



BUREAU D'ÉTUDE VHDL:

Réalisation d'un pilote de barre franche pour un voilier

Encadré par:

Mr . Thierry PERISSE

Réalisé par:

TISSAFI JAOUHARI M'hammed

FARHAT Rida

Table des matières

| | |
|--|----|
| 1- Présentation :..... | 4 |
| 1.1 Matériels de mise en œuvre : | 5 |
| 2- Réalisation du projet:..... | 5 |
| 2.1 Fonctionnement de l'anémomètre..... | 7 |
| a. Analyse fonctionnelle..... | 7 |
| b. Implémentation : | 8 |
| c. Spécification du module gestion_anemometre :..... | 8 |
| d. Simulation : | 9 |
| e. Conception du SOPC | 12 |
| 2.2 Gestion du vérin : | 14 |
| a. Fonctionnement du vérin: | 14 |
| b. Analyse fonctionnelle..... | 14 |
| c. Implementation : | 16 |
| d. L'analyseur logique Signal Tap | 16 |
| e. Conception du SOPC | 18 |

Table des figures

| | |
|--|----|
| Figure 1: Vue générale de barre franche | 4 |
| Figure 2: Décomposition de pilote de barre franche..... | 5 |
| Figure 3: carte DE0-Nano | 5 |
| Figure 4: schéma fonctionnel de l'anémomètre | 7 |
| Figure 5: BSF de l'anémomètre..... | 8 |
| Figure 6: clk, frq, stim process dans le testbench | 9 |
| Figure 7: parametre de simulation | 10 |
| Figure 8: Modélisme interface..... | 10 |
| Figure 9: Information dans la Console de modélisme | 11 |
| Figure 10 : signal carré de 90Hz générer par le GBF et la sortie de la carte DE0..... | 11 |
| Figure 11: Les composants dans platform designer | 12 |
| Figure 12: Le block générer par plateform designer..... | 13 |
| Figure 13 : Code en C | 13 |
| Figure 14: la réponse de la carte après le téléchargement de code en C..... | 14 |
| Figure 15: les fonctions principales du circuit gestion de vérin..... | 14 |
| Figure 16: schéma fonctionnel du vérin | 15 |
| Figure 17: BSF du système de gestion de vérin | 16 |
| Figure 18: interface de l'analyseur logique..... | 16 |
| Figure 19: les paramètres du trigger in..... | 17 |
| Figure 20: Nombre des fronts d'horloge quand cs est nul | 17 |
| Figure 21: Valeur minimal de l'angle de barre..... | 17 |
| Figure 22: Valeur maximale de l'angle de barre | 17 |
| Figure 23: Valeur arbitraire de l'angle de barre..... | 18 |
| Figure 24: composant de l'interface Avalon du vérin dans platform designer..... | 18 |
| Figure 25: Le block générer par plateform designer de l'interface avalon du vérin..... | 19 |
| Figure 26: les registres de l'interface Avalon | 19 |
| Figure 27: la valeur de l'angle de barre et les autres varibales | 20 |
| Figure 28: mémoire de notre système..... | 20 |

Introduction

Dans le cadre de notre formation Master Systèmes et Microsystèmes Embarqués de l'Université Paul Sabatier Toulouse III, nous sommes amenés à concevoir le pilote de barre franche sous forme d'un système sur puce programmable SOPC décrite à l'aide du VHDL (Very High Speed Hardware Description Language) basé sur l'analyse de la spécification et la décomposition fonctionnelle du système choisi et la conception du circuit d'interface numérique en VHDL, effectuer une vérification de simulation sur le modèle, programmer un processeur (CPU), puis l'implémenté sur un système reprogrammable (FPGA) NIOS II en langage C, Avalon pour vérifier le fonctionnement de SOPC afin d'obtenir un système complet qui permet de réaliser les différentes missions spécifiques.



1- Présentation :

Le pilote automatique est un appareil électronique qui maintient un navire dans une direction prédéterminée sans intervention humaine. Les pilotes automatiques modernes sont des appareils très fiables qui maintiennent avec précision le cap d'un navire même dans les conditions les plus difficiles. Ces aides sont particulièrement utiles sur les voiliers comme le ray marine ev 400 sail. Ils aident à maintenir le cap du bateau lors de la montée ou de la descente des voiles. Ils aident également le bateau à garder le cap dans des conditions de vent changeantes.

Notre système se compose d'une unité de commande pilote équipée de vérins. Les boutons de la commande servent à faire entrer et sortir le vérin qui, en entrant ou sortant, pilote la barre franche du voilier, avec ses boutons on peut aussi choisir un mode de navigation pour le voilier.

En connexion avec ce système, on trouve une boussole (compas), un GPS, une interface NMEA pour le GPS et la commande du vérin, une girouette pour calculer la direction du vent et un anémomètre pour la vitesse du vent. Toutes ces données seront utilisées pour calculer la trajectoire et permettre au voilier de naviguer selon plusieurs modes (Conservateur d'Allure, Automatique...)

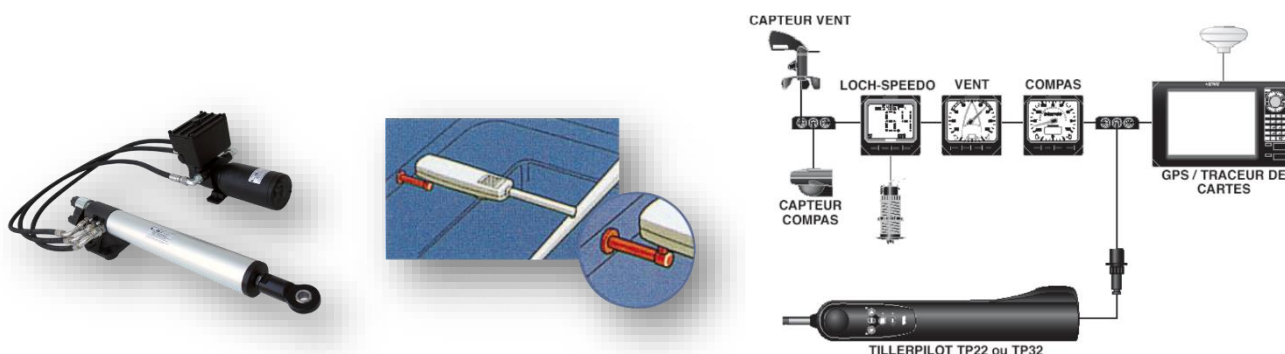


Figure 1: Vue générale de barre franche

Le système à réaliser est divisé en sous-systèmes, représenté dans la figure ci-dessous :

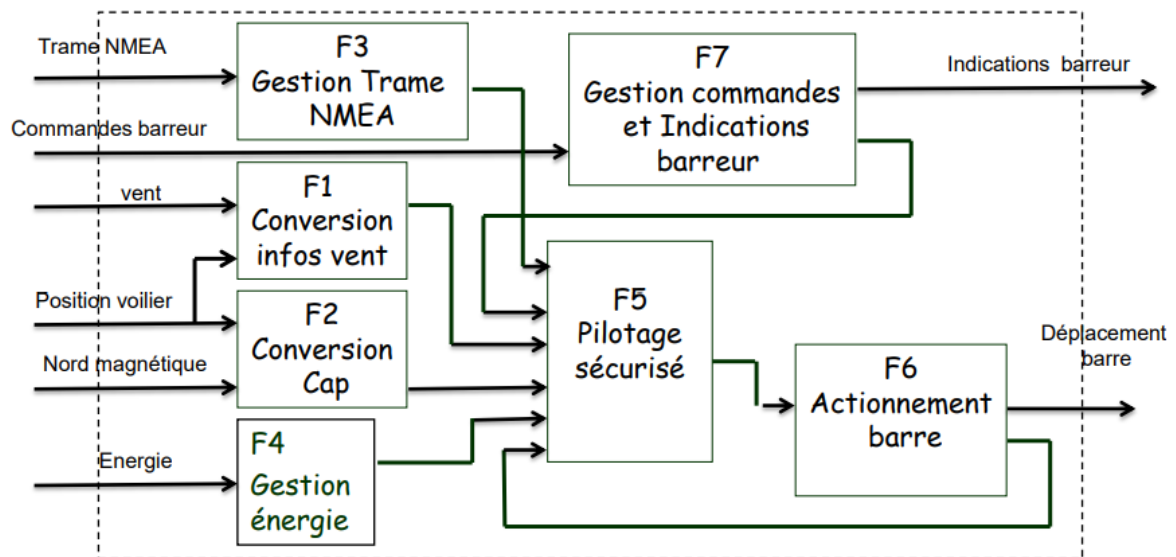


Figure 2: Décomposition de pilote de barre franche

1.1 Matériels de mise en œuvre :

- Carte DE0 – Nano :

La carte DE0-Nano est une version compacte des cartes FPGA standard, adaptée à une large gamme de projets de conception portables, tels que des robots et des projets mobiles comme dans notre cas, on l'utilisera pour réaliser le traitement de données collectées par les différents capteurs et réaliser les commandes de notre système.



Figure 3: carte DE0-Nano

2- Réalisation du projet :

Pour notre projet, on réalisera deux parties :

- Partie acquisition (fonction simple)

C'est une partie où on traite les données reçues d'un **anémomètre** et les affiche. Pour l'implémenter, on devra lire un signal en entrée d'une fréquence qui varie entre 0 Hz et 250 Hz. Ce signal devra ensuite être traduit en vitesse de vent qui varie aussi de 0 à 250 km/h.



Type de mesure : Mesure de la vitesse du vent

Signal de sortie : logique fréquence variable 0 à 250 Hz

Limite de destruction : Supérieure à 75 m/s (270 km/h)

Température de fonctionnement : -30 à +70 C

Plage : 0-250Km/h

- Commande

Dans la deuxième partie, celle de commande (fonction complexe), on a la fonction réalisant le mouvement de la barre franche est réalisée avec le **circuit vérin**. Ce circuit est composé de 5 fonctions principales qui vont permettre le pilotage du vérin qui contrôle la barre franche du voilier.



2.1 Fonctionnement de l'anémomètre

a. Analyse fonctionnelle

Pour répondre aux exigences du circuit que nous allons concevoir, nous avons réalisé la description fonctionnelle suivant :

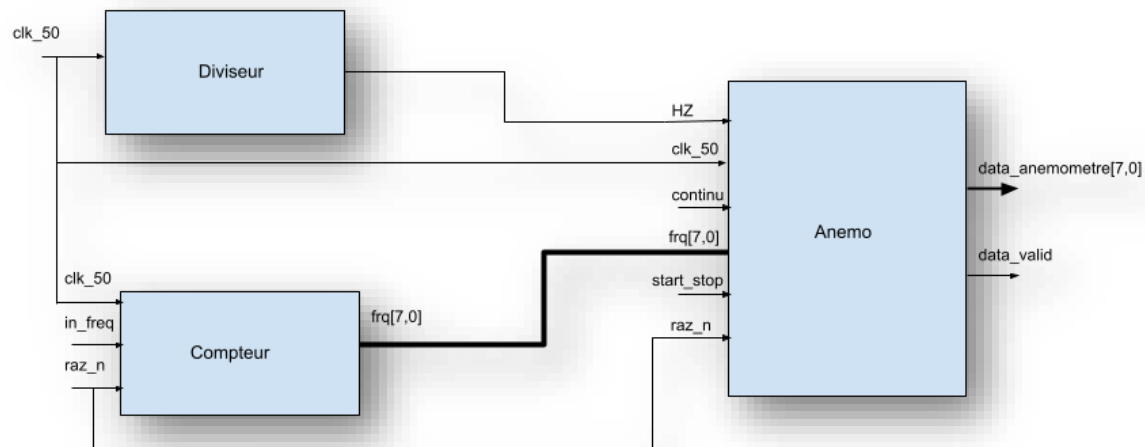


Figure 4: schéma fonctionnel de l'anémomètre

- **Diviseur :**

C'est un bloc qui permet de générer une horloge de 1 kHz afin de synchroniser les différents process de notre circuit.

- **Compteur :**

C'est un bloc qui permet de compter le nombre des fronts montant du signal numérique dans le but de mesurer la fréquence.

- **Anemo :**

C'est une fonction qui gère les modes de mesure de la fréquence in_freq, qui sera mémorisée dans une variable de sortie codé de 8 bits nommé data_anemometre, lorsqu'une mesure est valide, le circuit met sa sortie data_anemometre et data valid.

b. Implémentation :

La partie anémomètre est constituée de plusieurs blocs logiques interconnectés entre autres. Cet ensemble de blocs nous donne en sortie les mesures souhaitées. Dans la figure suivante, on retrouve un schéma BDF (Block Design File) qui représente le système.

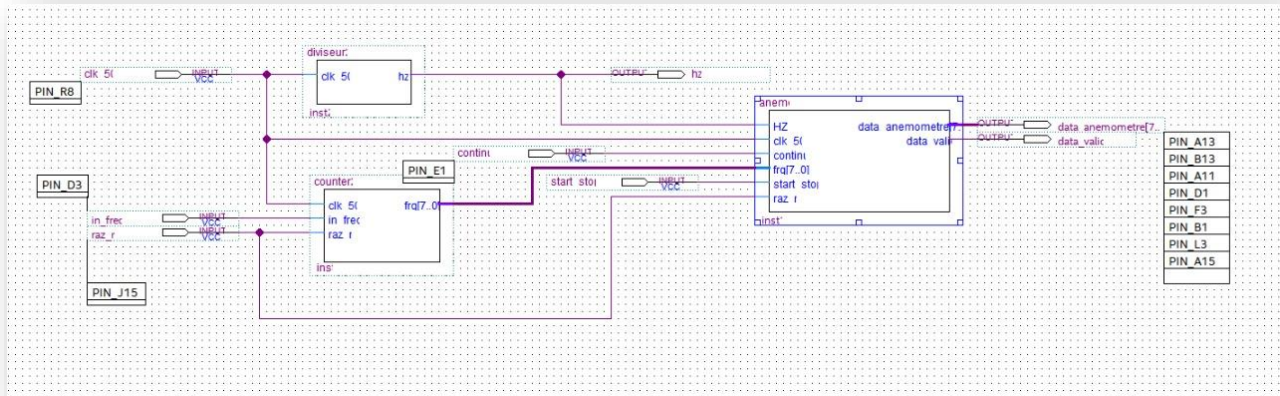


Figure 5: BSF de l'anémomètre

c. Spécification du module gestion_anemometre :

Entrées:

| | | |
|--------|-------------------|---|
| 0 1 | CLK_50M | horloge 50MHz |
| 2 | RAZ_N | reset actif à 0 => initialise le circuit |
| 0 3 | N_FREQ_ANEMOMETRE | signal de fréquence variable de 0 à 250 HZ |
| 0 4 | CONTINU | si=0 mode monocoup, si=1 mode continu |
| 0 5 | START_STOP | en monocoup si=1 démarre une acquisition, si=0 -- remet à 0 le signal data_valid |

Sorties:

| | | |
|--------|-----------------|---|
| 0 4 | DATA_VALID | = 1 lorsqu' une mesure est valide (est remis à 0 quand start_stop passe à 0) |
| 0 5 | DATA_ANEMOMETRE | vitesse vent codée sur 8 bits |

d. Simulation :

Avant de manipuler le fonctionnement de notre système nous avons créé notre fichier de simulation (TestBench) sous format .vhd

Voici un extrait de code qui montre la procédure de simulation qu'on a fait, au premier temps on a fixé continu à 1 puis on a attendu 1700 ms afin de remettre a zéro notre signal (raz_n=0) puis on a changé les autres variables pour avoir une simulation complète.

```
49 |
50 | clk_process : process
51 | | begin
52 | |   clk_50 <= '0';
53 | |   wait for clk_period/2;
54 | |   clk_50 <= '1';
55 | |   wait for clk_period/2;
56 | | end process;
57 |
58 | frq_process : process
59 | | begin
60 | |   in_freq <= '0';
61 | |   wait for frq_period/2;
62 | |   in_freq <= '1';
63 | |   wait for frq_period/2;
64 | | end process;
65 |
66 | stim_proc : process
67 | | begin
68 | |   wait for 1700 ms;
69 | |   raz_n <= '0';
70 | |   wait for 100 ms;
71 | |   raz_n <= '1';
72 | |   continu <= '0';
73 | |   start_stop <= '1';
74 | |   wait for 2000 ms;
75 | |   start_stop <= '0';
76 | |   wait for 1600 ms;
77 | |   raz_n <= '0';
78 | |   wait for 100 ms;
79 | |   continu <= '1';
80 | |   wait;
81 | | end process;
```

Figure 6: clk, frq, stim process dans le testbench

Ensuite nous avons lancé une simulation sur le logiciel **ModelSim** en choisissant notre fichier test comme il est indiqué dans la figure suivante :

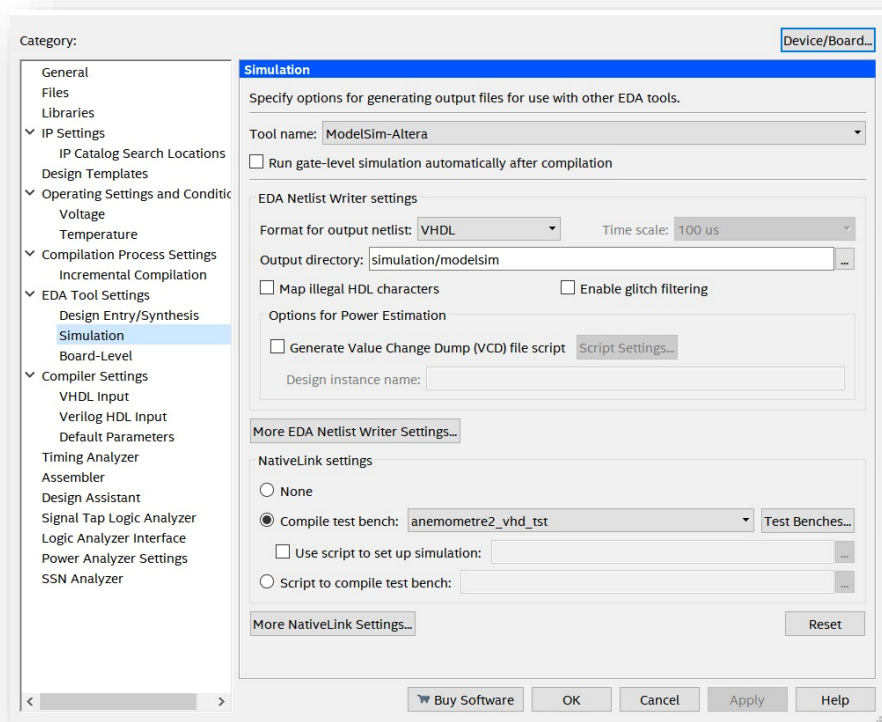


Figure 7: paramètre de simulation

Par conséquent, on a ce résultat de simulation :

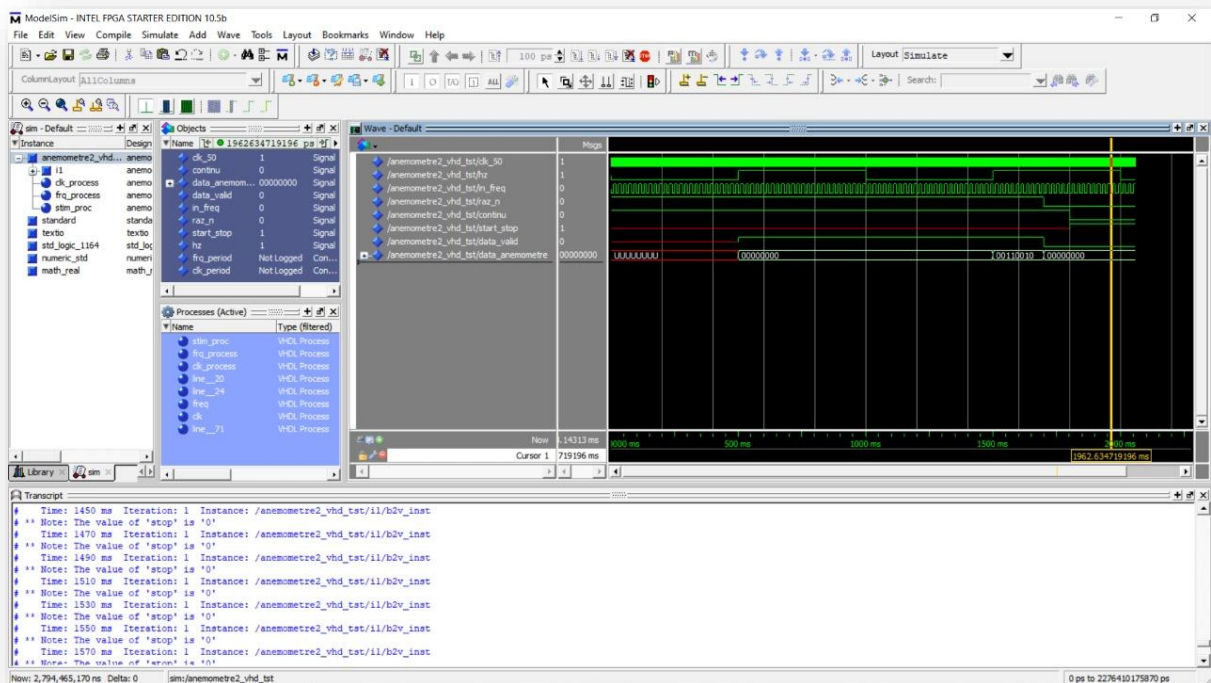


Figure 8: Modélisme interface

Pour le débogage on a décidé d'afficher la valeur de « stop » chaque 20ms.

```
report "The value of 'stop' is " & std_logic'image(stop);
```

Et voici ce qu'on voit dans la console de modélisme.

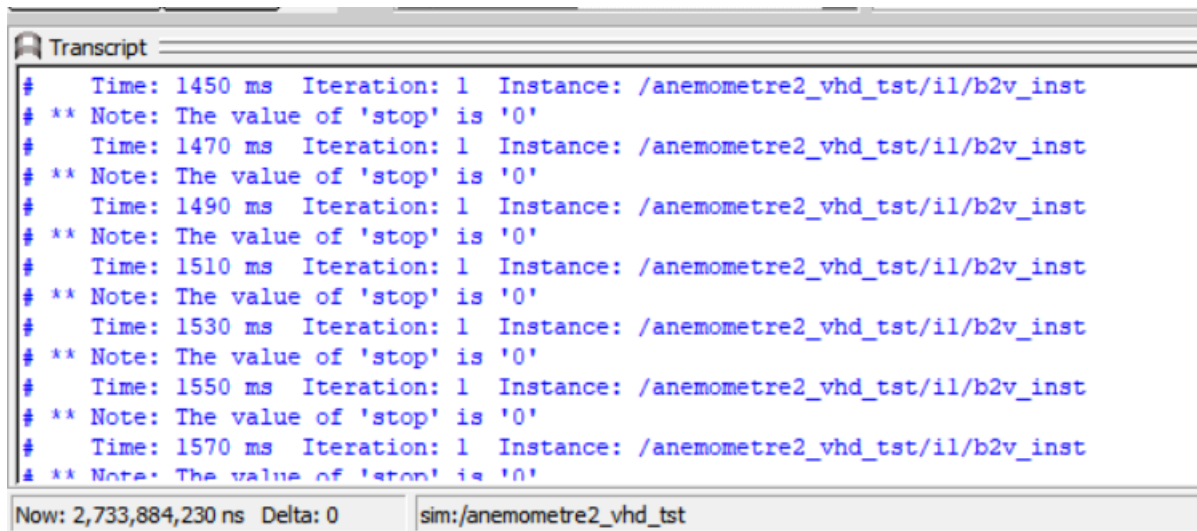


Figure 9: Information dans la Console de modélisme

Pour valider le fonctionnement de notre code VHDL, nous avons utilisé un GBF pour produire des signaux carrés avec certains paramètres afin de faire une validation de notre fonction “gestion anémomètre”. Nous avons visualisé notre signal d’entrée qui varie de 0 à 250 Hz (on a choisi 90hz et un rapport cyclique de 50% dans ce cas) sur l’oscilloscope. Et dont la correspondance de la vitesse est de 0 à 250 Km/h qui s’affiche sur les leds de notre carte.

Le code binaire dans la sortie de la carte c’est 01011010 qui égale à 90 en binaire, donc notre fonction marche bien.

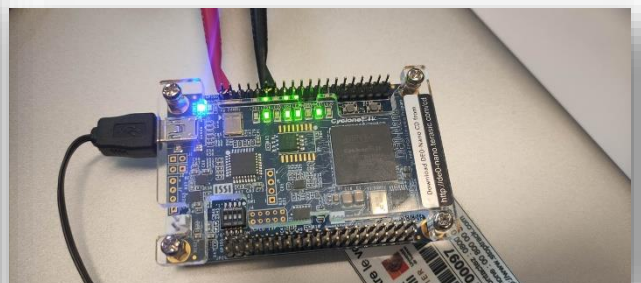


Figure 10 : signal carré de 90Hz générer par le GBF et la sortie de la carte DE0

e. Conception du SOPC

L'interface Avalon dispose de 2 registres tels que décrits ci-dessous :

| Registre | Adresse | Type | Bits concernés |
|----------|---------|------|-------------------------------------|
| Config | 0 | R/W | b2=Start/Stop, b1=continu, b0=raz_n |
| Code | 1 (4) | R/W | b9=valid, b7...b0= data_anemometre |

Afin de télécharger le circuit sur la carte DE0 nous avons utilisé platform designer afin de créer le microprocesseur et les éléments périphériques en intégrant l'anémomètre avec interface Avalon. On a ajouté le code VHDL de l'interface Avalon dans platform designer à côté d'un CPU, on-chip memory, Jtag et qlq PIO.

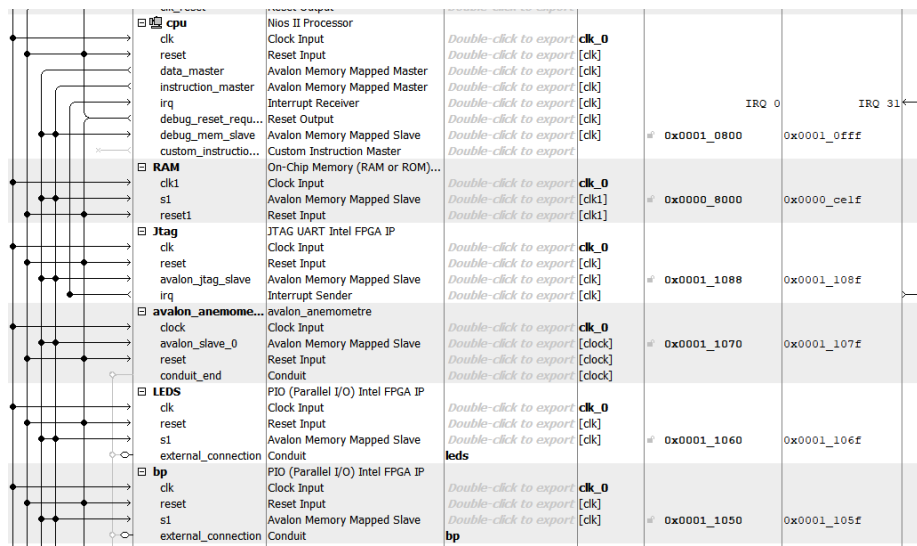


Figure 11: Les composants dans platform designer

Après génération HDL on a obtenu les BSF indiqué dans la figure ci-dessous.

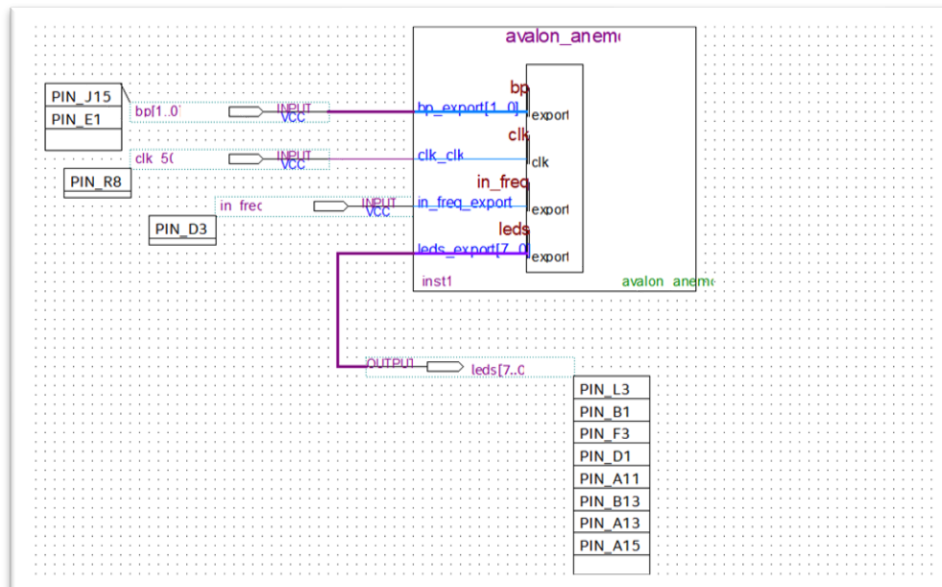


Figure 12: Le block généré par platform designer

Ensuite on a écrit le code en C pour faire fonctionner tout le système et voir la sortie data_anemometre et data_valid.

```
#define leds (unsigned int *) LEDS_BASE
#define button (unsigned char *) BP_BASE
|
#define config (unsigned int *) (AVALON_ANEMOMETRE_0_BASE)
#define code (unsigned int *) (AVALON_ANEMOMETRE_0_BASE + 4) //01

unsigned int a,b,c,d;

int main()
{
    alt_putstr("Hello from Nios II!\n");

    *config = 0x0003;

    printf("config = %d\n", *config);

    /* Event loop never exits. */
    while (1){

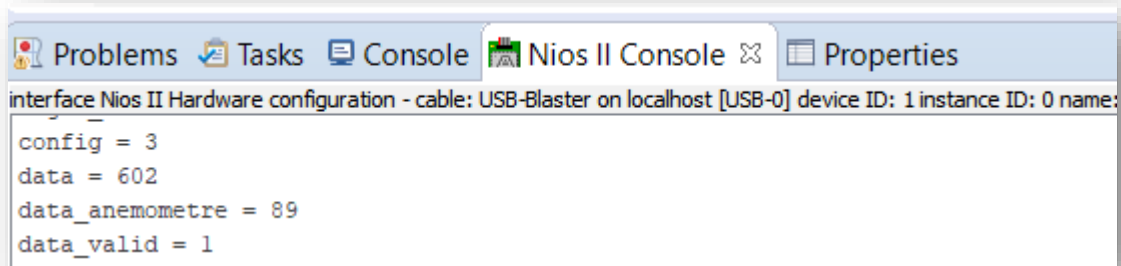
        *config = 0x0003;
        printf("inside while \n");
        // printf("config = %d\n", *config);
        printf("data = %d\n", *code);

        *leds = *code - 512;

        printf("data_anemometre = %d\n", *leds);
        if (*leds < 512) {
            printf("data_valid = %d\n", 0);
        }
        else printf("data_valid = %d\n", 1);
    }
}
```

Figure 13 : Code en C

On peut voir les résultats de notre système dans la figure suivant :



```
interface Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name:  
config = 3  
data = 602  
data_anemometre = 89  
data_valid = 1
```

Figure 14: la réponse de la carte après le téléchargement de code en C

2.2 Gestion du vérin :

a. Fonctionnement du vérin :

Le circuit de gestion du vérin est constitué de quatre fonctions principales :

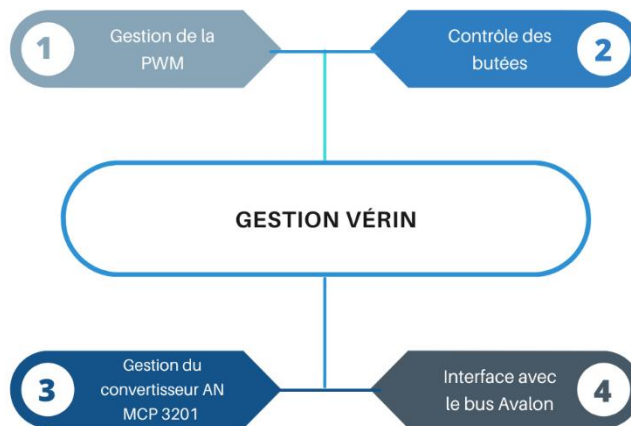


Figure 15: les fonctions principales du circuit gestion de vérin

b. Analyse fonctionnelle

Pour répondre bien aux exigences de notre circuit à concevoir, nous avons réalisé la description fonctionnelle suivant :

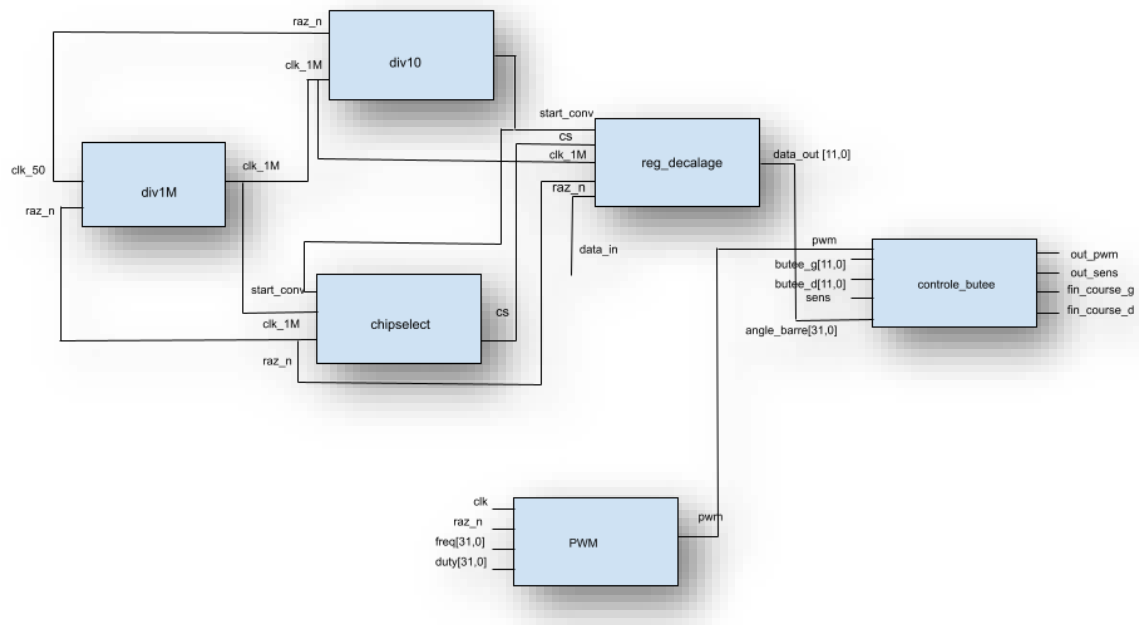


Figure 16: schéma fonctionnel du vérin

- **Div1M** : C'est un bloc qui permet de générer une horloge de 1 Mhz à partir de 50 Mhz, cette horloge est nécessaire pour le fonctionnement de la MCP3201 et aussi pour les autres blocs.
- **Div10** : C'est un bloc qui permet de générer une horloge de 10 Hz à partir de 50 Mhz, cette horloge est nécessaire pour la génération périodique (toutes les 100ms) du signal « start_conv ».
- **Chip_select** : ce bloc génère un signal CS qui est obligé pour le fonctionnement de la MCP3201.
- **Reg_decalage** : registre à décalage pour récupérer la donnée du convertisseur mcp3210, cette donnée (angle barre) va être injecter dans le bloc suivant.
- **Contrôle_butee** : ce bloc permet de mettre le signal PWM à 0 si angle barre se situe en dehors des butées butee_g et butee_d et selon le sens de rotation du moteur et il génère aussi les signaux fin_course_g et fin_course_d.

c. Implémentation :

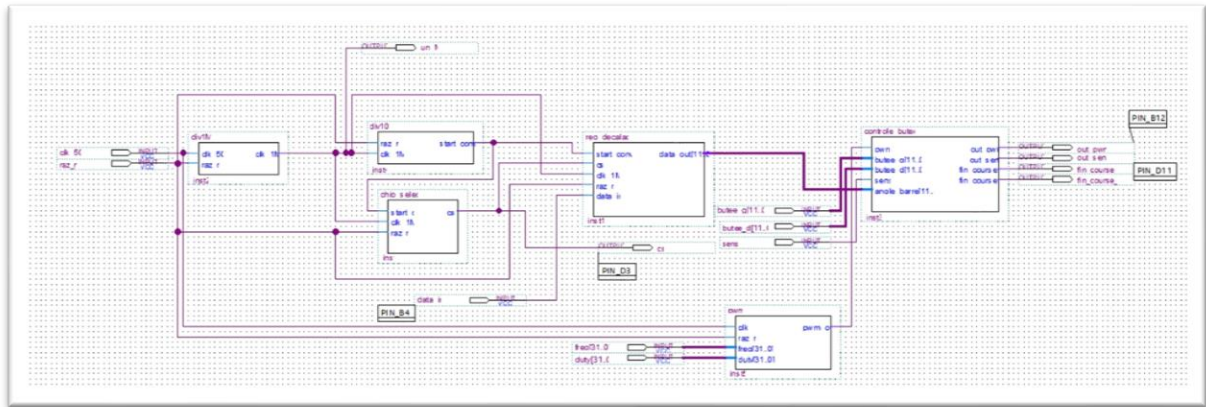


Figure 17: BSF du système de gestion de vérin

d. L'analyseur logique Signal Tap

L'analyseur logique Signal Tap capture et affiche le comportement du signal en temps réel afin de déboguer le comportement des signaux internes pendant le fonctionnement normal de système, sans nécessiter de broches d'E/S supplémentaires ou d'équipement de laboratoire externe.

L'interface graphique de l'analyseur logique Signal Tap nous aide à définir et à modifier rapidement la configuration du signal Signal Tap et à afficher les signaux capturés pendant l'analyse, démarrer et arrêter l'analyse, ainsi que d'afficher et enregistrer les données du signal. On a choisi de faire un trigger avec le start_conv puisque on veut voir le signal juste dans le cas où start_conv est en falling_edge.

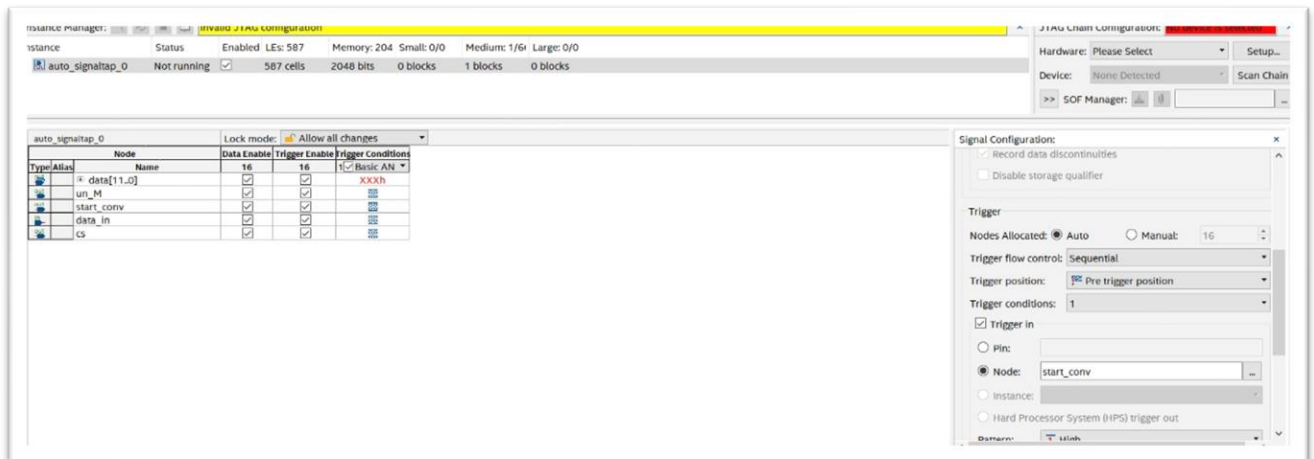


Figure 18: interface de l'analyseur logique

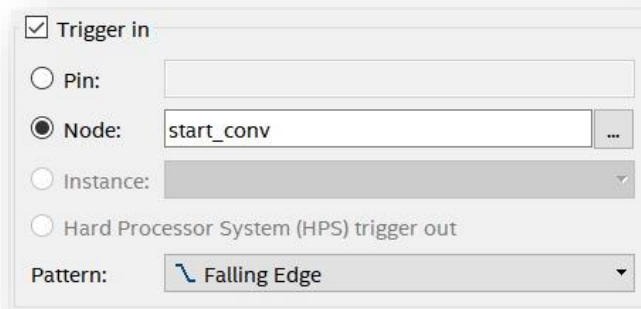


Figure 19: les paramètres du trigger in

Comme indiqué ci-dessous, lorsque CS est bas, nous remarquons qu'il y a 15 fronts d'horloge. Ce qu'est adéquat avec la datasheet de mcp3210.

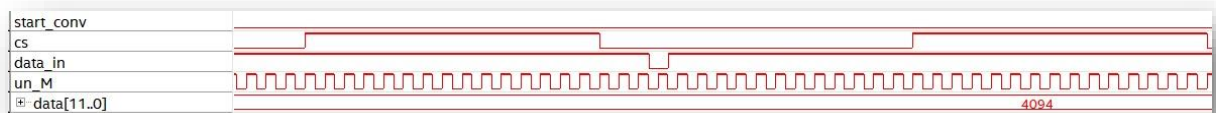


Figure 20: Nombre des fronts d'horloge quand cs est nul

Lorsque le potentiomètre est tourné au minimum. Le signal Tap Analyzer affiche 0 donc le résultat est correct, et la même chose quand le potentiomètre est tourné au maximum, on trouve la valeur 4094.

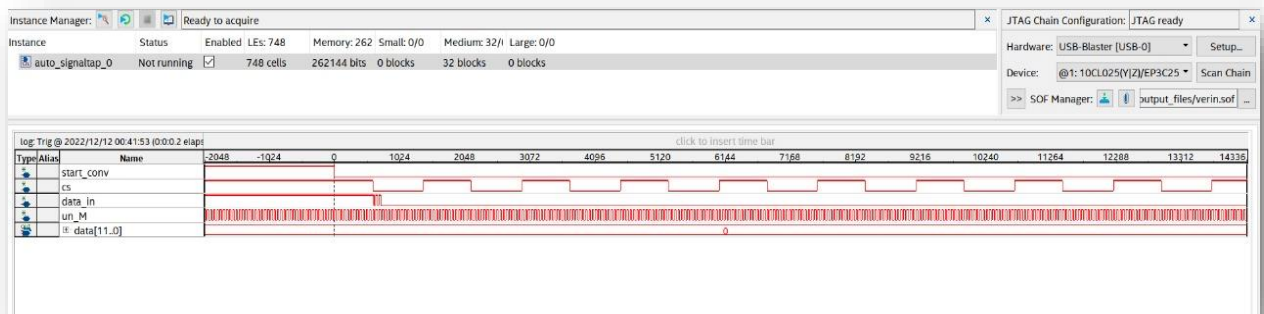


Figure 21: Valeur minimal de l'angle de barre

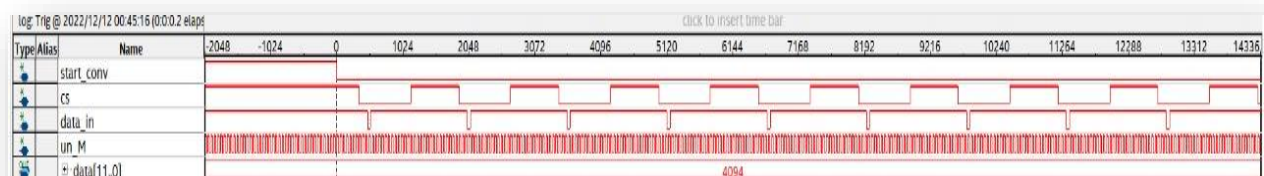


Figure 22: Valeur maximale de l'angle de barre

La valeur de la sortie du registre à décalage pour une entrée arbitraire.

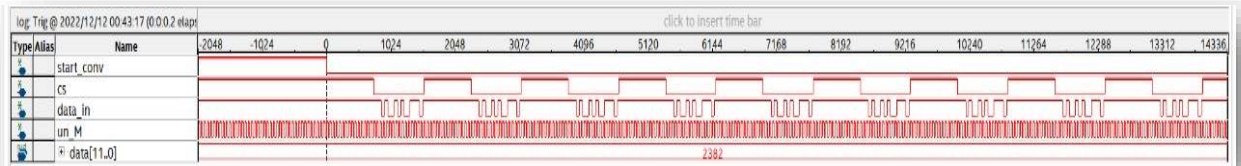


Figure 23: Valeur arbitraire de l'angle de barre

D'après les résultats du signal Tap Analyzer, nous pouvons dire que notre système fonctionne correctement.

e. Conception du SOPC

En utilisant platform designer, on assemble notre composant avec une ram, un cpu et le bus Avalon, pour utiliser le processeur et développer en soft notre système avec le langage C.

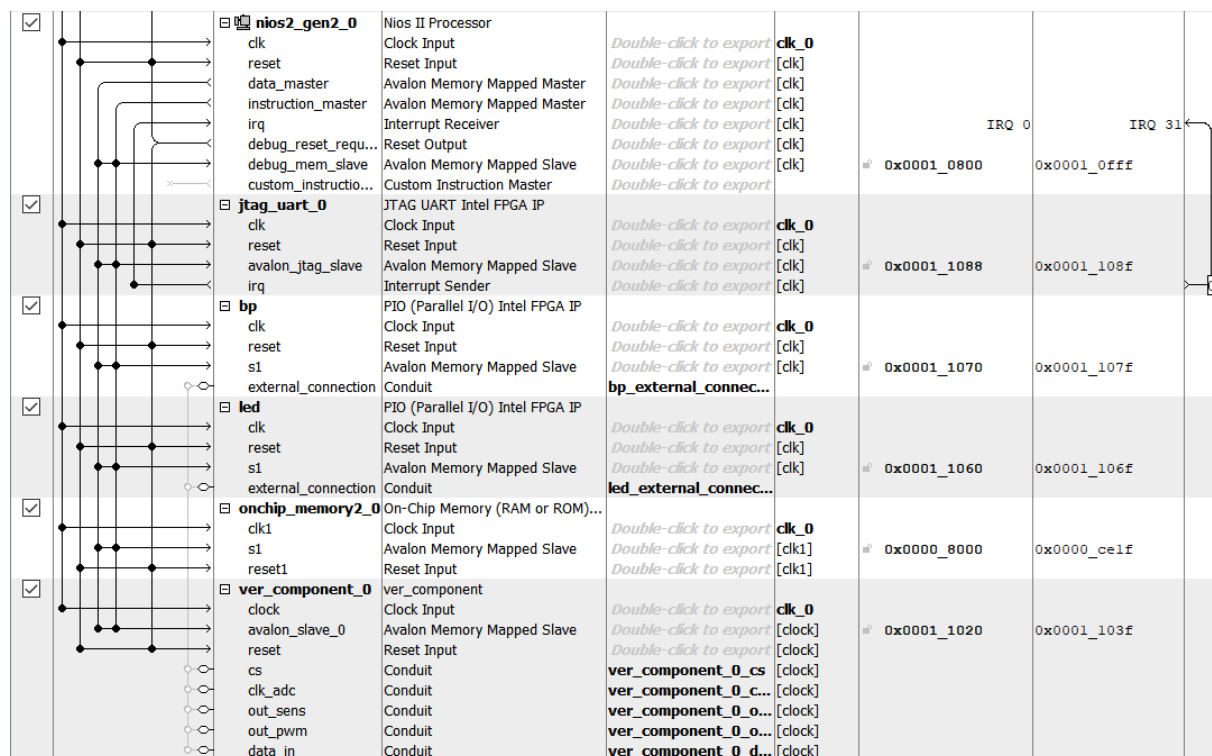


Figure 24: composant de l'interface Avalon du vérin dans platform designer

Après génération HDL on a obtenu les BSF indiqué dans la figure ci-dessous.

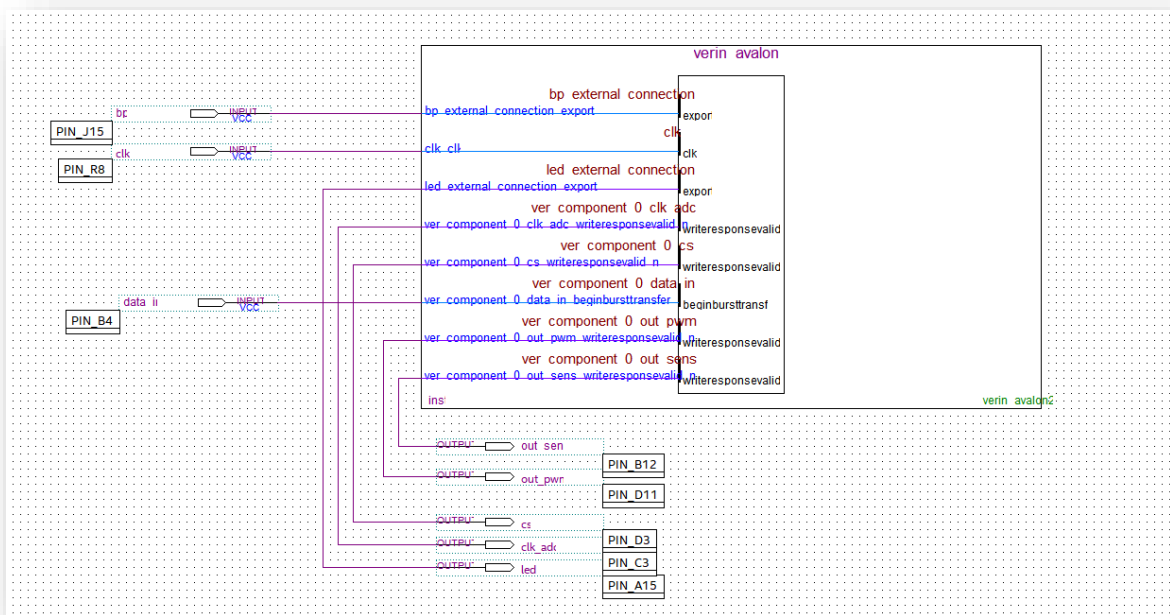


Figure 25: Le block généré par platform designer de l'interface avalon du vérin

L'interface Avalon dispose de 6 registres tels que décrits ci-dessous :

```
#define freq (int *)VER_COMPONENT_0_BASE
#define duty (int *) (VER_COMPONENT_0_BASE + 4)
#define butee_g (int *) (VER_COMPONENT_0_BASE + 8)
#define butee_d (int *) (VER_COMPONENT_0_BASE + 12)
#define config (int *) (VER_COMPONENT_0_BASE + 16)
#define angle_barre (int *) (VER_COMPONENT_0_BASE + 20)
```

Figure 26: les registres de l'interface Avalon

- **Freq** : fixe la fréquence de la PWM moteur (exemple : si freq = 2000 alors la fréquence de la PWM = $\text{clk}/2000$ soit 25KHz si $\text{clk}=50\text{MHz}$)
- **Duty** : fixe le rapport cyclique
- **Butee_g** et **butee_d** : réglables de 0 à 4095 elles fixent les valeurs limites que le vérin (angle_barre) ne doit pas dépasser. En dehors de ces valeurs, le moteur est automatiquement coupé. Ces valeurs sont mises à 0 par défaut et doivent être initialisées au démarrage du système.
- **Config** : registre de configuration du circuit. La description du rôle de chaque bit est donnée dans le tableau suivant :

| | | | |
|---------------|---------------|------------|--|
| config | 4 (16) | R/W | b0 :raz_n (à 0=reset circuit) |
| | | R/W | b1 :enable_pwm (1= pwm actif) |
| | | R/W | b2: sens rotation (0=gauche) |
| | | R | b3:fin_course_d (=1 si fin course_d) |
| | | R | b4: fin_course_g (=1 si fin course_g) |

- **Angle_barre** : il donne la valeur de l'angle de barre codée sur 12 bits (résultat de la conversion analogique numérique) ...

On peut voir l'angle de la barre, les butées ainsi que le registre de configuration et Freq et Duty dans la console de NIOS comme on voit sur la figure suivante.

```

interface Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtag_uart_0.jtag
config= 1
angle_barre= 698
code_fonction= 1
freq= 2000
duty= 1500
butee_d= 1320
butee_g= 410
config= 1
angle_barre= 698
  
```

Figure 27: la valeur de l'angle de barre et les autres variables

Pour s'assurer que tout marche bien on peut voir le mémoire du système dans le mode debugger.

Pour nous on voit bien que les valeurs dans la mémoire changent après chaque affectation en C, 920 c'est l'angle de la barre après qu'on a tourné le potentiomètre.

| Monitors | | | | | |
|--|-------|-------|-------|---------|--|
| 0x00011020: 0x11020 <Unsigned Integer> | | | | | |
| Address | 0 - 3 | 4 - 7 | 8 - B | C - F | |
| 00011020 | 2000 | 1500 | 410 | 1320 | |
| 00011030 | 1 | 920 | 0 | 0 | |
| 00011040 | 0 | 0 | 0 | 0 | |
| 00011050 | 0 | 0 | 0 | 0 | |
| 00011060 | 1 | 0 | 0 | 0 | |
| 00011070 | 1 | 0 | 0 | 0 | |
| 00011080 | 0 | 0 | 9216 | 3154944 | |
| 00011090 | 0 | 0 | 0 | 0 | |
| 000110A0 | 0 | 0 | 0 | 0 | |
| 000110B0 | 0 | 0 | 0 | 0 | |

Figure 28: mémoire de notre système

Conclusion

Les projets décrits dans ce document auxquels nous avons participé ce semestre nous ont permis de développer nos compétences en VHDL, notamment en électronique numérique. De plus, ce bureau d'études nous a permis de voir la différence de la programmation en VHDL, qui permettait de produire des descriptions matérielles destinées à représenter et concevoir le comportement et l'architecture des systèmes électroniques numériques. Lors de la conception de certaines fonctions, nous nous sommes retrouvés limités par le matériel, et nous avons dû bien comprendre chaque composant numérique afin de faire fonctionner certains blocs fonctionnels. Nous pouvons également ajouter que les outils de simulation dont disposent les logiciels de programmation VHDL sont des outils très puissants pour la mise en œuvre de systèmes électroniques grâce à la conception assistée par ordinateur.