COMPTE RENDU FPGA MINI PROJET

LONGUET Axel & LAKHLEF Rayan

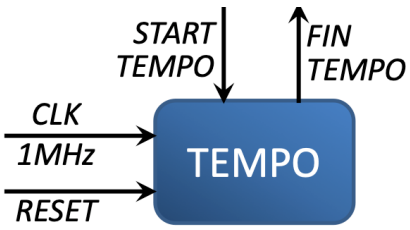
But du projet

Le but de ce mini-projet était de réaliser un générateur de trame DCC sur la carte FPGA Nexys. Dans le cadre de ce projet, nous avons réussi à le terminer complètement en implémentant les différentes fonctionnalités demandées, et cela, également pour la partie MICROBLAZE.

Architecture de la centrale DCC

Compteur_Tempo

Ce composant, qui est synchronisé avec l'horloge CLK_1MHz, a pour but de mesurer l'écart de temps entre deux trames DCC, ici dans notre cas 6 ms.



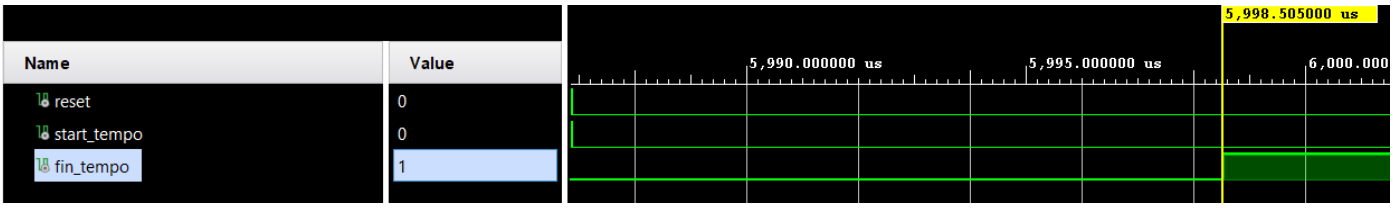
En analysant ce composant, on définit nos ports de la façon suivante :

```
Port ( Clk     : in STD_LOGIC; -- Horloge 100 MHz
       Reset     : in STD_LOGIC; -- Reset Asynchrone
       Clk1M     : in STD_LOGIC; -- Horloge 1 MHz
       Start_Tempo   : in STD_LOGIC; -- Commande de Demarrage de la Temporisation
       Fin_Tempo   : out STD_LOGIC -- Drapeau de Fin de la Temporisation
);
```

Nous avons :

- Clk notre horloge à 100 Mhz
- Le reset asynchrone.
- Clk1M l'horloge à 1Mhz
- Start_Tempo qui est généré par la MAE et qui lance la temporisation
- Fin_Tempo qui est mis à 1 lorsque notre temporisation est terminée, Ce signal est automatiquement remis à 0 dès que la commande Start_Tempo est remise à 0

Simulation



Diviseur_Horloge

Le rôle de ce module est de produire un signal d'horloge de 1 MHz à partir de notre horloge de 100 MHz. Cette horloge est ensuite utilisée pour al gestion de nos temps d'attente pour le protocole DCC.

En analysant ce composant, on définit nos ports de la façon suivante :

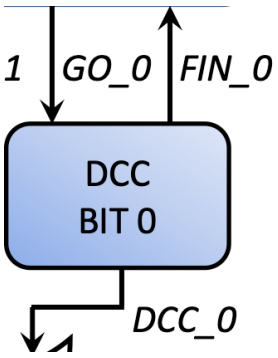
```
Port ( Reset   : in STD_LOGIC; -- Reset Asynchrone
       Clk_In    : in STD_LOGIC; -- Horloge 100 MHz de la carte Nexys
       Clk_Out   : out STD_LOGIC);     -- Horloge 1 MHz de sortie
```

Avec :

- Le reset asynchrone
- Clk_IN notre horloge de base à 100Mhz
- Clk_Out l'horloge de sortie à 1Mhz

DCC_Bit_0

Ce module permet de générer un bit 0 selon le protocole DCC. Ce qui correspond à une impulsion à 0 de 100 µs puis une impulsion à 1 de 100 µs.



En analysant ce composant, on définit nos ports de la façon suivante :

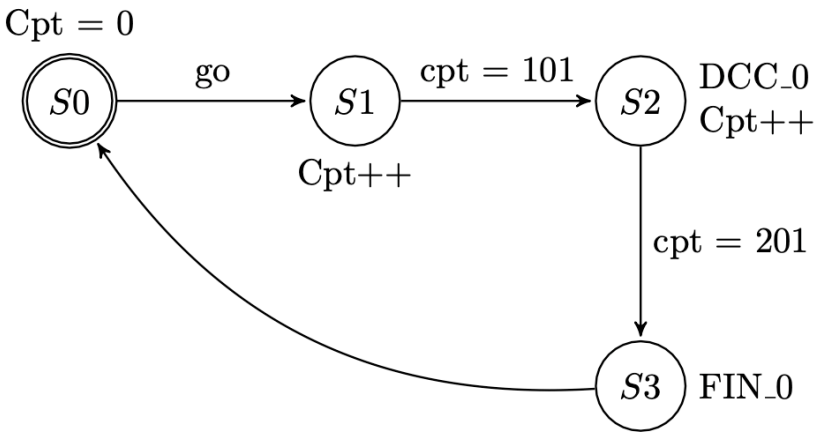
```
Port (CLK_100MHz : in std_logic;
      CLK_1MHz   : in std_logic;
      reset      : in std_logic;
      go         : in std_logic;
      fin        : out std_logic;
      DCC_0      : out std_logic
    );
```

Nous avons :

- CLK_100MHz l'horloge de notre carte à 100Mhz
- CLK_1MHz l'horloge divisée à 1Mhz
- le reset asynchrone
- go la commande envoyée par le module MAE pour lancer la génération du bit
- fin la commande renvoyé au module MAE lorsque la génération du bit est terminée
- DCC_0 le signal correspondant à notre bit

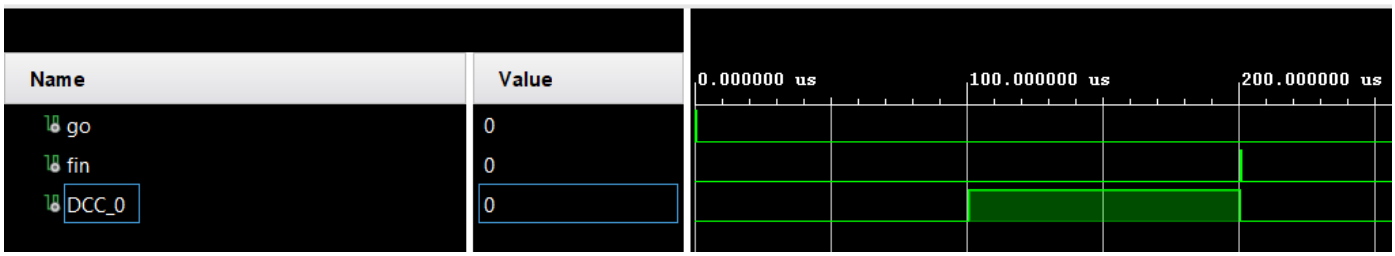
Dans ce module, on trouve une machine à états et un compteur

La Machine à Etat et compteur



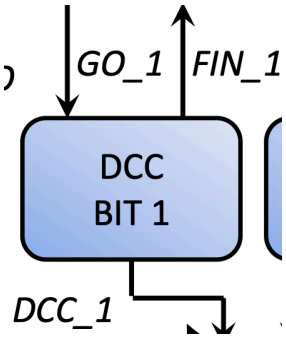
Dans notre MAE, on reste 100us dans S1 pour générer le signal DCC_0 = 0, puis 100us dans S2 pour générer le signal DCC_0 = 1. Au final, nous avons généré complètement notre signal DCC codant un 0, on peut donc mettre le signal FIN à 1 pour indiquer à la MAE que notre 0 a été généré.

Simulation



DCC_Bit_1

Ce module permet de générer un bit 1 selon le protocole DCC. Ce qui correspond à une impulsion à 0 de 58 µs puis une impulsion à 1 de 58 µs.



En analysant ce composant, on définit nos ports de la façon suivante :

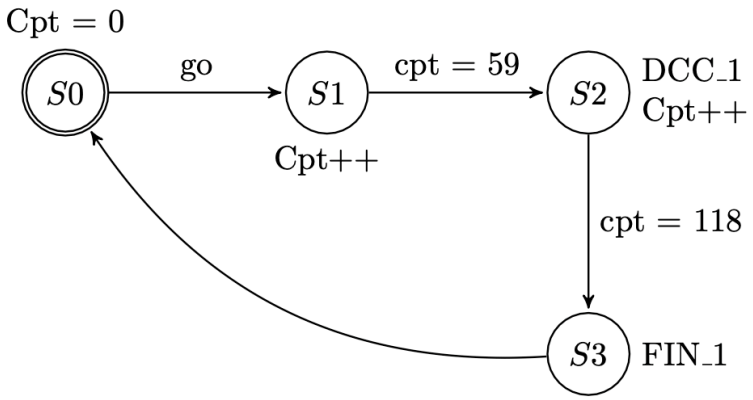
```
Port (CLK_100MHz : in std_logic;
      CLK_1MHz   : in std_logic;
      reset      : in std_logic;
      go         : in std_logic;
      fin        : out std_logic;
      DCC_1      : out std_logic
    );
```

Nous avons :

- CLK_100MHz l'horloge de notre carte à 100Mhz
- CLK_1MHz l'horloge divisée à 1Mhz
- le reset asynchrone
- go la commande envoyée par le module MAE pour lancer la génération du bit
- fin la commande renvoyé au module MAE lorsque la génération du bit est terminée
- DCC_1 le signal correspondant à notre bit

Dans ce module, on trouve une machine à états et un compteur

La Machine à Etat et compteur



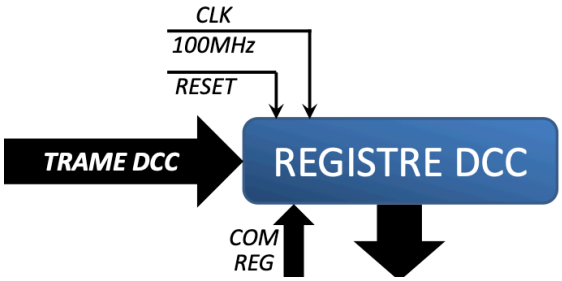
Dans notre MAE, on reste 58us dans S1 pour générer le signal DCC_1 = 0, puis 58us dans S2 pour générer le signal DCC_1 = 1.

Au final, nous avons généré complètement notre signal DCC codant un 0, on peut donc mettre le signal FIN à 1 pour indiquer à la MAE que notre 1 a été généré.

Simulation

Registre_DCC

Registre DCC est un registre à décalage qui sauvegarde la trame DCC envoyée par l'utilisateur. On effectue ensuite des décalages pour transmettre cette trame bit par bit. Ce module est contrôlé par le module MAE pour savoir quand il doit sauvegarder la trame ou effectuer un décalage.



On en déduit donc les ports suivants :

```
Port (
        clk_100MHz : in std_logic;
        reset      : in std_logic;
        com_reg    : in  std_logic_vector(1 downto 0);
        Trame_DCC  : in STD_LOGIC_VECTOR(50 downto 0);
        bit_s      : out std_logic
     );
```

Avec :

- clk_100MHz notre horloge à 100Mhz
- reset, le reset asynchrone
- com_reg la commande à effectuer envoyé par la MAE

- Trame_DCC la trame DCC envoyé par l'utilisateur à sauvegarder
- bit_s le bit courant de notre trame

Realisation

Tous d'abord, nous utilisons un signal interne pour sauvegarder notre trame DCC :

```
signal trame_reg : std_logic_vector(50 downto 0)
```

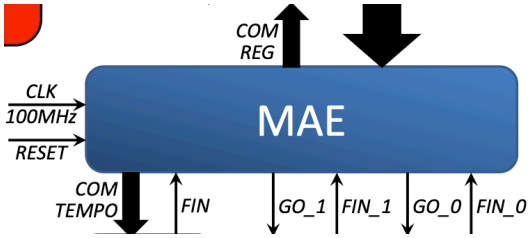
Ensuite, en fonction de notre commande, nous effectuons plusieurs actions :

```
when "00" => null;
when "01" => trame_reg <= trame_dcc;
when "10" => bit_s <= trame_reg(50) ; trame_reg <= trame_reg(49 downto 0) & '1';
when "11" => null;
when others => null;
```

Quand nous recevons la commande 01, on sauvegarde la trame courante dans le registre.
Avec la commande 10, on décale notre registre vers la gauche et le bit sortant de notre registre est envoyé sur la sortie bit_s.
Les autres commandes sont désactivées.

MAE

C'est ce module qui va contrôler les différents autres blocs de notre système. Pour ce faire, il enverra différente commande aux différents blocs. Cela a pour but de générer la trame DCC qui aura été envoyer par un utilisateur.



En analysant le schéma, on en déduit la déclaration suivante de notre module :

```
Port (
    FIN_1,FIN_0,FIN       : IN std_logic;
    GO_1,GO_0,START_TEMPO : OUT std_logic;
    COM_REG               : out std_logic_vector(1 downto 0 );
    CLK_100Mhz            : in std_logic;
    reset                 : in std_logic;
    bit_s                 : in std_logic;
 );
```

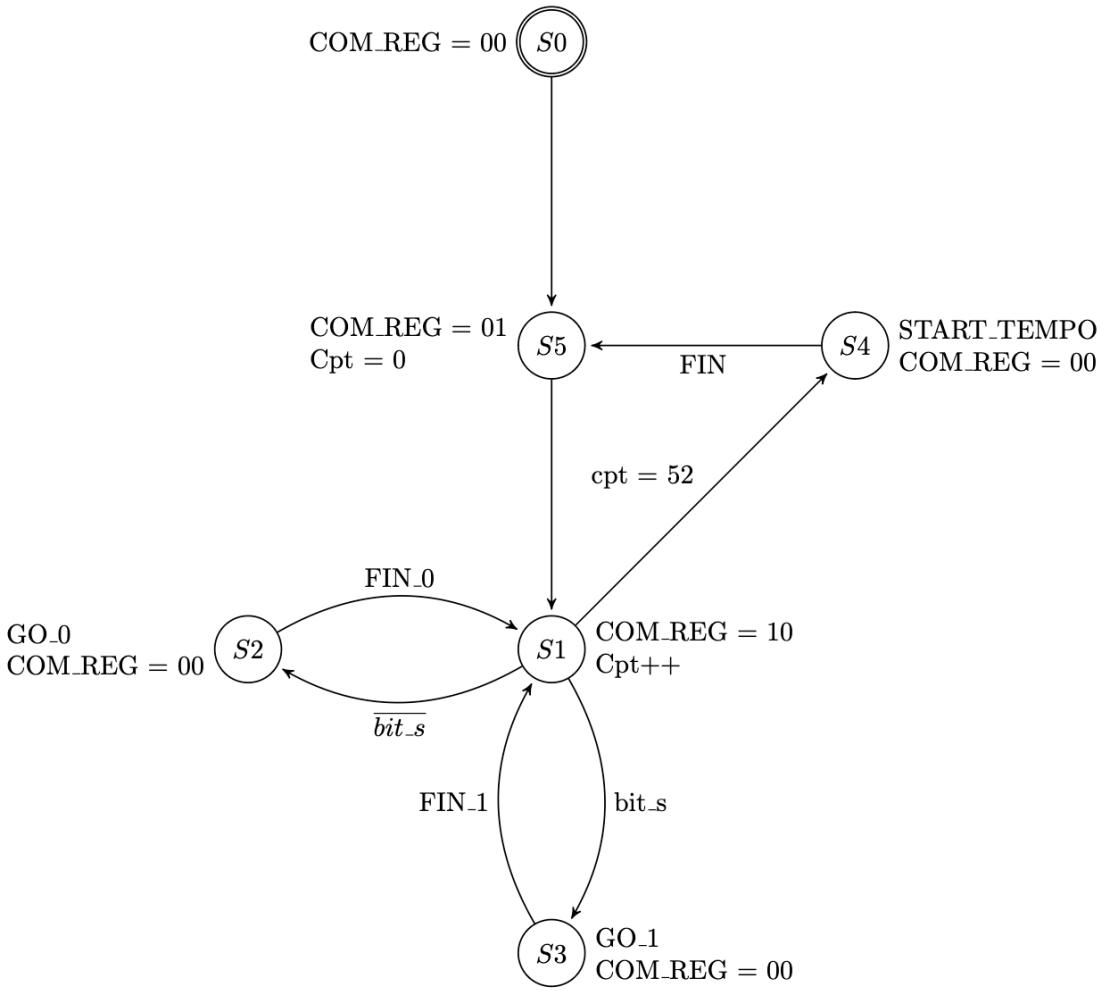
Avec :

- FIN_1,FIN_0,FIN les signaux indiquant la fin de respectivement la génération du bit 1, du bit 0 et de la temporisation
- GO_1,GO_0,START_TEMPO les signaux indiquant l'activation de respectivement la génération du bit 1, du bit 0 et de la temporisation
- COM_REG la commande envoyée au registre_DCC
- CLK_100Mhz l'horloge à 100Mhz
- reset, le reset asynchrone
- bit_s le bit provenant de registre_DCC

Nous utiliserons aussi un compteur pour savoir combien de bit de notre trame ont été générés

MAE

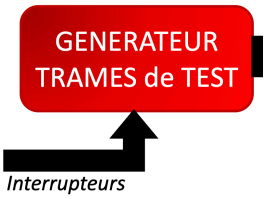
Pour faire fonctionner ce module, on définit la MAE suivante :



Dans S5, nous indiquons au registre_DCC qu'il doit sauvegarder une nouvelle trame et nous mettons notre compteur à 0.
Dans S1, nous allons récupérer le bit du registre_DCC et incrémenter notre compteur. En fonction de la valeur de ce bit, nous irons soit dans S2 pour générer un bit 0 ou dans S3 pour un bit 1. Une fois que toute notre trame a été générée, on passe dans S4 pour effectuer notre temporisation de 6ms. Une fois cette temporisation finie, nous allons dans S5 pour recommencer ces opérations pour une nouvelle trame.

Generateur_Trames

Le but de ce bloc est de générer une trame DCC sur sa sortie en fonction de la valeur des switch en entrée :



Avec comme ports Interrupteur en entrer et Trame_DCC en sortie sur respectivement 8 et 51 bits.

Top

Le but du module top est de connecter tous nos précédents module ensemble pour que notre génération de trames soit fonctionnelle

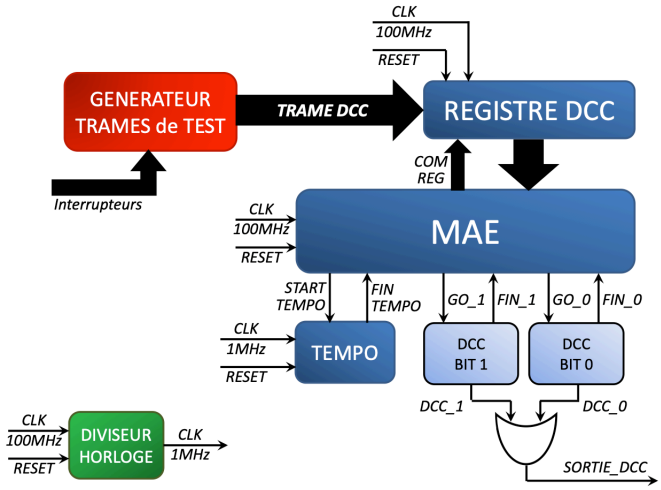


Figure 3 – Architecture de la Centrale DCC

En analysant cette figure, on définit les ports suivants :

```
Port (Clk_In       : in std_logic;
      reset        :  in std_logic;
      Interrupteur : in  STD_LOGIC_VECTOR(7 downto 0);
      sortie_DCC   : out std_logic
     );
```

Avec :

- Clk_In l'horloge de notre carte Nexys
- Reset le reset asynchrone de notre carte
- Interrupteur les différents switch de notre carte

- sortie_DCC notre signal DCC

Réalisation

Pour réaliser ce module nous allons tous d'abord déclarer les signaux suivants :

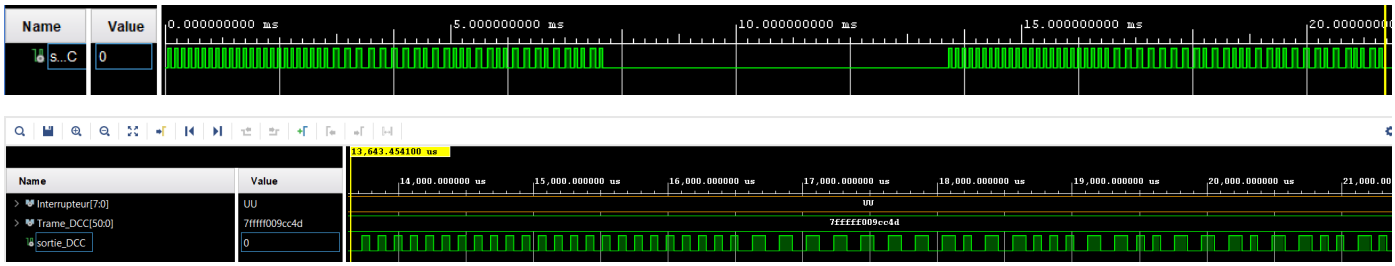
```
signal Trame_DCC  : STD_LOGIC_VECTOR(50 downto 0);
signal start_tempo     : std_logic;
signal Clk_Out         :  STD_LOGIC;
signal DCC_0           : std_logic;
signal DCC_1           : std_logic;
signal FIN_1,FIN_0,FIN : std_logic;
signal GO_1,GO_0       : std_logic;
signal COM_REG         : std_logic_vector(1 downto 0 );
signal bit_s           : std_logic;
```

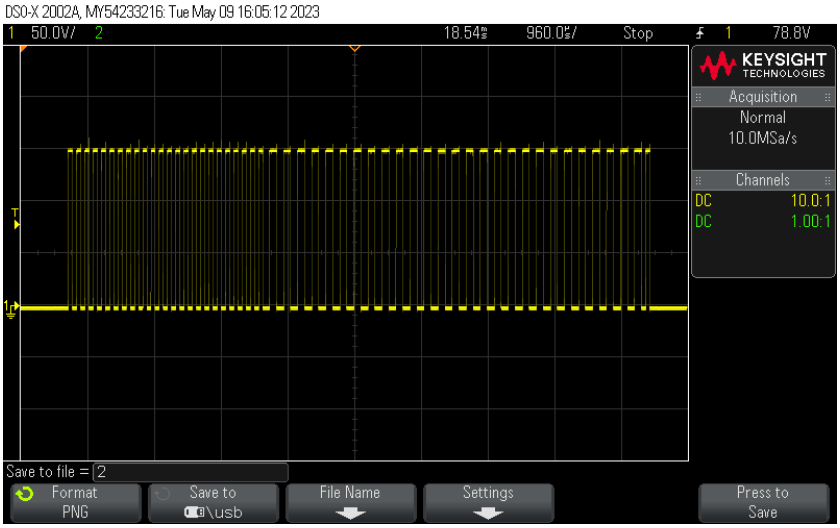
Ces différents signaux seront ensuite connectés à nos modules en suivant le schéma précédent. Clk_out correspond à notre horloge de 1Mhz en sortie du module diviseur. Bit_s est le bit en provenance du module Registre vers le module MAE.

De plus, pour totalement reproduire le schéma, nous devons encore effectuer le OU entre DCC_1 et DCC_0 On définit donc la sortie du module TOP de la façon suivante :

```
sortie_DCC <= DCC_0 or DCC_1;
```

Simulation





Partie 2 : CONCEPTION D'IP POUR LE MICROBLAZE

Le but de cette partie était de créer un IP MICROBLAZE à partir de notre Centrale_DCC. Notre idée est d'avoir 2 slaves registers de 32 bits qui vont contenir une partie de la trame DCC chacun.

Création de l'IP DCC

Block TOP

En reprenant le block TOP de la partie précédente, on change uniquement les ports en supprimant l'entrée des switch et en ajoutant une entrée TRAME_DCC qui sera généré grâce à nos slaves Register.

De plus nous supprimons l'instanciation du block Generateur_Trames qui ne sera plus utilisé ici, car la génération se fera par l'intermédiaire des slaves registers.

Ajout de SORTIE DCC sur l'AXI

On ajoute dans notre fichier centrale_DCC_v1_0.vhd la déclaration de notre signal de sortie sortie_DCC

On lui ajoute la sortie :

```
port (  -- Users to add ports here
        sortie_DCC  : out STD_LOGIC;
        -- User ports ends
```

De même dans la déclaration de centrale_DCC_v1_0_S00_AXI dans son architecture :

```
port ( Sortie_DCC   : out STD_LOGIC;
```

Et enfin, on affecte la valeur dans le port MAP :

```
port map ( Sortie_DCC => Sortie_DCC,
```

Instanciation de TOP dans l'AXI

Il faut maintenant instancier notre bloc TOP dans le fichier centrale_DCC_v1_0_S00_AXI.vhdl pour que nous puissions utiliser les slaves registers pour envoyer la Trame_DCC. On lui ajoute donc dans un premier temps la sortie DCC dans ses ports :

```
port ( -- Users to add ports here
       Sortie_DCC   : out STD_LOGIC;
       -- User ports ends
```

Maintenant, nous allons instancier notre bloc TOP, on déclare tous d'abord un signal qui va nous permettre d'utiliser les slaves register pour transmettre l'entrée DCC à notre module top :

```
signal Trame_DCC : STD_LOGIC_VECTOR(50 downto 0);
```

On instancie notre bloc top :

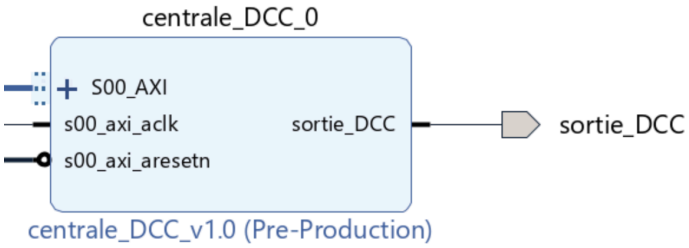
```
-- Add user logic here
L0: entity work.top port map (S_AXI_ACLK,S_AXI_ARESETN,Trame_DCC,Sortie_DCC);
```

Ensuite, nous pouvons affecter nos 2 slaves registers à l'entrée DCC de notre module TOP :

```
Trame_DCC <= slv_reg0 & slv_reg1( 18 downto 0);
-- User logic ends
```

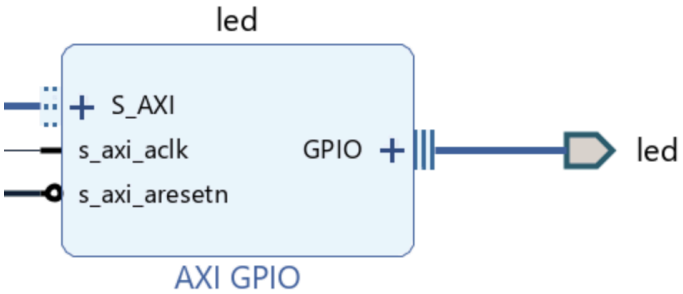
Intégration de l'IP au système Microblaze

Nous avons dans un premier temps ajouter notre IP centrale_DCC créer précédemment dans notre bloc design avec sa sortie DCC :

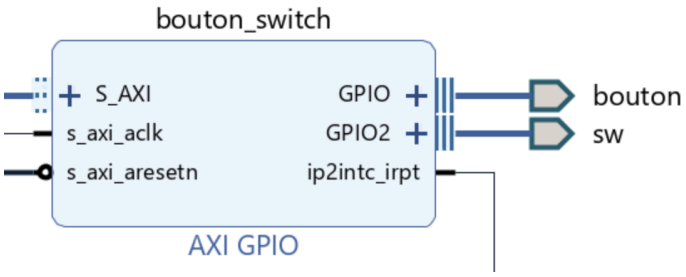


Ensuite, nous avons défini 2 GPIO pour pouvoir interagir avec la carte.

Le GPIO en output led sur 16 bits qui va nous permettre d'afficher des informations à l'utilisateur.



Le GPIO en input bouton_switch qui va nous permettre de récupérer les valeurs des boutons et des switchs.



VITIS

Une fois le design validé, nous sommes passés sur la partie programmation en c en utilisant VITIS.

Principe de fonctionnnement :

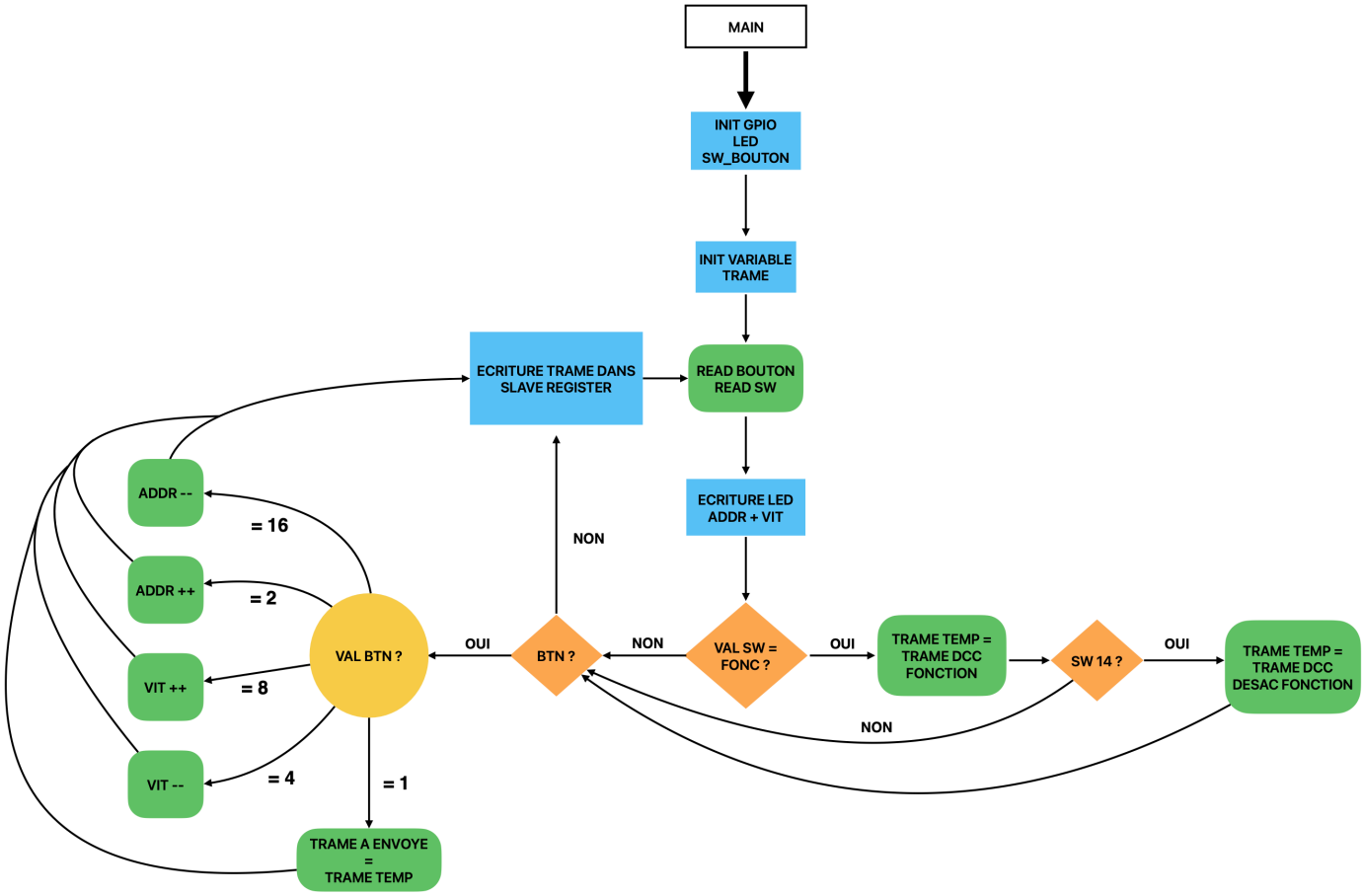
L'idée globale du programme est que l'utilisateur va choisir la fonction qu'il veut envoyer en utilisant les switchs.

La fonction est codée en base 2. Par exemple, pour choisir la fonction 3 il faut activer la switch 0 et 1. Pour la désactivation d'une fonction (par exemple les fonctions qui produisent du son), il faut, en plus des switchs pour la fonction, activer la swicth 14

L'utilisateur peut aussi augmenter ou réduire la vitesse du train avec les boutons droite et gauche. Il peut aussi choisir quel est le train dont la trame est destinée en utilisant les boutons haut et bas.

Enfin, lorsque l'utilisateur a choisi ces différents paramètres, il doit appuyer sur le bouton du milieu pour envoyer la trame.

Organigramme du code :



Réalisation :

Nous avons réalisé toutes les fonctions fournies par le protocole DCC. Dans la suite de cette partie, nous allons vont présenter le globalement sans trop rentrer dans les détails, le principe de notre code C pour notre programme DCC. Nous n'allons également pas montrer toutes les fonctions pour éviter d'avoir un rapport trop long, mais toutes les fonctions ont été implémentées et fonctionnent.

Initialisation des GPIO :

Dans un premier temps nous allons initialiser nos GPIO LED en sortie et SW_BOUTON en entrée

```
XGpio led;
XGpio button_sw;
```

Constante de notre TRAME DCC

Pour rendre notre code plus clair, nous allons définir plusieurs constantes.

Tous d'abord une constante pour gérer notre préambule dans le cas où nous avons une fonction sur 1 octet ou 2 octets :

```
u32 preambu_long = 0b11111111111111111111111;
u32 preambu_short = 0b11111111111111;
```

Ensuite, nous définissons la vitesse et l'adresse de base pour nos trames :

```
u32 vit = 0b00000;
u32 addr = 0b00000011;
```

On définit ensuite ces variables pour respectivement le premier octet de la fonction, le deuxième et le résultat du XOR :

```
u32 fonction, fonction1, xor;
```

Enfin, on définit la trame de base que nous allons envoyer au RESET qui correspond à la trame de RESET des trains :

```
u32 send_trame = 0b11111111111111111111111000000000;
u32 send_trame1 = 0b0000000000000000001;
```

Lecture des switchs et boutons :

Nous allons lire et stocker dans ces variables, les valeurs des swicths et des boutons

```
val = XGpio_DiscreteRead(&button_sw,2);
btn = XGpio_DiscreteRead(&button_sw,1);
```

Trames temporaire :

Comme le changement de TRAME ne s'effectue uniquement lorsque l'on appuie sur le bouton central, nous utilisons des variables temporaires pour stocker la possible nouvelle TRAME

```
u32 trame, trame1;
```

Exemple de fonction 1 (ou 2) octet(s) :

Pour toutes nos fonctions, on effectue le traitement suivant (Avec quelque petit changement en fonction des cas)

Dans un premier temps, on regarde si la valeur des switch correspond au numéro de notre fonction :

```
if ((val & 0xFF) == 1)
```

Ensuite, on affecte la bonne valeur à la variable fonction et on calcule le xor pour le checksum :

```
// 1 OCTET                  // DEUX OCTETS
fonction = 0b10010000;      // fonction  = 0b11011110;
                            // fonction1 = 0b00000010;
xor = addr ^ fonction;      // xor = (addr ^ fonction) ^ fonction1;
```

Enfin, on affecte les bonnes variables aux bons bits de nos trames grâce à des décalages :

```
trame = (preambu_long << 9) + (0b0 << 8) + addr;
trame1 = (0b0 << 11) + (fonction << 10) + (0b0 << 9) + (xor << 1) + 0b1;
```

Dans le cas des fonctions à 2 octets, on utilise le préambule short et les décalages sont différents pour la premiere partie de la trame.

Exemple de désactivation fonction :

Pour toutes les fonctions de désactivation, on effectue le traitement suivant.

On commence par vérifier que la switch 14 est activée avec une certaine fonction

```
if ((((val & 0xFF) == 1) || ((val & 0xFF) == 2) || ((val & 0xFF) == 3) || ((val & 0xFF)
== 4) || ((val & 0xFF) == 5)) && (val & 0b100000000000000))
```

On affecte à fonction la valeur correspondant à la désactivation (que des 0). Puis, on effectue le calcul de checksum :

```
fonction = 0b10000000;
xor = addr ^ fonction;
```

Changement Vitesse :

Pour changer de vitesse, on doit vérifier si l'on appuie sur le bouton de gauche ou de droite :

```
if (btn & 8) // Reduction : if (btn & 4)
```

Ensuite, on augmente la vitesse (ou on la réduit). On peut donc coder la bonne fonction et calculer le checksum.

```
vit = (vit + 0b1) % 0b100000; // Reduction : vit = (vit - 0b1) % 0b100000;
```

Changement Adresse :

Pour changer d'adresse, on doit vérifier que l'on appuie soit sur le bouton haut ou bas :

```
if (btn & 2) // Reduction : if (btn & 16)
```

En fonction de cela, on augmente (ou réduit) la valeur de l'adresse.

```
addr = (addr + 0b1) % 0b100000000; // Reduction : addr = (addr - 0b1) % 0b100000000;
```

Envoie trames :

Pour envoyer les trames, on doit vérifier que le bouton central est appuyé.

```
if (btn & 1)
```

Ensuite, on affecte nos trames temporaires à nos trames à envoyer :

```
send_trame = trame;
send_trame1 = trame1;
```

Ecriture Slave register :

Pour l'écriture de nos slaves registers, on va écrire la variable trame dans le slave register 0 qui va coder les 32 premiers bits de notre TRAME DCC. Ensuite la trame 1 dans le slave register 1 qui va coder les 19 bits restants de notre TRAME DCC.

```
CENTRALE_DCC_mWriteReg(XPAR_CENTRALE_DCC_0_S00_AXI_BASEADDR,
CENTRALE_DCC_S00_AXI_SLV_REG0_OFFSET,send_trame );
CENTRALE_DCC_mWriteReg(XPAR_CENTRALE_DCC_0_S00_AXI_BASEADDR,
CENTRALE_DCC_S00_AXI_SLV_REG1_OFFSET,send_trame1);
```