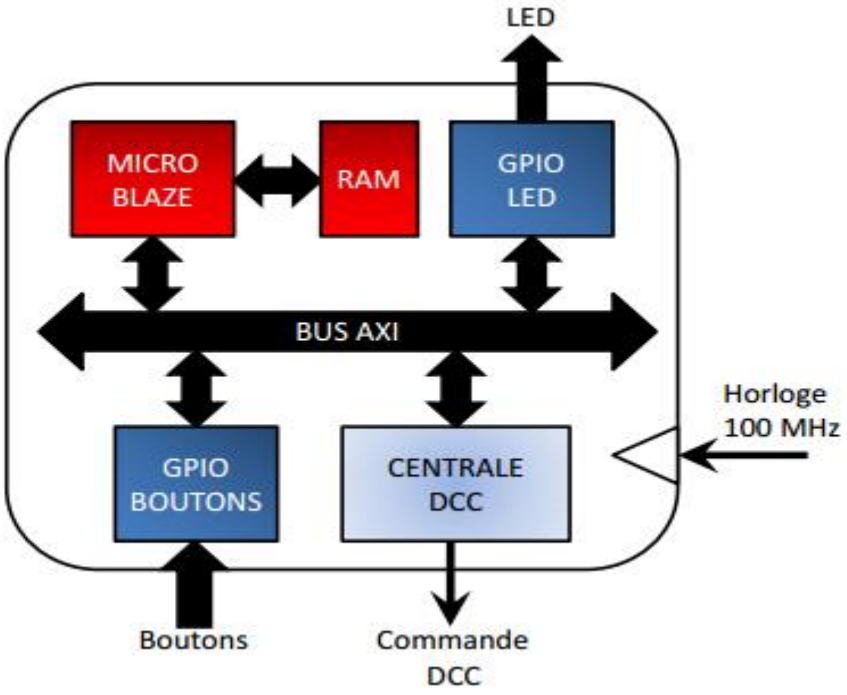
| Consigne(1 downto 0) | L'octet de donnée de FONCTION | Fonction |
|---|---|---|
| 00 | 10010111 | Allumage des phares/ Klaxon n°1 en continu/bruit de moteur |
| 01 | 10100011 | Sifflet en continu/bruit de moteur |
| 10 | 10101000 | Klaxon n°2 bref |
| 11 | 10100100 | Klaxon n°1 bref |

```
else
      if consigne(1 downto 0) = "00" then
            byte <= "10010111";
      elsif consigne(1 downto 0) = "01" then
            byte <= "10100011";
      elsif consigne(1 downto 0) = "10" then
            byte <= "10101000";
      elsif consigne(1 downto 0) = "11" then
            byte <= "10100100";
      end if;
end if;
```

Pour conclure, la composition de trame est la suivante :

| Numéro de Trame | Type de Trame | Composition |
|---|---|---|
| 1 | IDLE | - Octet d'adresse : 11111111<br>- Octet de donnée : 00000000<br>- Octet de contrôle : 11111111 |
| | 400 µs | |
| 2 | VITESSE | - Octet d'adresse : Adresse locomotive (de 1 à 6)<br>- Octet de donnée : 01100000/01110110/01111010/ 01010110<br>- Octet de contrôle : XOR entre les deux octets précédents |
| | 400 µs | |
| 3 | FONCTION | - Octet d'adresse : Adresse locomotive (de 1 à 6)<br>- Octet de donnée : 10010111/10100011/10101000/ 10100100<br>- Octet de contrôle : XOR entre les deux octets précédents |
| | 400 µs | |
| 4 | IDLE | - Octet d'adresse : 11111111<br>- Octet de donnée : 00000000<br>- Octet de contrôle : 11111111 |

## II) Architecture de la centrale DCC




Architecture de la centrale DCC

Pour contrôler le système, on envoie les commandes en manipulant les **Boutons**, un **Microblaze** va analyser les consignes et les délivrer à **Centrale DCC.** Et le **LED** va demontrer les consignes visuellement.

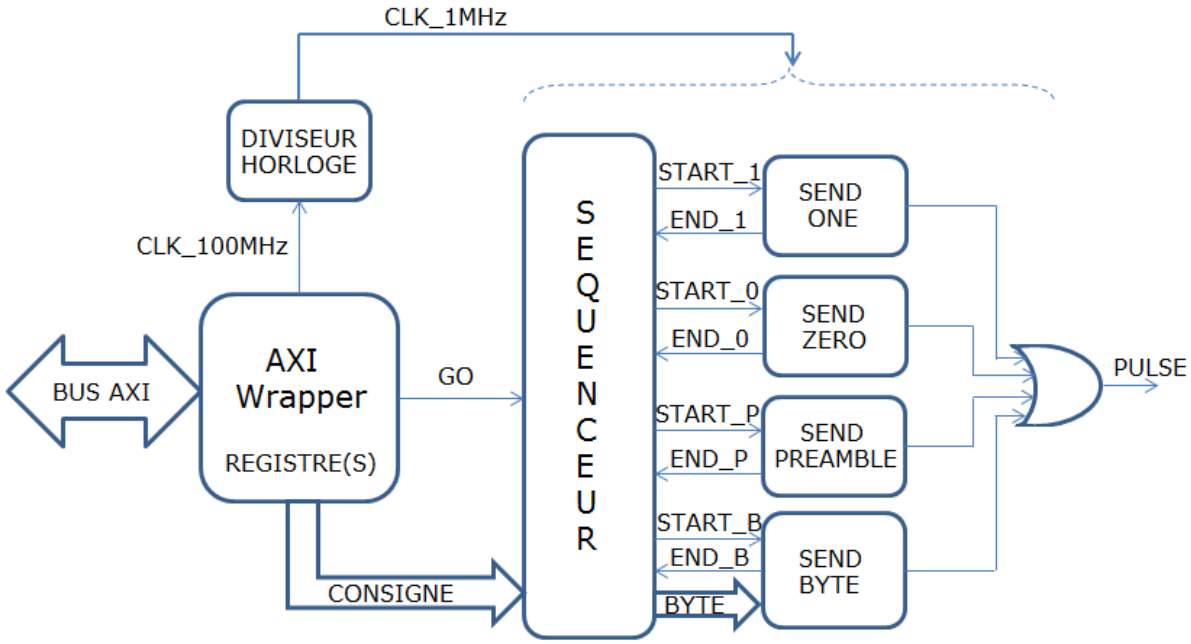
Détail du fonctionnement des boutons poussoirs

| Bouton | Rôle | Fonctionnement |
|---|---|---|
| **BTNC** | Démarrage | Activer GO |
| **BTNL / BTNR** | Choix de options<br>(BTNL : -1/ BTNR : +1) | - FONCTION : 00<br>- VITESSE : 01<br>- ADRESSE : 10 |
| **BTNU / BTND** | Choix de valeurs<br>(BTND : -1/ BTNU : +1) | - FONCTION : 2 bits- 4 fonctions<br>- VITESSE : 2 bits – 4 vitesses<br>- ADRESSE : 3 bits – 2 adresse (one hot) |

Détail du fonctionnement des LED

| LED | Rôle | Fonctionnement |
|---|---|---|
| **LED(15)** | DCC ON /OFF | Activer GO |
| **LED(14 :12)** | Mode<br>FONCTION/VITESSE/ADRESSE | - ADRESSE : 10 --------LED(14)<br>- VITESSE : 01 --------LED(13)<br>- FONCTION : 00 --------LED(12) |
| **LED(6 :0)** | Valeur<br>FONCTION/VITESSE/ADRESSE | - FONCTION : 2 bits- LED(1:0)<br>- VITESSE : 2 bits – LED(3 :2)<br>- ADRESSE : 3 bits – LED(6 :4) |

## III) Architecture de l'IP Centrale DCC/ Description et validation de l'IP Centrale DCC en VHDL



L'IP **Centrale DCC** est composée comme dans le déssin ci-dessus. Dans le premier temps, on essaie de décrire chaque blocs en VHDL, et pour vérifier le bon fonctionnement , on a écrit un module **INTERFACE** qui produit **GO** et **CONSIGNE** . Pour développer une version purement matérielle de la **Centrale DCC**, on remplace l'**INTERFACE** with **AXI Wrapper** pour que on puisse donner les consignes via les boutons sur la carte **Nexys4 DDR**.

**DIVISEUR D'HORLOGE :**

Pour gérer les timings propres au signal de commande DCC, le diviseur d'horloge sert à transmettre l'horloge de 100MHz de Microblaze à une horloge de fréquence 1 MHz pour cadencer le système.

```
entity diviseur is
    Port ( clk_100MHz : in STD_LOGIC;
           clk_1MHz : out STD_LOGIC);
end diviseur;

architecture Behavioral of diviseur is
signal compteur: std_logic_vector(5 downto 0):="000000";
signal output: std_logic:='0';
begin
    clk_1MHz <= output;
    process(clk_100MHz)
    begin
        if rising_edge(clk_100MHz) then
            compteur <= compteur + 1;
            if compteur = "110001" then
                output <= not output;
                compteur <= "000000";
            end if;
        end if;
    end process;

end Behavioral;
```

**SEND ONE et SEND ZERO :**

Ce sont les deux modules qui réalisent l'envoi d'un signal DCC correspondant à un 1 logique(SEND ONE) représenté par une impulsion à 0 de 58 µs puis implusion à 1 de 58 µs, ou un 0 logique(SEND ZERO) représenté par une impulsion à 0 de 100 µs puis implusion à 1 de 100 µs.

```
process(clk_1MHz,reset)
begin
    if reset = '0' then
        compteur <= "00000000";
        end_1_inter <= '0';
        op <= '0';
    elsif rising_edge(clk_1MHz) and (start_1 = '1' or end_1_inter = '1') then
        if compteur = "00111001" then
            op <= '1';
        elsif compteur = "01110010" then
            end_1_inter <= '1';
        elsif compteur = "01110011" then
            end_1_inter <= '0';
            op <= '0';
        end if;
        if compteur = "01110011" then
            compteur <= "00000000";
        else
            compteur <= compteur + 1;
        end if;
    end if;
end process;
```

send_one

```
process(clk_1MHz,reset)
begin
    if reset = '0' then
            compteur <= "00000000";
            end_0_inter <= '0';
            op <= '0';
    elsif rising_edge(clk_1MHz) and (start_0 = '1' or end_0_inter = '1') then
        if compteur = "01100011" then
            op <= '1';
        elsif compteur = "11000110" then
            end_0_inter <= '1';
        elsif compteur = "11000111" then
            end_0_inter <= '0';
            op <= '0';
        end if;
        if compteur = "11000111" then
            compteur <= "00000000";
        else
            compteur <= compteur + 1;
        end if;
    end if;
end process;
```

send_zero

**SEND PREAMBLE:**

**SEND PREAMBLE** consiste en 14 bits à 1, donc ce module est composé d'un sous module **SEND ONE** et un **COMPTEUR.**

**Remarque :** Pour orchestrer les deux sous-modules, avant le front ascendant de l'horloge, il faut que **start_p** devient 1, après le front ascendant de l'horloge **start_1** change à 1 et la valeur est transmit à **SEND ONE** .

```
architecture Behavioral of send_preamble is
signal compteur: std_logic_vector(3 downto 0):="0000";
signal end_send1,output_send1,start_1: std_logic;
begin
    send1: entity work.send_one(Behavioral)
           port map(clk_1MHz,start_1,reset,end_send1,output_send1);
    start_1 <= start_p;
    process(end_send1,reset)
    begin
        if reset = '0' then
            compteur <= "0000";
        elsif falling_edge(end_send1) then
            if compteur = "1101" then
                compteur <= "0000";
            elsif start_p = '1' then
                compteur <= compteur + 1;
            end if;
        end if;
    end process;

    end_p <= '1' when end_send1 = '1' and compteur = "1101" else
             '0';
    output <= output_send1;
end Behavioral;
```

send_preamble

**SEND OCTET:**

Ce module est composé de 2 sous blocs et d'une porte **OU** .

Un sous module **SEND ONE**, un sous module **SEND ZERO** identique à celui précédemment décrit.

```
begin
    send1: entity work.send_one(Behavioral)
           port map(clk_1MHz,start_1,Reset,end_send1,output_send1);

    send0: entity work.send_zero(Behavioral)
           port map(clk_1MHz,start_0,Reset,end_send0,output_send0);

    end_ensem <= end_send1 or end_send0;
```

send_octet

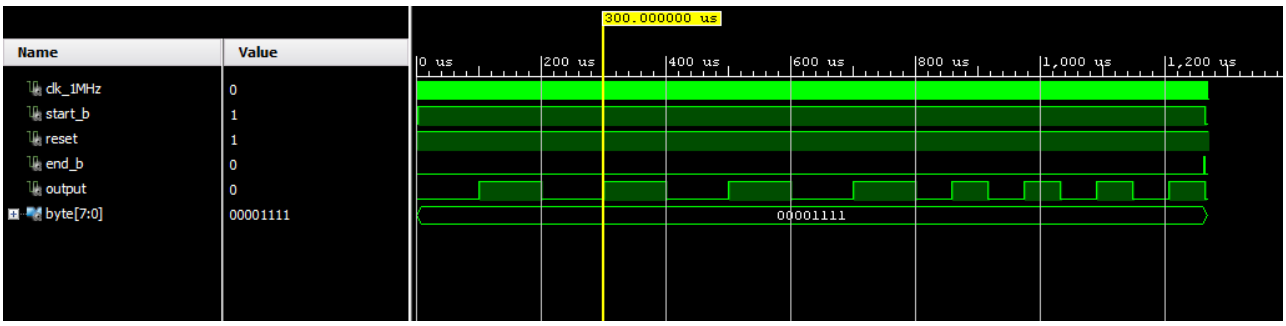
Et une porte **OU** qui prend en entrée la sortie des deux sous-modules précédents pour constituer la sortie du module.

```
start_1 <= start_b when ((compteur = "000" and byte(7) = '1') or
                         (compteur = "001" and byte(6) = '1') or
                         (compteur = "010" and byte(5) = '1') or
                         (compteur = "011" and byte(4) = '1') or
                         (compteur = "100" and byte(3) = '1') or
                         (compteur = "101" and byte(2) = '1') or
                         (compteur = "110" and byte(1) = '1') or
                         (compteur = "111" and byte(0) = '1')) else
           '0';

start_0 <= start_b when ((compteur = "000" and byte(7) = '0') or
                         (compteur = "001" and byte(6) = '0') or
                         (compteur = "010" and byte(5) = '0') or
                         (compteur = "011" and byte(4) = '0') or
                         (compteur = "100" and byte(3) = '0') or
                         (compteur = "101" and byte(2) = '0') or
                         (compteur = "110" and byte(1) = '0') or
                         (compteur = "111" and byte(0) = '0')) else
           '0';


end_b <= '1' when end_ensem = '1' and compteur = "111" else
         '0';
output <= output_send1 when start_1 = '1' else
          output_send0 when start_0 = '1' else
          '0';
```

o Simuler puis faire une synthèse pour vérifier le bon fonctionnement.




Simulation post-synthèsis de send_octet

Remarque : Dans la simulation on a fournie une octet 00001111, et le output change en correspondant le output_send1 et output_send0 comme on veut. end_b va passer à 1 pour indiquer la fin d'un octet.

**PULSE :**

- Les 4 sorties des modules **SEND ONE , SEND ZERO, SEND OCTET, SEND PREMBLE** sont combinées dans un **OU** pour constituer le signal DCC de sortie.

```
PULSE <= out_one or out_zero or out_pre or out_octet;
```

Et le signal **PULSE** sera connecté à la broche 4 du connecteur **PMOD A**, cela correspond à la broche **G17** de la carte Nexys4 DDR. (on modifie dans le fichier constraint)

```
set_property -dict { PACKAGE_PIN G17   IOSTANDARD LVCMOS33 } [get_ports
{ PULSE }]; #IO_L18N_T2_A23_15 Sch=ja[4]
```

**AXI WRAPPER / INTERFACE:**

C'est un module qui réalise l'interface avec le bus AXI pour relier l'IP au Microblaze. Il contient un ou plusieurs registres de configurations et il faut piloter l'IP corrélativement.

```
    -- Add user logic here
L0: entity work.top_DCC port map (S_AXI_ACLK,go,S_AXI_ARESETN,consigne,PULSE);
go <= slv_reg1(0);
consigne <= slv_reg0(7 downto 0);
    -- User logic ends
```

Il produit 2 signaux au séquenceur :

**GO :**

- Signal périodique de période 53 ms environ, et de rapport cyclique 50%.

```
process(reset,clk)

        begin

                if reset = '0' then
                      count_go<=(others => '0');
                      out_go <= '0';
                elsif rising_edge(Clk) and active_go = '1'
then
                   count_go <= count_go + 1;
                   if count_go = "10100001101111100010000" then
                       out_go <= not out_go;
                       count_go<=(others => '0');
                   end if;
                end if;

        end process;
```

- Sert à renvoyer périodiquement la séquence de 4 trames vers les rails. (à voir dans interface.vhd)
- Ce signal est généré uniquement si l'IP est en mode envoi de la commande est contrôlé par Bouton Central. (active_go)

```
            if cpt_btnc = "11111010000000" then
                if BTNC = '1' then
                    active_go <= not active_go;
                end if;
                start_c <= '0';
            end if;
```

- Pour stabiliser, on a introduit **lock** pour chaque **bouton** avec un compteur de 160 µs,voici un exemple :

```
process(clk)
begin
    if rising_edge(clk) then
        if BTNC = '1' and lock_c = '0' then
            start_c <= '1';
            cpt_btnc <= (others => '0');
        else
            if start_c = '1' then
                cpt_btnc <= cpt_btnc + 1;
            end if;
            if cpt_btnc = "11111010000000" then
                if BTNC = '1' then
                    active_go <= not active_go;
                end if;
                start_c <= '0';
            end if;
        end if;
        lock_c <= BTNC;
    end if;
end process;
```

**CONSIGNE :**

-Contient l'adresse du train visé ainsi que la vitesse et la fonction que l'on souhaite lui donner .

```
consigne(7) <= '0';
consigne(6 downto 4) <= adresse_t;
consigne(3 downto 2) <= vitesse_t;
consigne(1 downto 0) <= fonction_t;
```

**SEQUENCEUR :**

Ce module commande les 4 modules de génération de données conformément au protocole DCC.

- Les commandes **START_X** activent les modules correspondants.
- Les entrées **END_X** indiquent au séquenceur que l'opération de génération demandée est terminée.

Selon la composition de la trame, ce module est décrit sous la forme d'une machine à états :

| | | | |
|---|---|---|---|
| **Etat 0** | IDLE | **Etat 5** | Octet de donnée |
| **Etat 1** | Préambule | **Etat 6** | Start Bit (SEND ZERO) |
| **Etat 2** | Start Bit (SEND ZERO) | **Etat 7** | Octet de contrôle |
| **Etat 3** | Octet d'adresse | **Etat 8** | Stop Bit (SEND ONE) |
| **Etat 4** | Start Bit (SEND ZERO) | **Etat 9** | 400 µs pour séparer deux trames |

**Précision 1 :**
**Séquenceur** est activé par **GO**, il faut faire attention à la durée du signal **GO**, parce que si GO est ecore  activé quand on a fini envoyer 1 commande (4 trame), on va continuer à envoyer les autres 4 trames identiques.
Pour cela, on a introduit **go_inter**,  qui indique une impulsion de **GO.** Pour produire **go_inter**, on a utilisé **pre_go** pour représenter la valeur précédente de **GO**.

```
process(clk_1MHz)
begin
    if rising_edge(clk_1MHz) then
        if go = '1' and pre_go = '0' then
            go_inter <= '1';
        else
            go_inter <= '0';
        end if;
        pre_go <= go;
    end if;
end process;
```

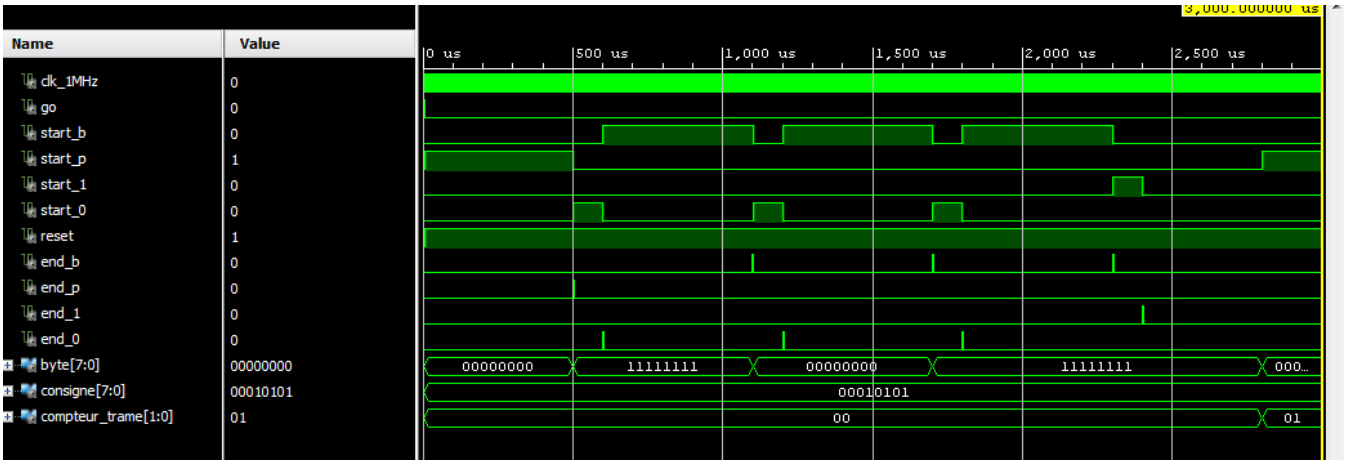
Et puis dans l'état de **IDLE** , on démarre la machine à états par **go_inter** :

```
case ep is
when etat0 =>                                   --IDLE
    if go_inter = '1' then
        ef <= etat1;
    else
        ef <= etat0;
```

**Précision 2 :**
Dans la machine à état, les valeurs obtenues dans chaque état ne transmettent pas aux autres états. Tandis que pour les données de contrôle, il faut faire XOR entre les 2 octets précédents. Donc on a réécrit les partie d'octet d'adresse et d'octet de commande dans l'état 6 et l'état 7. Sinon on faillit à envoyer les commandes correctement.

Simuler puis faire une synthèse pour vérifier le bon fonctionnement :
On a donné la désigne comme ci-dessous :

```
    clk_1MHz <= not clk_1MHz after 0.5 us;
    reset <= '0' after 2 us,'1' after 4 us;
    go <= '1' after 7 us, '0' after 8 us;
    consigne <= "00010101";
    end_p <= '1' after 500 us,'0' after 501 us;
    end_0 <= '1' after 600 us,'0' after 601 us, '1' after 1200 us, '0' after
1201 us, '1' after 1800 us, '0' after 1801 us;
    end_b <= '1' after 1100 us,'0' after 1101 us,'1' after 1700 us,'0' after
1701 us,'1' after 2300 us,'0' after 2301 us;
    end_1 <= '1' after 2400 us, '0' after 2401 us;
```

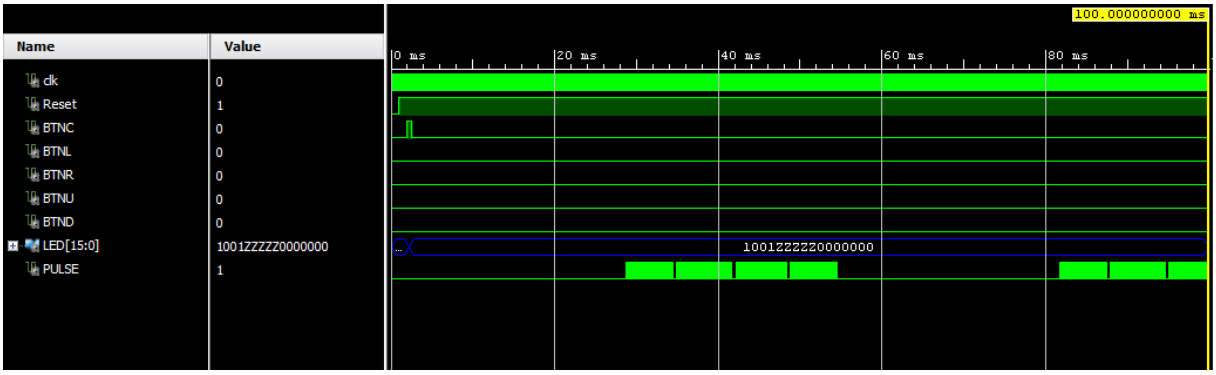



Simulation post syntèsis de séquenceur

Remarque : On peut voir dans la simulation que les **START_X**s correspondent avec les **END_X**s donnés, et après end_1 du stop bit, la trame commencera après 400 µs.

**TOP :**

On a écrit **top.vhd** pour décrire l'ensemble de **Centrale DCC**, ensuite on a fait la simulation et a vérifié le foncionnement :




Simulation post syntèsis de top

Remarque : Après avoir modifié l'utilisation de **GO** , on peut voir dans la simulation qu'il y a plus de situation de la répétition de commande. Et on a réussi à envoyer une commande complète qui contient tous les 4 trames.

## Travaux réalisés :

Et puis on a implémenté le code sur le FPGA de la carte Nexys4 DRR, et on a appuyé les boutons correspondants pour tester de différents fonctions avec différent vitesse sur les 3 trains. Et puis les LEDs affiche les valeurs d'adresse, de vitesse et de fonctions, aussi les trains roulent comme on a désigné. Donc on a réussi à envoyer les commandes au modélisme ferroviaire.

# IV) Intégration de l'IP Centrale DCC dans un système Microblaze

Dans cette partie on va intégrer **l'IP Centrale DCC** dans un système mixte métériel/logiciel organisé autour d'un **Microblaze**. Il faut donc remplacer l'**interface** de la partie précédente par un **wrapper AXI** afin que la **Centrale DCC** puisse échanger des informations avec le processeur.

Le **wrapper AXI** est créé automatiquement par **Vivado**, qui ajoute un certain nombre de registres de configuration (**slv_reg**). Ces registres sertent à faire le lien entre le code C exécuté par le **Microblaze** et la **CONSIGNE** qui donne les trames à envoyer aux trains.

## 1) Spécification et génération de la plate-forme matérielle

**Créer une plate-forme Microblaze sous Vivado :**

- Dans la partie **IP Integrator** du **Flow Navigator**, cliquer sur **Create Block Design**. Une fenêtre **Diagram** se crée à droite de l'écran.

- Pour instancier un **Microblaze**, cliquer sur **Add IP** dans le bandeau vert en haut de la fenêtre **Diagram**, taper les premières lettres de **Microblaze** et sélectionner **Microblaze** dans la liste filtrée des **IP** disponibles.

- Un module **Microblaze** est ajouté au **Diagram**. Cliquer sur **Run Block Automation** puis **/microblaze_0** pour configurer le processeur. Modifier les paramètres suivants :
a) Local Memory : Passer de 8 à 32KB.
b)Interrupt Controller : Cocher cette case.

-Cliquer sur OK. Vivado ajoute alors automatiquement d'autres modules au **Diagram**.

**Créer une IP Centrale DCC:**

- Aller dans le menu **Tools**, sélectionner **Create and Package IP**. Une fenêtre s'ouvre : cliquer sur Next puis dans la fenêtre suivante choisir **Create a new AXI4 peripheral**.
Dans la fenêtre suivante, donner un nom (**centrale_DCC2**) et éventuellement une description à votre IP puis Next .
Dans la fenêtre suivante, choisir comme interface bus AXI4-Lite puis cliquer sur Next .
Dans la fenêtre suivante, vérifier les paramètres sont bien ceux qu'on veut et cliquer sur Next.
Dans la fenêtre suivante, terminer en cliquant sur **Edit IP** puis Finish.

- Une deuxième fenêtre **Vivado** s'ouvre alors sous la forme d'un projet vous permettant de créer ou modifier la fonctionnalité de votre **IP**.
Aller dans le menu **File** et cliquer sur **New File** et créer un fichier **top_DCC.vhd**, dans ce fichier décrire l'architecture de l'IP **Centrale DCC**.

- Ouvrir d'abord le fichier **central_DCC2_v1_0.vhd.**
Dans l'entité, après la ligne -- Users to add ports here, ajouter : ***PULSE: out std_logic***;
Chercher la ligne – component declaration. Dans la liste des ports, ajouter avant **S_AXI_ACLK** la ligne suivante : ***PULSE: out std_logic***;

Chercher la ligne port map( et ajouter dans la liste des instanciations la ligne suivante : ***PULSE =>PULSE***.

- Ouvrir ensuite le fichier **centrale_DCC2_v1_0_S00_AXI.vhd**.
Dans l'entité, après la ligne -- Users to add ports here, ajouter : ***PULSE: out std_logic***;
Juste avant le begin de l'architecture, ajouter la ligne suivante : ***signal go: std_logic*** et ***signal consigne: std_logic_vector(7 down to 0);***
Chercher la ligne -- Add user logic here et ajouter la ligner suivante : **L0: entity work.top_DCC port map (S_AXI_ACLK ,go, S_AXI_ARESETN ,consigne,PULSE)**;

- Après sauvegarde des fichiers, l'entité led est à présent rattachée au module **centrale_DCC2_v1_0_S00_AXI.**

- Dans le fichier **centrale_DCC2_v1_0_S00_AXI.vhd**, chercher la déclaration des signaux **slv_reg0,1,2,3** , ces 4 signaux correspondent aux 4 registres de configuration de votre IP.Nous allons fixer pour l'IP le comportement suivant : le bit 0 du slv_reg1 est à **go**, le bit 7 à 0 du slv_reg0 est à **consigne**.

- Après sauvegarde des fichiers, cliquer sur l'onglet **Package IP**.

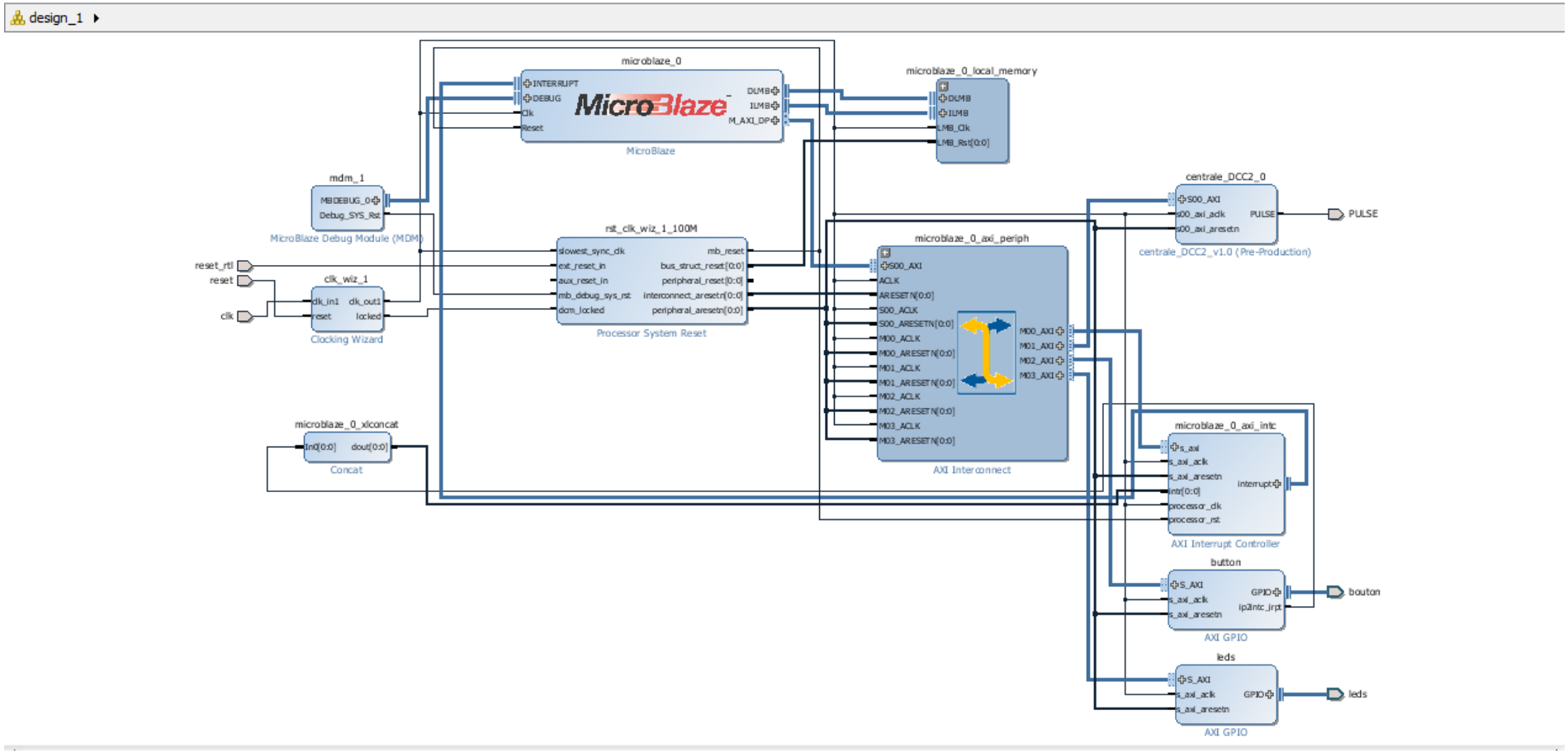
- Cliquer sur **IP File Groups** puis sur **Merge Changes from IP File Groups Wizard** dans le bandeau jaune. Faire de même pour les autres IP Packaging Steps jusqu'à ce qu'il n'y ait plus que des verts devant chaque Packaging Step. Cliquer alors sur **Review and Package** puis sur **Repackage IP**. Vivado ferme alors la fenêtre du projet de l'IP.

**<u>Ajouter l'IP Centrale DCC au système :</u>**

- Cliquer avec le bouton droit sur le **Diagram** et choisir **Add IP**. Chercher puis ajouter l'**IP centrale_DCC2**. Cliquer sur **Run Connection Automation**. Vérifier que le paramètre Auto est sélectionné puis valider. Cliquer sur le **Diagram** avec le bouton droit et choisir **Create Port**. Ajouter les ports suivants : **PULSE / Reset / clk**. Connecter manuellement ce port aux sorties corrélatives.

-Cliquer sur **Run Block Automation** puis **/centrale_DCC2_0** pour configurer le processeur. Cliquer sur OK. **Vivado** ajoute alors automatiquement d'autres modules au **Diagram**.

- Cliquer sur **Validate Design** pour vérifier qu'il n'y a pas d'erreurs dans le **Diagram**.




**Référence :**

https://fr.wikipedia.org/wiki/Digital_Command_Control

https://www.picotech.com/library/oscilloscopes/digital-command-control-dcc-protocol-decoding