

## SYSTEMES PROGRAMMABLES— CENTRALE DCC SUR FPGA

SYSTEMES PROGRAMMABLES  
— CENTRALE DCC SUR FPGA

Encadrant :J.Denoulet

Sorbonne Université  
M1 SESI  
FPGA1QIN Guanting  
28605121

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Synthèse VHDL de Centrale DCC</b>	<b>3</b>
2.1	Architecture de la Centrale DCC . . . . .	4
2.1.1	Generateur Trames Registre DCC . . . . .	4
2.1.2	Registre DCC . . . . .	5
2.1.3	DCC bit 0/ DCC bit 1 . . . . .	6
2.1.4	MAE . . . . .	8
2.1.5	TOP . . . . .	10
2.2	Génération du Bitstream et Démonstration . . . . .	11
2.2.1	Modification de Contraintes . . . . .	11
2.2.2	Post-Synthesis/Implementation Functional Simulation . . . . .	11
2.2.3	Génération du Bitstream et Résultat . . . . .	11
<b>3</b>	<b>Conception d'IP pour le Microblaze</b>	<b>13</b>
3.1	Création d'IP Centrale DCC . . . . .	13
3.2	Modification de Contraintes . . . . .	14
3.3	Conception d'Overlay . . . . .	15
<b>4</b>	<b>Amélioration et Conclusion</b>	<b>17</b>

# Chapitre 1

## Introduction

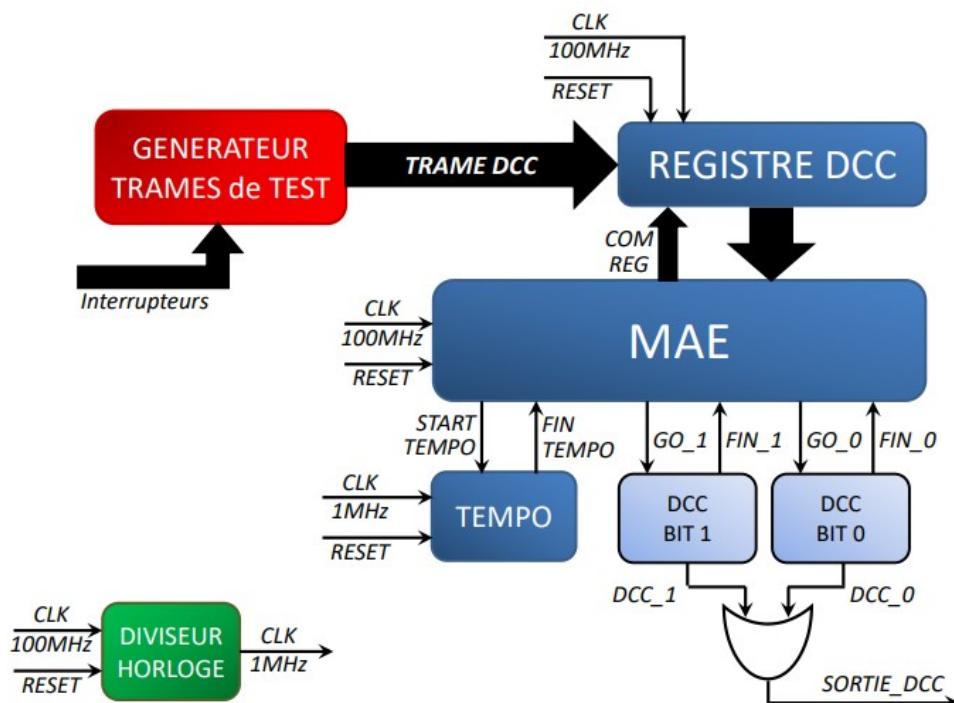
Dans ce projet, nous allons utiliser le logiciel Vivado et Vitis comme environnement pour réaliser un module de contrôle des trains sur la voie. L'objectif du projet est d'utiliser le Nexys A7-100T, que nous avons utilisé dans les TPs précédents, pour contrôler le train en contrôlant les boutons et les interrupteurs afin de transmettre les signaux carrés contenant les commandes aux rails du train, permettant ainsi au train d'avancer, de reculer, de changer de vitesse, de faire retentir le klaxon et d'allumer les lumières.

Ce projet se compose de deux parties. La première partie consiste à utiliser l'environnement Vivado et à écrire des programmes VHDL pour implémenter les fonctions des différents modules. De plus, afin de vérifier la fonctionnalité de chaque module, nous devons écrire un Testbench pour chaque module afin de simuler son comportement. La deuxième partie du projet consiste à encapsuler l'ensemble du code du module Centrale DCC dans une IP destinée à être connectée au Microblaze. Dans l'environnement Vitis, nous ensuite écrivons l'Overlay en Language C, en remplaçant le module Générateur Trames.

# Chapitre 2

## Synthèse VHDL de Centrale DCC

Dans cette section, nous allons mettre en œuvre l'architecture de la centrale DCC (schéma ci-dessous). Cette section est composée de sept sous-parties : Diviseur Horloge et Tempo (les deux sont fournis par l'enseignant), DCC\_Bit\_1 / DCC\_Bit\_0, Registre DCC, MAE (Machine à Etats) et Générateur Trames de Test. La mise en œuvre de chaque partie est décrite dans les sections suivantes.



Architecture de la Centrale DCC

## 2.1 Architecture de la Centrale DCC

### 2.1.1 Générateur Trames Registre DCC

Dans cette section, nous implémentons l'écriture de trames prédéfinis correspondant à chaque fonction du train à utiliser pour se lancer sur les voies.

Par exemple, si nous voulons utiliser le train numéro 2, nous définirons l'adresse du train comme suit : adr = 0x02. Ici, nous voulons que l'interrupteur 7 corresponde à la fonction du train numéro 2 avancer à pleine vitesse, le champ commande doit être 0b01111111 et nous avons le code suivant :

```
-- Interrupteur 7 Activé
--> Trame Marche Avant du Train d'Adresse i
if Interrupteur(7)='1' then

    Trame_DCC <= "111111111111111111111111"
        & '0'
        & x"02"
        & '0'
        & "01111111"
        & '0'
        & (x"02" xor "01111111")
        & '1' ;

-- Préambule
-- Start Bit
-- Champ Adresse
-- Start Bit
-- Champ Commande
-- Start Bit
-- Champ Contrôle
-- Stop Bit
```

Nous avons terminé les 7 fonctions restantes :Allumage / Extinction des Phares, Activation / Réamorçage du Klaxon , Annonce SNCF, et l'arrêt par défaut du train numéro 2.

Dans Testbench, nous réglons chacun des huit états de l'ensemble d'interrupteurs pour observer la sortie du module. Par vérification, nous avons constaté que chaque valeur hexadécimale de sortie correspond à notre valeur binaire de 51 bits prédéfinie et la vérification a été réussie.

```
begin
    Generateur_Trame : entity work.DCC_FRAME_GENERATOR
    port map(Interrupteur=>Interrupteur,Trame_DCC=>Trame_DCC);

    Interrupteur <= "00000001" after 10 ms,
                "00000010" after 20 ms,
                "00000100" after 30 ms,
                "00001000" after 40 ms,
                "00010000" after 50 ms,
                "00100000" after 60 ms,
                "01000000" after 70 ms,
                "10000000" after 80 ms,
                "00000000" after 90 ms;

end Behavioral;
```

Name	Value
> #!Interrupteur[7:0]	00
> #!Trame_DCC[50:0]	7fffff01180c5 7fffff01180c5 7fffe026f001b9 7fffe026f005b5 7fffff0128145 7fffff012914d 7fffff0120105 7fffff0124125 7fffff0117ccb 7fffff011fcfe5

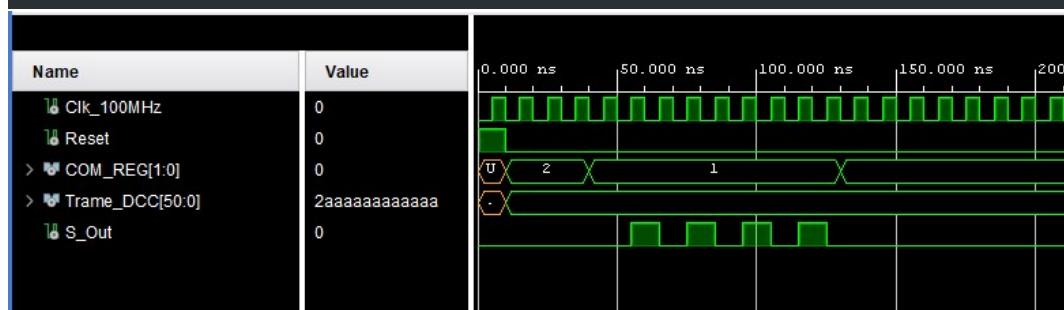
### 2.1.2 Registre DCC

Dans ce module, nous voulons implémenter un registre à décalage. Ce registre à décalage est utilisé pour charger la trame DCC préparée dans le générateur de trames de test, puis le décaler lors de la transmission de la trame afin de faciliter la sortie de cette trame de test de 51 bits un par un.

Il est important de noter que, tout d'abord, la taille du registre correspond à la taille de la plus longue trame du protocole DCC, soit 51 bits. Deuxièmement, les bits de contrôle du registre sont utilisés comme valeurs d'entrée pour ce module et sont fournis par le MAE. Ici, nous définissons cette variable, COM\_REG, comme étant une valeur binaire de deux bits correspondant aux fonctions, respectivement, "0b10" : charger la trame, "0b01" : envoyer la trame, "0b00" : arrêter la transmission de la trame.

Notre idée de la vérification est de supposer d'abord qu'une trame relativement facile à vérifier doit être envoyée, par exemple "010101010 ....." puis nous utilisons l'instruction *assert*, et lorsque nous exécutons le résultat pour obtenir un cadre avec la même valeur que celle attendue, nous n'envoyons aucune information. Si nous constatons que la trame de sortie est différente de ce que nous voulons, nous affichons le message suivant : *Erreur S\_Out sur Trame\_DCC[Position\_de\_bit] - expected : "Valeur attendue" returned : "Valeur réelle"*.

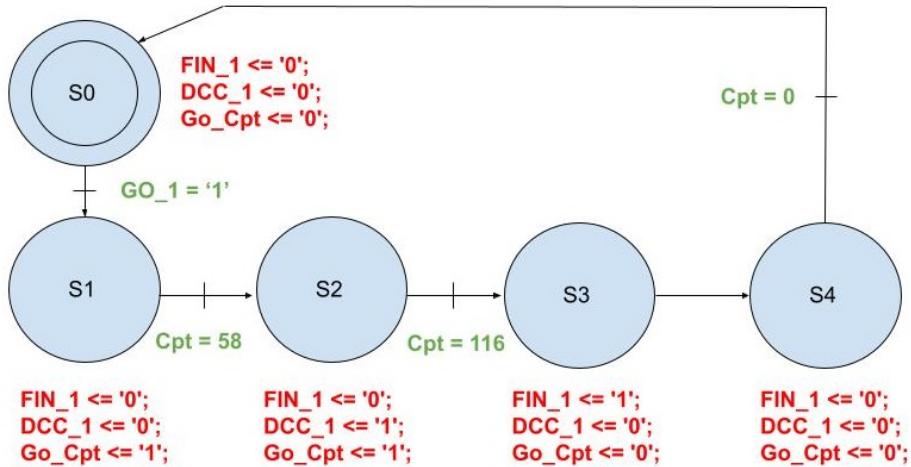
Enfin, nous avons vérifié avec succès le module. Après le Reset est à 1, nous chargeons d'abord la trame, puis nous commençons à l'envoyer. Lorsque nous lui commandons d'arrêter l'envoi, le module commence à décaler la sortie une par une. La sortie est exactement ce que nous avons défini au début : "010101010.....". La vérification est terminée.



### 2.1.3 DCC bit 0/ DCC bit 1

Dans cette section, notre objectif est de générer respectivement un bit à 1 ou à 0 au format DCC. Nous devons écrire une machine d'état simple pour nous aider à générer des signaux.

Dans ce module, nous avons trois processus à écrire, un pour détecter le front montant de l'horloge CLK\_100MHz afin de gérer l'initialisation de la machine d'état, un pour définir la fonction du compteur, et un autre pour la MAE elle-même. Comme les machines d'état des deux modules DCC bit 0/ DCC bit 1 sont différentes, sauf pour la valeur zéro du compteur (116 pour le bit 1 et 200 pour le bit 0), nous ne montrons ici que la machine d'état du DCC bit 1. Nous avons conçu cinq états. Le schéma de la machine d'état est ci-dessous :

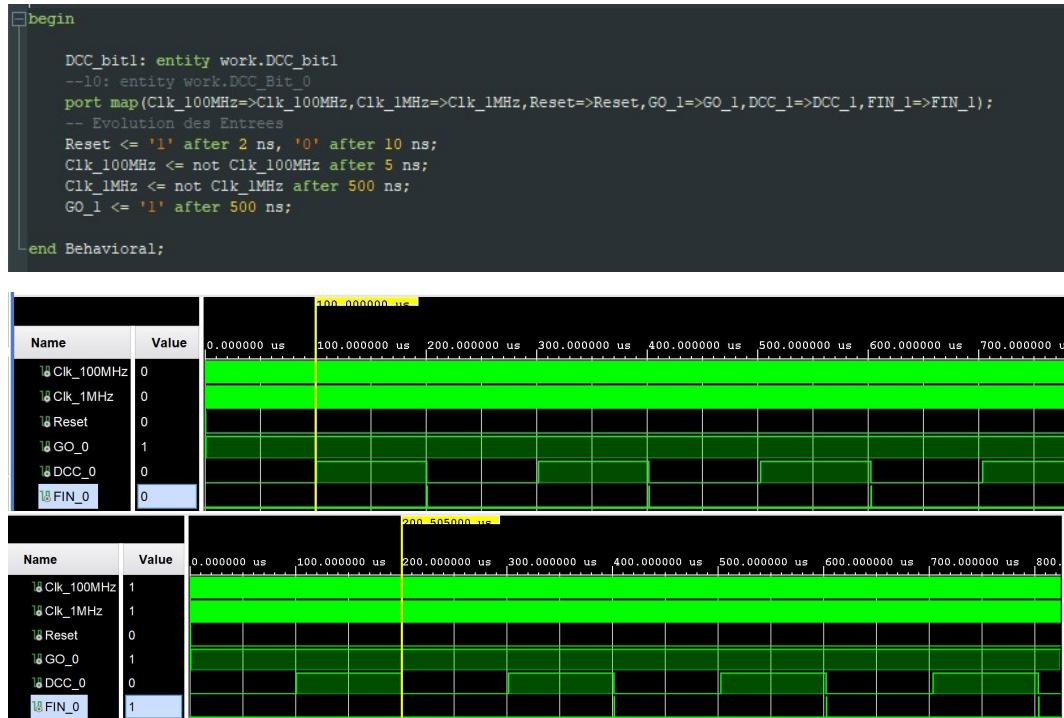


Machine d'état de DCC bit 1

1. L'état initial S0, où toutes les sorties (FIN\_1, DCC\_1) sont égales à 0. Le signal Go\_Cpt utilisé pour déclencher le compteur est également égal à 0.
2. Quand le signal d'entrée Go\_1 = 1, l'état présent passe de S0 à S1. Le compteur est déclenché (Go\_Cpt = 1). FIN\_1 et DCC\_1 restent égaux à 0.
3. Lorsque le compte du compteur atteint 58, S1 atteint l'état S2. À cet état, Go\_Cpt est encore 1 car une période du bit 1 n'a pas encore été achevée. DCC\_1 est égal à 1, c'est-à-dire que le signal de sortie DCC représentant le bit 1 est élevé. FIN\_1 est égal à 0.
4. Lorsque le compteur atteint 116, S2 atteint l'état S3. Le compteur s'arrête et passe à 0 car un cycle du bit 1 a été effectué. DCC\_1 est en état bas et égal à 0. FIN\_1 est égal à 1.

- Une fois le comptage terminé, S3 saute inconditionnellement à S4. Toutes les sorties sont à 0. Lorsque le compteur est initialisé, c'est-à-dire Cpt = 0, la machine d'états revient à S0.

Dans le testbench, il suffit de définir les périodes de deux cycles d'horloge, d'initialiser Reset et de mettre à 1 la condition d'entrée de S1 pour voir le fonctionnement de DCC\_1. Nous pouvons observer que DCC\_1 reste en état haut pendant 58us avec une période de 116us. DCC\_0 reste en état haut pendant 100us avec une période de 200us. La vérification est terminée.



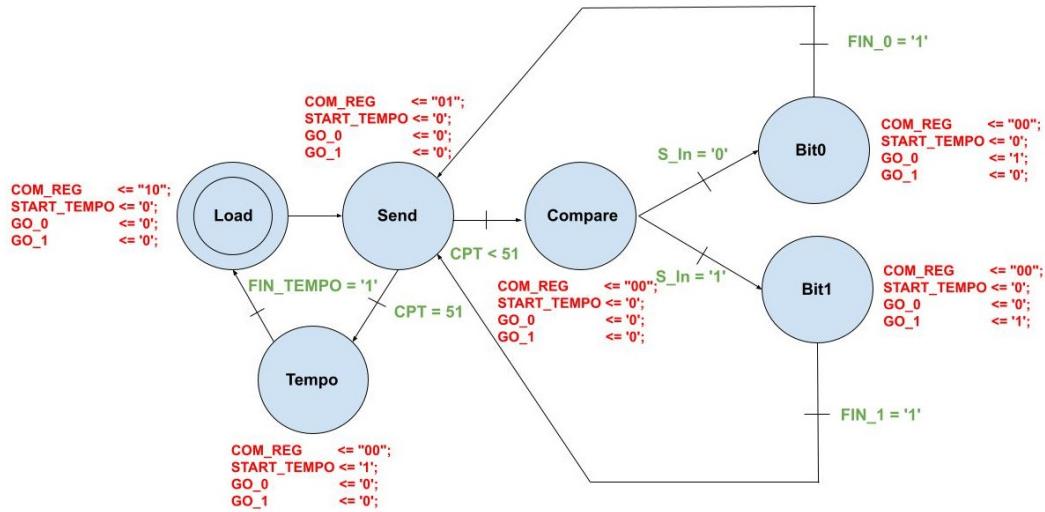
État haut de DCC bit 0 : 100us



État haut de DCC bit 1 : 58us

## 2.1.4 MAE

Dans cette section, nous allons créer une machine d'état pour la Centrale DCC. Elle est chargée d'envoyer des commandes aux quatre modules : TEMPO, REGISTRE DCC, DCC Bit 1 et DCC Bit 0. Elle permet aussi de générer sans interruption la trame de commande DCC préparée par l'utilisateur sur le signal Sortie\_DCC. Le schéma de la machine d'état est ci-dessous :



MAE de Centrale DCC

En outre, dans un autre processus, il est également nécessaire de définir un compteur (codes VHDL ci-dessous) qui compte le nombre de bits reçus par S\_In. Lorsque la machine d'état est à l'état de Load, le compteur est réinitialisé à zéro. Lorsque la machine d'états est dans l'état Send, le compteur commence à incrémenter et compter.

```

begin
  -- Configuration du Compteur-----

  process (Clk_100MHz,RESET)
  begin
    if RESET='1' then CPT <= 0;
    elsif rising_edge(Clk_100MHz) then
      if EP = Send then CPT <= CPT + 1;
      elsif EP = Load then CPT <= 0;
      else CPT <= CPT;
      end if;
    end if;
  end process;
  
```

Nous avons conçu six états, Load, Send, Tempo, Compare, Bit0 et Bit1 :

1. L'état initial de la machine d'état est Load, ce qui correspond au chargement de trame du Registre DCC. Donc à ce moment, COM\_REG est égal à 0b10. Les autres sorties : START\_TEMPO, GO\_0, GO\_1, sont toutes égales à 0.
2. L'état Load entre dans l'état Send inconditionnellement. Dans l'état Send, COM\_REG correspond à 0b01 et toutes les autres sorties : START\_TEMPO, GO\_0, GO\_1, sont encore à 0.
3. Si le compteur n'atteint pas 51, c'est-à-dire que la MAE n'a pas encore reçu la trame complète, la MAE entre dans l'état Compare pour distinguer si le bit reçu maintenant représente un bit '0' ou un bit '1'. Si S\_In = '0', elle entre dans l'état Bit0, sinon elle entre dans l'état Bit1.
4. Lorsque la MAE entre dans l'état Bit0 ou Bit1,  $GO\_0/GO\_1 \leq '1'$ . Ils commencent à donner des instructions à DCC\_Bit\_0 ou à DCC\_Bit\_1 pour qu'ils émettent un signal carré vers la sortie de la Centrale DCC. Après produire une sortie de signal, FIN\_0 ou FIN\_1 devient 1 et la machine d'état retourne à l'état Send, pour recommencer l'envoi des bits suivants.
5. Notez que nous ne pouvons pas ignorer ce qui se passe lorsque la machine d'état a reçu les 51 bits et que la machine d'état passe de Send à Tempo. Start\_Tempo devient '1'. Le module Tempo commence à compter le délai de temporisation. Lorsque ce délai est écoulé, un signal Fin\_Tempo est mis à 1 puis envoyé à MAE. La machine d'état retourne à Load. Une trame complète est maintenant reçue et le compteur passe à zéro à ce stade.

### 2.1.5 TOP

Enfin, nous pouvons définir le routage de l'ensemble du module dans TOP. Après avoir connecté les signaux d'entrée et de sortie de chaque module dans le port map, nous devons définir une porte OU qui joint les signaux de sortie de module DCC\_Bit\_0 et DCC\_Bit\_1 :

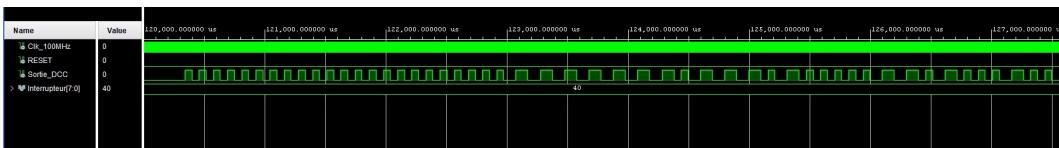
```
Sortie_DCC <= DCC_1 or DCC_0;
```

Dans le testbench de TOP (code VHDL ci-dessous), si tous les modules précédents fonctionnent correctement, il suffit de définir les signaux d'entrée pour tout le module, c'est-à-dire le signal d'horloge avec une fréquence de 100MHz, Reset, et les états des huit interrupteurs, et nous pouvons commencer à observer les valeurs de Sortie\_DCC.

```
Clk_100MHz <= not Clk_100MHz after 5 ns;
Reset <= '1' after 2 ns, '0' after 10 ns;

Interrupteur <= "10000000" after 50 ms, "01000000" after 100 ms, "00100000" after 150 ms,
               "00010000" after 200 ms, "00001000" after 250 ms, "00000010" after 300 ms,
               "00000010" after 350 ms, "00000001" after 400 ms;
```

Enfin, dans la simulation de comportement, nous pouvons voir que lorsque l'interrupteur est levé, la Centrale DCC génère les trames dont nous avons besoin pour pouvoir passer des instructions, comme nous l'attendions, par exemple, lorsque l'interrupteur[6] est levé, nous pouvons voir (schéma ci-dessous) que l'état de la trame est effectivement correct, correspondant à ce que nous avons conçu dans le générateur de trames avant, Trame\_DCC = 0x7fff0117cbb.



Simulation pour Interrupteur[7 :0] = "40"



Simulation de comportement de TOP

Nous commençons à nous préparer à la génération du bitstream.

## 2.2 Génération du Bitstream et Démonstration

### 2.2.1 Modification de Contraintes

Pour que les périphériques de la Nexys A7-100T reconnaissent les signaux d'entrée/sortie que nous avons définis, nous devons également modifier le fichier de contraintes Naxy4DDR\_Master.xdc.

D'abord, nous mettons en place le signal d'horloge.

```
## Clock signal
set_property -dict { PACKAGE_PIN E3    IOSTANDARD LVCMOS33 } [get_ports { Clk_100MHz }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {Clk_100MHz}];
```

Ensuite, nous configurons les interrupteurs dont nous avons besoin, les Interrupteur [0-7] sont utilisés pour ajuster la fonction et l'Interrupteur[15] est utilisé pour le Reset.

```
##Switches
set_property -dict { PACKAGE_PIN J15    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[1] }]; #IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[2] }]; #IO_L6N_T0_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13    IOSTANDARD LVCMOS33 } [get_ports { Interrupteur[7] }]; #IO_L5N_T0_D07_14 Sch=sw[7]

set_property -dict { PACKAGE_PIN V10    IOSTANDARD LVCMOS33 } [get_ports { RESET }]; #IO_L21P_T3_DQS_14 Sch=sw[15]
```

Enfin, nous définissons JA[4] pour le signal de Sortie\_DCC.

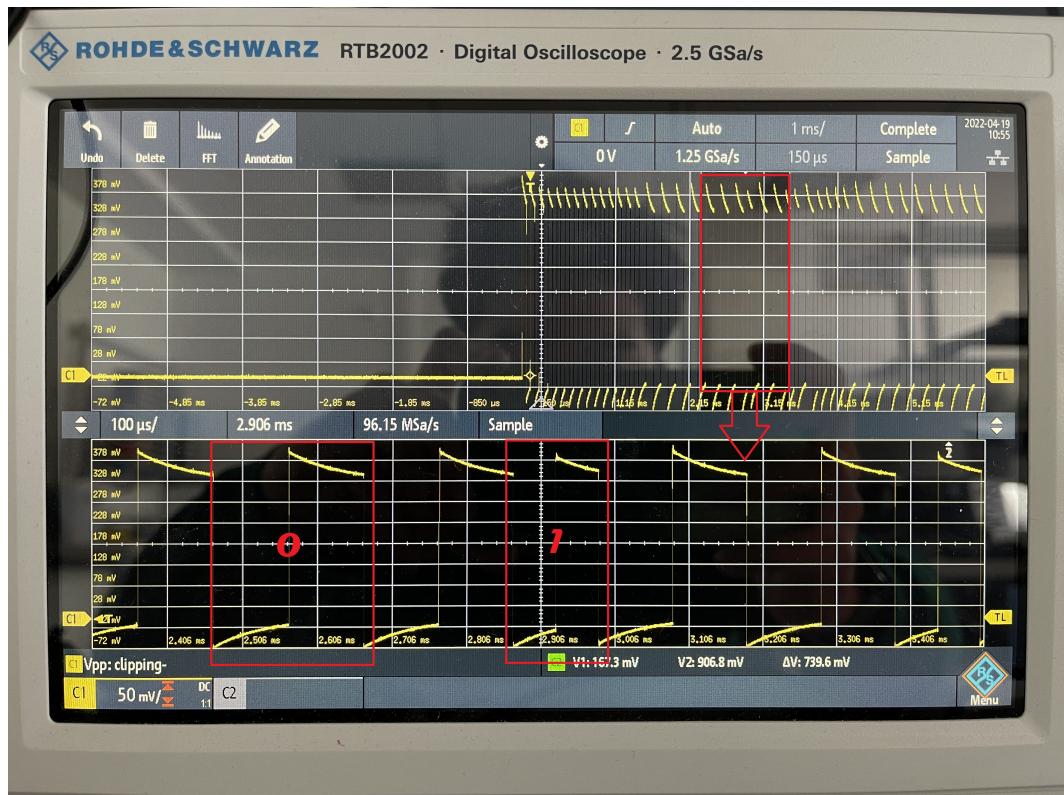
```
set_property -dict { PACKAGE_PIN G17    IOSTANDARD LVCMOS33 } [get_ports { Sortie_DCC }]; #IO_L18N_T2_A23_15 Sch=ja[4]
```

### 2.2.2 Post-Synthesis/Implementation Functional Simulation

Dans la simulation fonctionnelle post-synthèse et post-implémentation, nous avons également constaté certains problèmes que nous n'avions pas trouvés avant, comme le fait que certains signaux qui devaient figurer dans la liste de sensibilité n'étaient pas inclus. Heureusement, il n'y a pas eu d'autres problèmes majeurs.

### 2.2.3 Génération du Bitstream et Résultat

Enfin, nous avons réussi à générer un flux binaire. et testé les résultats sur un oscilloscope. Tout le code fonctionne avec succès. Enfin, sur la voie, le train a également fonctionné correctement et a rempli toutes ses fonctions comme prévu. La première partie de projet a été achevée avec succès.



Trame par default : Arrêt du Train d'Adresse 0b02 sur l'oscilloscope

# Chapitre 3

## Conception d'IP pour le Microblaze

### 3.1 Creation d'IP Centrale DCC

Pour cette section, nous devons suivre les etapes expliquees dans les sujets prec ´edents. Nous avons supprime le genateur de trames prec ´edent et commente le signal d'entree, Interrupteur, dans TOP.vhd. En effet, ce signal d'entree seront traite dans l'overlay qu'on va concevoir apres . Il est interessant de noter que, contrairement aux TPs prec ´edents, notre utilisation de slv\_reg a change.

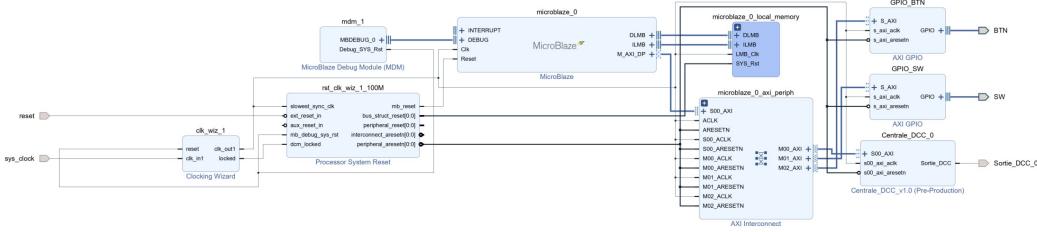
Dans le fichier Centrale\_DCC\_v1\_0\_S00\_AXI.vhd nous avons decide d'utiliser deux registres d'esclave. En effet, notre reg\_slv est un registre de 32 bits et si on veut stocker une trame de 51 bits, il nous faut au moins deux registres.

```
process (S_AXI_ACLK)
variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
begin
    if rising_edge(S_AXI_ACLK) then
        if S_AXI_ARESETN = '0' then
            slv_reg0 <= (others => '0');
            slv_reg1 <= (others => '0');
```

Apres initialisation de registres, on precise : reg\_slv0 correspond aux 32 bits de poids faible de la trame et le reg\_slv1 correspond aux 19 bits de poids fort de la trame.

```
-- Add user logic here
Trame(31 downto 0) <= slv_reg0;
Trame(50 downto 32) <= slv_reg1(18 downto 0);
```

Après avoir enveloppé les codes précédents dans une IP, le schéma bloc a finalement été créé et connecté.



Schema bloc avec Centrale DCC

En outre, nous avons également ajouté deux modules AXI\_GPIO pour gérer les entrées de signaux pour les boutons et les interrupteurs que nous utiliserons.

### 3.2 Modification de Contraintes

Comme nous allons introduire de nouveaux boutons poussoir, et de nouveaux interrupteurs, nous avons modifié le code associé dans le fichier Naxy4DDR\_Master.xdc. Pour des raisons pratiques, nous avons activé tous les boutons et interrupteurs sur la carte.

```
## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMS33 } [get_ports { sys_clock }]; #IO_L12P_T1_MRCC_35 Sch=clk100mhz
## create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {diff_clock_rtl_0}];

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[0] }]; #IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[1] }]; #IO_L3N_TO_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[2] }]; #IO_L6N_TO_D08_VREF_14 Sch=sw[2]
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[3] }]; #IO_L13N_T2_MRCC_14 Sch=sw[3]
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[4] }]; #IO_L12N_T1_MRCC_14 Sch=sw[4]
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[5] }]; #IO_L7N_T1_D10_14 Sch=sw[5]
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[6] }]; #IO_L17N_T2_A13_D29_14 Sch=sw[6]
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[7] }]; #IO_L5N_TO_D07_14 Sch=sw[7]
set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMS18 } [get_ports { SW_tr1_i[8] }]; #IO_L24N_T3_34 Sch=sw[8]
set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMS18 } [get_ports { SW_tr1_i[9] }]; #IO_25_34 Sch=sw[9]
set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[10] }]; #IO_L15P_T2_DQS_RDWR_B_14 Sch=sw[10]
set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[11] }]; #IO_L32P_T3_A05_D19_14 Sch=sw[11]
set_property -dict { PACKAGE_PIN H6       IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[12] }]; #IO_L24P_T3_35 Sch=sw[12]
set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[13] }]; #IO_L20P_T3_R08_D24_14 Sch=sw[13]
set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMS33 } [get_ports { SW_tr1_i[14] }]; #IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10     IOSTANDARD LVCMS33 } [get_ports { reset }]; #IO_L21P_T3_DQ5_14 Sch=sw[15]

##Buttons
#set_property -dict { PACKAGE_PIN C12     IOSTANDARD LVCMS33 } [get_ports { RESET }]; #IO_L3P_TO_DQS_AD1P_15 Sch=cpu_resetn

set_property -dict { PACKAGE_PIN N17     IOSTANDARD LVCMS33 } [get_ports { BTN_tr1_i[0] }]; #IO_L9P_T1_DQS_14 Sch=btnc
set_property -dict { PACKAGE_PIN M18     IOSTANDARD LVCMS33 } [get_ports { BTN_tr1_i[1] }]; #IO_L4N_TO_D05_14 Sch=bttnu
set_property -dict { PACKAGE_PIN P17     IOSTANDARD LVCMS33 } [get_ports { BTN_tr1_i[2] }]; #IO_L12P_T1_MRCC_14 Sch=bttnl
set_property -dict { PACKAGE_PIN M17     IOSTANDARD LVCMS33 } [get_ports { BTN_tr1_i[3] }]; #IO_L10N_T1_D15_14 Sch=bttnr
set_property -dict { PACKAGE_PIN P18     IOSTANDARD LVCMS33 } [get_ports { BTN_tr1_i[4] }]; #IO_L9N_T1_DQS_D13_14 Sch=btnd
```

### 3.3 Conception d'Overlay

Après avoir généré avec succès le nouveau bitstream, nous ouvrons Xilinx Vitis et commençons à concevoir l'Overlay.

Tout d'abord, nous avons défini quelques macros pour faciliter l'écriture du code.

```
#define SW0 0x001
#define SW1 0x002
#define SW2 0x004
#define SW3 0x008
#define SW4 0x010
#define SW5 0x020
#define SW6 0x040
#define SW7 0x080
#define SW8 0x100 //Choisir l'adresse de train
#define SW9 0x200 //Choisir la vitesse de train (avancer)
#define SW10 0x400 //Choisir la vitesse de train (reculer)

#define BTND 0x1 //Right: incrementer
#define BTNC 0x2 //OK, Envoyer
#define BTNG 0x4 //Left: Diminuer

#define STOP_REG0 0xF01180C5
#define STOP_REG1 0xFFFF
//stop trame :0x7FFFFF01180C5
```

Nous avons défini une fonction writeRegs() parce que plus tard, nous aurons souvent besoin d'écrire des valeurs des trame dans les registres d'esclave, et le fait de la définir la rend plus facile à programmer.

```
void writeRegs(int Trame_reg0, int Trame_regl){
    // Ecriture des valeurs des Trame reg0 dans le Slave Registre 0
    CENTRALE_DCC_mWriteReg(XPAR_CENTRALE_DCC_0_S00_AXI_BASEADDR, CENTRALE_DCC_S00_AXI_SLV_REG0_OFFSET,Trame_reg0);
    // Ecriture des valeurs des Trame reg1 dans le Slave Registre 1
    CENTRALE_DCC_mWriteReg(XPAR_CENTRALE_DCC_0_S00_AXI_BASEADDR, CENTRALE_DCC_S00_AXI_SLV_REG1_OFFSET,Trame_regl);
}
```

Nous avons également défini quelques fonctions simples qui sont spécifiquement chargées de modifier l'adresse de train ou sa vitesse de déroulement. Nous pouvons régler la vitesse en 8 shifts.

```
void incr_Adr(int adr){
    adr++;
    if(adr > 4){adr = 1;}
}

void dimi_Adr(int adr){
    adr--;
    if(adr == 1){adr = 4;}
}

void incr_Vit(int vit){
    vit = vit + 4;
    if(vit == 0b11111){vit = 0b00011;}
}

void dimi_Vit(int vit){
    vit = vit - 4;
    if(vit == 0b00011){vit = 0b11111;}
}
```

Enfin, nous voulons mettre en œuvre des fonctions telles que, par exemple, lorsque nous relevons l'interrupteur[10], nous pouvons sélectionner la vitesse à laquelle le train 2 recule en appuyant sur les boutons gauche et droit.

Les fonctionnalités plus détaillées peuvent être trouvées directement dans le code soumis avec le rapport. Pour cette partie, nous avons également effectué des tests sur les voies.

# Chapitre 4

## Amélioration et Conclusion

Bien que ce projet soit terminé, il y a encore quelques domaines que je dois améliorer. Pour la section Overlay, étant donné que le projet a été fait par moi seul, je n'ai pas eu le temps de le rendre trop parfait. J'aurais pu afficher l'adresse du train déjà sélectionnée par l'utilisateur sur l'afficheur LED 7 segments, ainsi que la vitesse sélectionnée, etc. On peut également confirmer une variable (adresse du train) sur le trame chaque fois que nous appuyons sur le bouton. Après l'avoir confirmée, vous pouvez passer à la variable suivante (avancer ou réculer, la vitesse...). Une fois que toutes les sélections ont été faites, on peut envoyer la trame complète sur les voies. C'est plus logique pour l'utilisateur.

Ce projet nous a familiarisés avec l'utilisation de Vivado, un logiciel puissant, et nous a aidés à démêler les idées de base du développement FPGA. Pour la première fois, nous avons pu concevoir nous-mêmes une IP complète et la faire fonctionner avec succès sur un FPGA. Pour autant que je sache, la création d'une IP pour un FPGA ne se limite pas à ce que nous avons appris jusqu'à présent. Nous pouvons également l'écrire en C++ dans Vitis HLS, définir les ports d'entrée/sortie, utiliser les fonctions de la bibliothèque pour aider à concevoir les algorithmes à l'intérieur d'une IP, la compresser dans un fichier .zip et l'importer dans le logiciel Vivado. Pour les cartes FPGA suffisamment puissantes, comme Xilinx Pynq Z2 avec une architecture hétérogène, nous pouvons copier l'image Linux sur la carte et effectuer l'importation du fichier .bit et du fichier .hwh (tous deux exportés par Vivado) dans l'environnement Linux en utilisant Python. Cela vaut la peine d'être exploré dans de futurs cours.

Page laissée intentionnellement vide.

Sorbonne Université  
Campus Pierre et Marie Curie,  
4 Pl. Jussieu,  
75005 Paris