BUREAU D'ETUDE BE-VHDL

M2 SME

Synthèse et Mise en Œuvre des Systèmes
Pilote de barre franche

Auteurs :

Armou Oumayma

EL BATCHY Wafaa

SARI Mouad

Encadrant :

CARVALHO MENDES Pedro

Table de matières

Listes des figures

Introduction

Dans le contexte de notre formation, systèmes et microsystèmes embarque, nous allons du travailler sur un projet est la conception d'un système pour la gestion automatique d'un voilier de barre franche y compris la mise en intégration Hardware/Software qui répond aux exigences mentionnées dans le cahier de charge.

Le but de ce projet est de prendre en main le langage VHDL et le logiciel Quartus afin de maitriser ses fondamentaux et mettre en œuvre les solutions sur les cartes FPGA Altera.

Pour réaliser cela, on commence tout d'abord par l'identification des besoins de système. Ensuite, en passant à la décomposition de chaque fonction sous forme des blocs. Cette étape va faciliter leur implémentation en code VHDL et en code C. Finalement, pour vérifier le bon fonctionnement, on va lancer des simulations sur la maquette a l'aide de logiciel Quartus et puis on fait un interfaçage avec NIOS II et Avalon.

1. Présentation du projet

Notre travail consiste à réaliser le pilote automatique de voilier a barre franche. C'est une combinaison d'une électronique très sophistiquée gérée par un logiciel avancé et d'une puissante mécanique, ils atteignent un niveau de performance élevé dans la précision de barre dans une large gamme de conditions de navigation.

Le système est constitué des éléments suivants :

- ✓ **Anémomètre :** un instrument qui permet de calculer la direction et la vitesse du vent.
- ✓ **Compas :** est une boussole à compensation d'inclinaison qui permet de faire l'acquisition des données sur l'angle d'inclinaison.
- ✓ **Les boutons, Leds et buzzer :** est une interface de communication entre l'operateur et le voilier qui permet de monitorer et configurer en temps réel les différents paramètres.
- ✓ **Vérin :** permet de contrôler la barre en fonction de sens souhaitée.

La figure ci-dessous présente le système en globalité, les missions qu'on doit réaliser, les fonctions nécessaires à l'implémentation de notre barre franche.

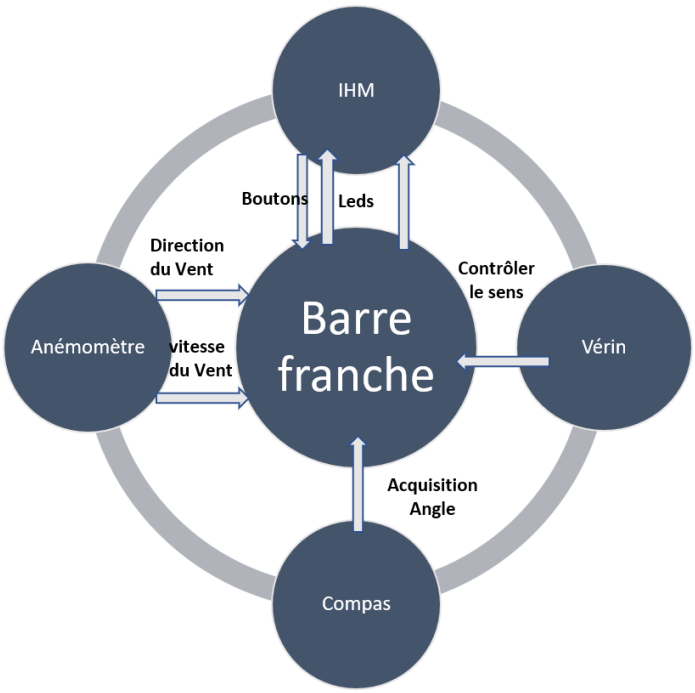

Figure 1: Système Barre Franche

2. Conception Matérielle

Vous trouvez ci-dessous les types des cartes qu'on a pu travaille durant le TP et le projet BE :

2.1. Carte DE0 – Nano

La carte DE0-Nano présente une plate-forme de développement FPGA compacte adaptée au prototypage de conceptions de circuits telles que les robots et les projets "portables". La carte est conçue pour être utilisée dans l'implémentation la plus simple possible, ciblant le circuit Cyclone IV avec un maximum de 22 320 éléments logiques. Les avantages de la carte DE0-Nano incluent sa taille et son poids, ainsi que sa capacité à être reconfigurée sans matériel superflu. Ces caractéristiques la distinguent des autres cartes de développement à usage général. Tous les fichiers de conception du microcontrôleur Propeller à 8 noyaux de Parallax sont en code source libre.

Figure 2: Carte DE0

2.2. Carte DE2

La carte DE2 (Develoment and Education Board), construite autour d'un FPGA Altera Cyclone II 2C35, permet de nombreuses applications, grâce aux circuits annexes implantés (mémoires, afficheurs, CODEC, décodeur vidéo, ports Ethernet, USB etc…)

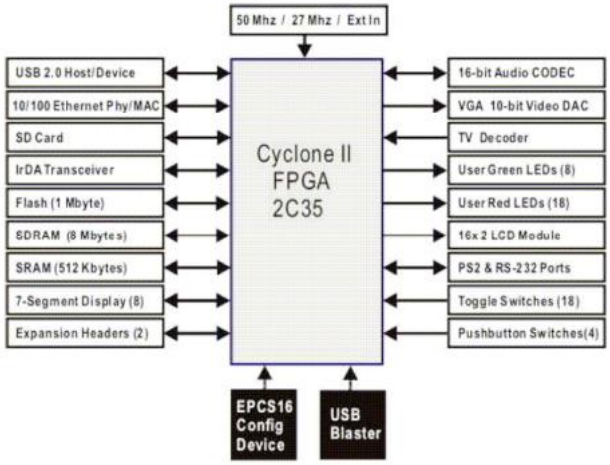

Figure 3: Schéma fonctionnel d'Altera

De nombreux connecteurs permettent l'interfaçage avec l'extérieur :

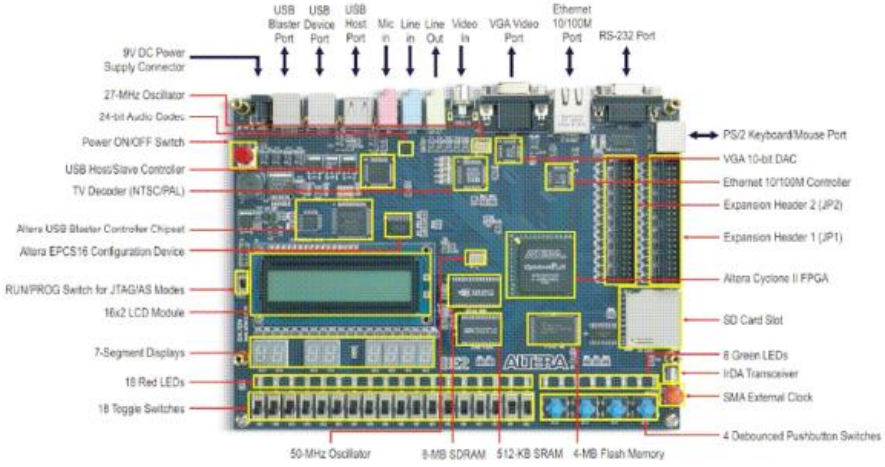

Figure 4: DE2

3. Fonction simple "Compas"

3.1. Présentation de Compas

Le module compas est une boussole à compensation d'inclinaison qui permet de faire l'acquisition des données sur l'angle d'inclinaison. Ce module fournie une plage de mesure de 0° à 359° avec une précision de 100 us/°. Pour mieux illustrer le fonctionnement notre module, notre compas est initialement positionné vers le nord qui est l'origine et le réfèrent de notre système plus augmente le degré d'inclinaison plus en ce dirige dans le sens suivant : Nord → Est →Sud → Ouest.

Le module Compas utilise comme entrée un signal PWM avec un état haut qui varie de 1 ms à 36 ms et avec un état bas qui est fixe à 65 ms, chaque 1 ms et équivalent à 10°.

Figure 5: Temps de fonctionnement de compas

3.2. Analyse Fonctionnelle

Afin de bien réussir à réaliser notre module, nous avons procédé à faire une décomposition de notre sous-système en des différents blocs internes, comme décrié dans la description fonctionnelle ci-dessous :

- **Diviseur 10 kHz :** Ce bloc est dédié pour la génération de la fréquence de 10 kHz qui est équivalent à un degré, ce dernier va nous aider à faire la comparaison du signal de 10 kHz avec L'état haut de notre signal d'entrée et comme ça en va savoir le nombre de degrés dans chaque période de la PWM.
- **Diviseur 1 Hz :** ce bloc est dans le but de répondre aux exigences fixées dans le cahier des charges pour le mode continu qui sert à rafraichir la donnée tous les seconds.
- **Fusionner :** ce bloque à comme entré le signal "in_PWM" et le signal de "10 kHz" et comme sortie le "le_signal_compteur". Quand le signal in_PWM est en état haut, en recopie le signal 10 kHz dans le signal de sortie, au niveau de l'état bas en force signal de sortie à zéro.

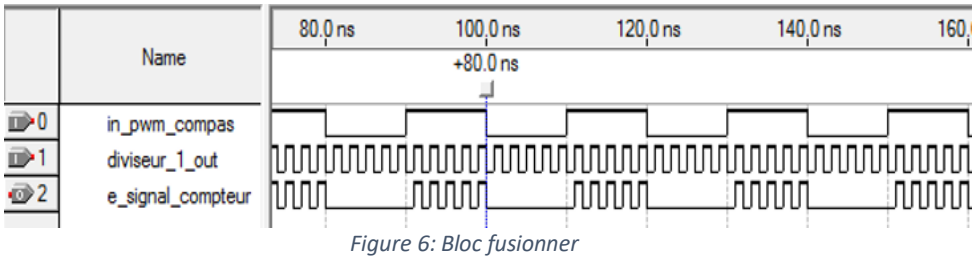

Figure 6: Bloc fusionner

- **Compteur_de_front :** Le signal issu du dernier bloc est injecté dans le bloc de compteur de front, car le nombre de fronts dans ce cas signifie aussi le nombre de degrés dans chaque information. La réutilisation du signal PWM dans ce bloc sert à définir quand est-ce que chaque période finie et avec cela, on n'est sûr que notre information est fiable. La sortie de notre signal est représentée sur 9 bits ($2^9 = 512 > 359$).

- **Traitement :** notre signal d'information est déjà construit dans la partie précédente dans d cette partie le but est de répondre aux exigences de cahier des charges tel que le mode de fonctionnement (monocoup - continue), la remise à zéro et finalement le star-stop ainsi de mettre le pin de sortie "Data-valid" à 1 quand la mesure est valide.

Vous trouvez ci-dessous le schéma global de fonctionnement de compas :

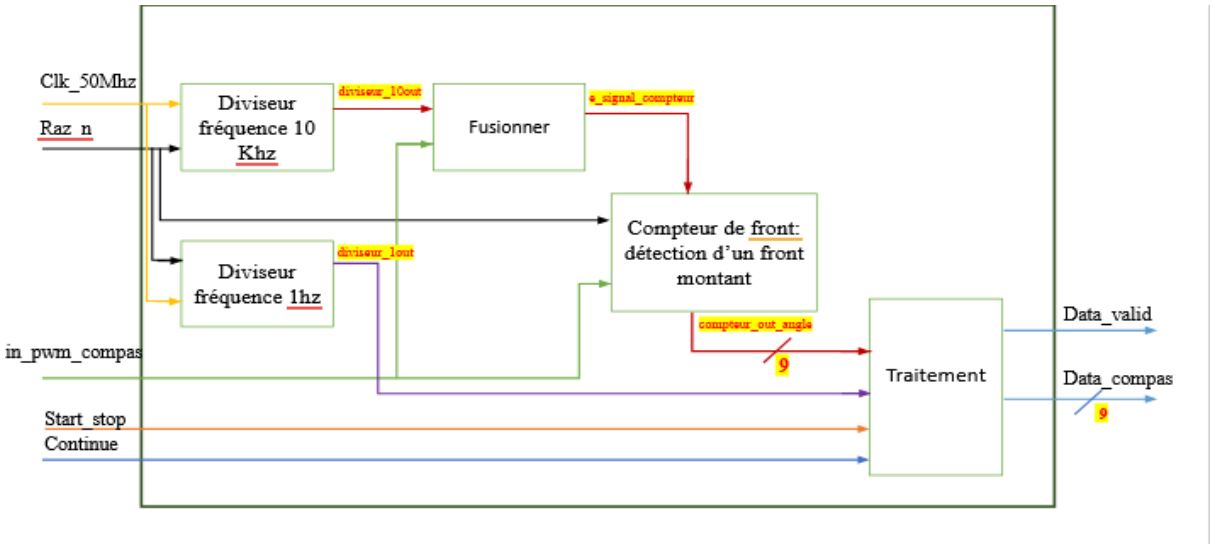

Figure 7: Bloc fonctionnel du Compas

3.3. Implémentation

Après avoir fait l'analyse fonctionnelle, nous avons basculé sur Quartus pour écrire les différents fichiers .vhd correspondant à chaque bloc, ces derniers vont servir, à générer le symbole de chaque bloc, le câblage du module est fait manuellement. Comme étape finale en génère le fichier gestion qui contient tous les composants et les connexions entre les blocs (PORTMAP).

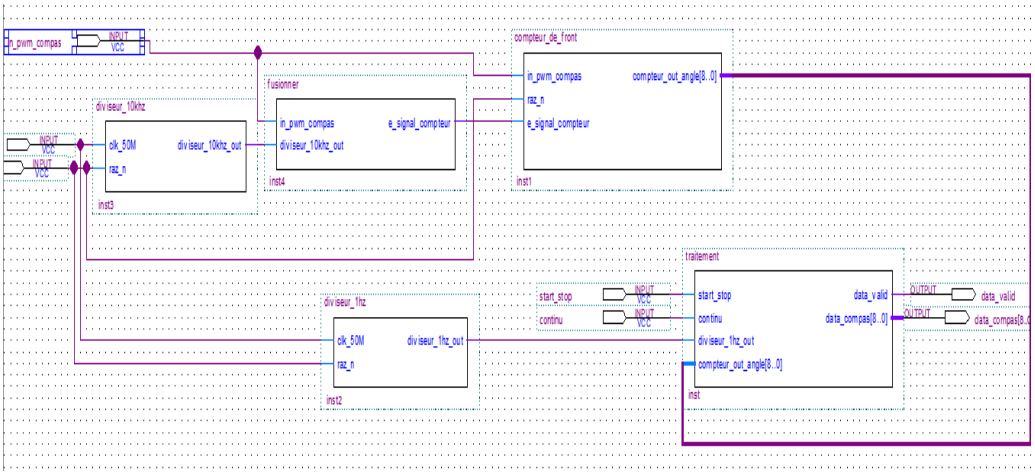

Figure 8:Schéma fonctionnel du compas

Avant de finir la partie implémentation, nous avons procédé à faire un “test bench “ est visualisé les résultats sur ModelSim, ce dernier va nous permettre de visualiser avec précision tous les états transit de nos systèmes ainsi que le comptage de valeurs.

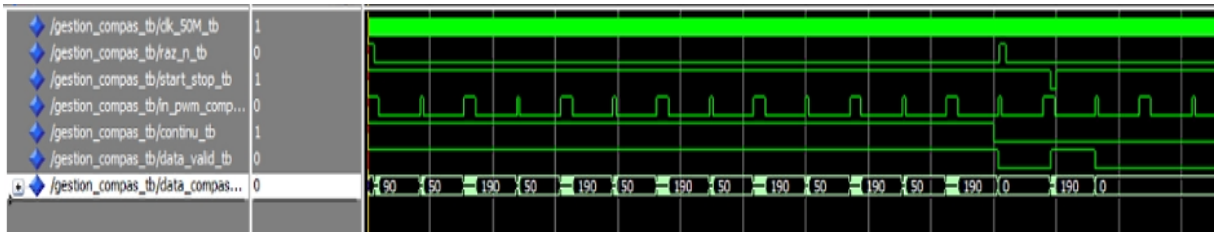

Figure 9:Compas sur ModelSim

Résultat :

On remarque sur les graphes deux valeurs d’angle (50° et 190°) qui représentent respectivement les impulsions fixaient par le PWM. Aussi en visualise que quand le signal continu descend à zéro en exécutant une seule acquisition et on s’arrête tout 'en mettent la data valid à 0 ;

Pour la simulation réelle et comme visualiser sur l’oscilloscope, nous avons mis en entrée un signal PWL avec un état haut de 10 ms et un état bas de 63 ms cette période est équivalente à 90° (EST).

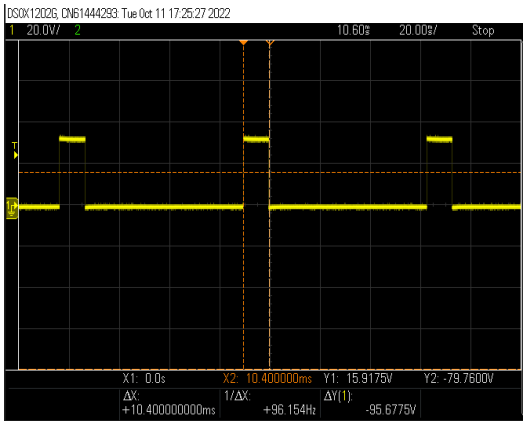

Figure 10: Visualisation sur le GPF

3.4. Conception du SOPC

Pour pouvoir réaliser la conception SOPC du Compas il y a deux étapes à respecter la première et de faire un fichier avalon.vhd ce dernier a comme rôle lié le module au bus avalon et de permettre la lecture et l'écriture sur ce bus aussi comme notre système général contient plusieurs modules donc le bus avalon sert à standardiser le transfert de données sur des registres de 32 bits.

```
signal start_stop, continu, in_pwm_compas, raz_n, data_valid : std_logic;
signal data_compas : std_logic_vector ( 8 downto 0);

BEGIN

process_write : process (clk, reset_n)
begin
if reset_n = '0' then
   raz_n <= '0';
   continu <= '0';
   start_stop <= '0';
   elsif clk'event and clk = '1' then
   if chipselect ='1' and write_n = '0' then
      if address = "00" then
         raz_n <= writedata(0);
         continu <= writedata(1);
         start_stop <= writedata(2);
      end if;
   end if;
end if;
end process;

-- lecture registres
process_Read : process(address, start_stop, continu, raz_n,data_compas,data_valid)
BEGIN
case address is
when "00" => readdata <= X"0000000"&"0"&start_stop&continu&raz_n;
when "10" => readdata <= X"00000"&"00"&data_valid&data_compas;
when others => readdata <= (others => '0');
end case;
END PROCESS process_Read ;

C1 : gestion_compas port map(raz_n, clk, start_stop, continu, in_pwm_compas, data_compas, data_valid);

END arch_compas_avalon ;
```

Figure 11: Code C du compas

La figure en haut montre les différents process existant dans le bus Avalon (écriture et lecture) aussi la condition sur la lecture par exemple de lire les pin Start_stop et continue et raz_n qu'on adresse et à 00 est ainsi de suite.
La deuxième étape et faite sur "Plateform designer" et elle consiste à créer le microprocesseur et les différents éléments dont on a besoin :

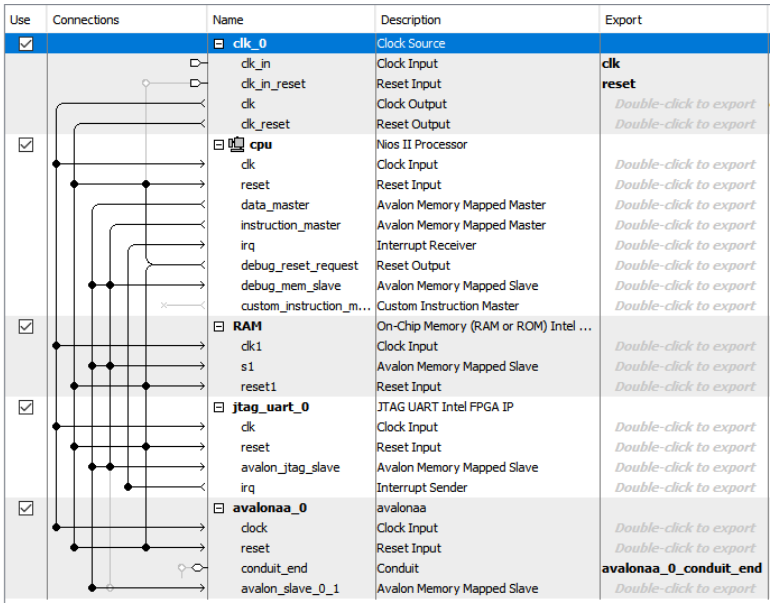

Figure 12: Création du Processeur sur Platform Designer

Ci-dessous le bloc qui montre clairement le bus Avalon avec ces différentes fonctionnalités :

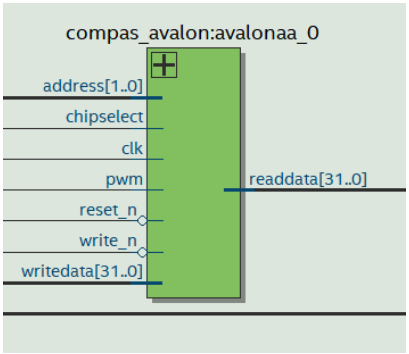

Figure 13: Bus Avalon du Compas

L'étape suivante est de créer un fichier nommé, TOP ce fichier a dans le rôle de gérer le bus Avalon et la gestion compas pour assurer le bon fonctionnement du code.

```
14    end entity top;
15
16       ARCHITECTURE rtl of top IS
17
18
19    component avalonaaaa is
20       port (
21          avalonaa_0_conduit_end_beginbursttransfer : in std_logic := 'X';
22          clk_clk                                   : in std_logic := 'X';
23          reset_reset_n                             : in std_logic := 'X'
24       );
25    end component avalonaaaa;
26
27
28
29       begin
30
31          u0 : component avalonaaaa
32       port map (
33          avalonaa_0_conduit_end_beginbursttransfer => pwm, -- avalonaa_0_
34          clk_clk                                   => clk_50M,
35          reset_reset_n                             => raz_n
36       );
37
38
39
40       end ARCHITECTURE;
41
```

Figure 14: Fichier Avalon

4. Fonction simple "Anémomètre"

4.1. Description de l'anémomètre

Le mot anémomètre vient du mot grec "anemos" signifiant vent et du suffixe "mètre" signifiant mesure. Un anémomètre est donc un appareil qui existe dans le domaine météorologique et qui sert à mesurer la vitesse du vent.

Le vent est le mouvement horizontal de l'air. Il se caractérise par sa force, sa direction et sa vitesse. La vitesse est mesurée en kilomètres par heure ou en mètres par seconde. Un anémomètre est un instrument utilisé pour effectuer ces mesures, prévoir le temps et prévenir les tempêtes.

Dans notre cas, l'anémomètre permet l'acquisition de la vitesse du vent dans une plage donnée [0…250km/h] et pour la suite la convertir en une fréquence [0..250Hz].

Figure 15:Anémomètre

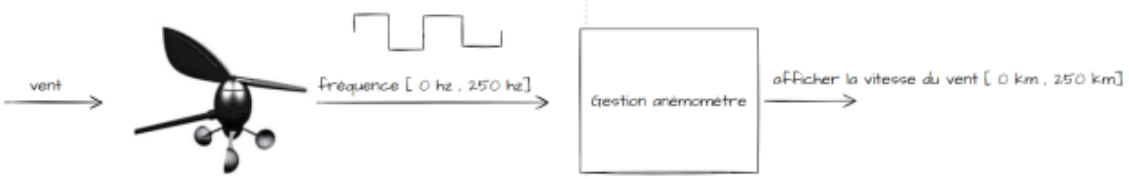

Figure 16: Mode de fonctionnement

4.2. Analyse Fonctionnelle

Pour réaliser cette fonction simple « Gestion Anémomètre », il faut répondre aux besoins et exigences. Mais avant de commencer à l'implémentation, nous allons par un schéma fonctionnel sous forme des blocs pour mieux cerner les fonctions principales suivant en respectant le cahier de charge :

- **Bloc de diviseur de fréquence :** C'est un bloc qui permet de générer une division de fréquence de 1Hz à partir de l'horloge interne de la carte FPGA de 50 MHz. Cette horloge fournit une acquisition toutes les secondes de la fréquence de l'anémomètre. Le code ci-dessous contribue à la réalisation de ce bloc.
- **Détecteur de front :** Ce bloc permet de détecter les fronts montants de la fréquence d'anémomètre afin de convertir la fréquence de 1Hz qui est toujours ajuster avec l'horloge interne en une donnée pourque le compteur puisse s'activer.
- **Counter :** Ce bloc va s'activer lors de la détection des fronts montants. Il va permettre de compter nombre de ces derniers toutes les secondes pour faire sortir la valeur de la fréquence en 8 bits.
- **Registre :** Ce bloc permet de stocker et mémoriser la valeur de la fréquence venant du compteur et par la suite l'initialiser le registre pour autoriser la réception de la nouvelle donnée sans écraser l'ancienne. Ainsi que ce registre rendre possible de visualiser les données puisque le calcul se fait à l'échelle d'une seconde.

Vous trouvez ci-dessous le schéma fonctionnel global de chaque bloc ainsi que le temps de fonctionnement de la fonction simple Gestion Anémomètre :

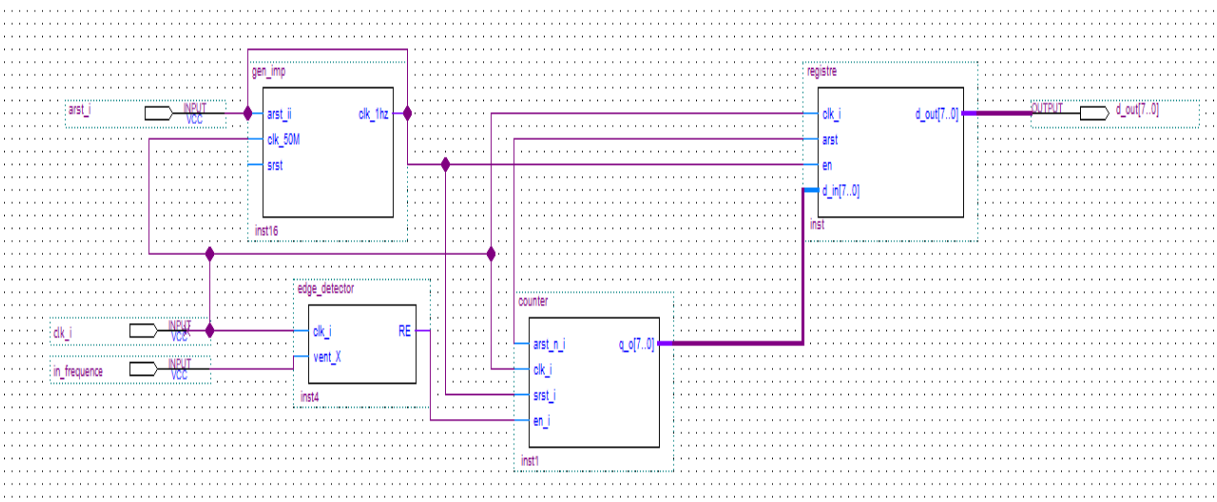

Figure 17: Schéma fonctionnel de l'anémomètre

4.3. Simulation

Apres la réalisation du schéma, on a passe à l'implémentation des blocs en code VHDL sur Quartus. Pour valider le bon fonctionnement de l'anémomètre nous avons procédé au lancement du ModelSim comme il est indiqué dans la figure ci-dessous :

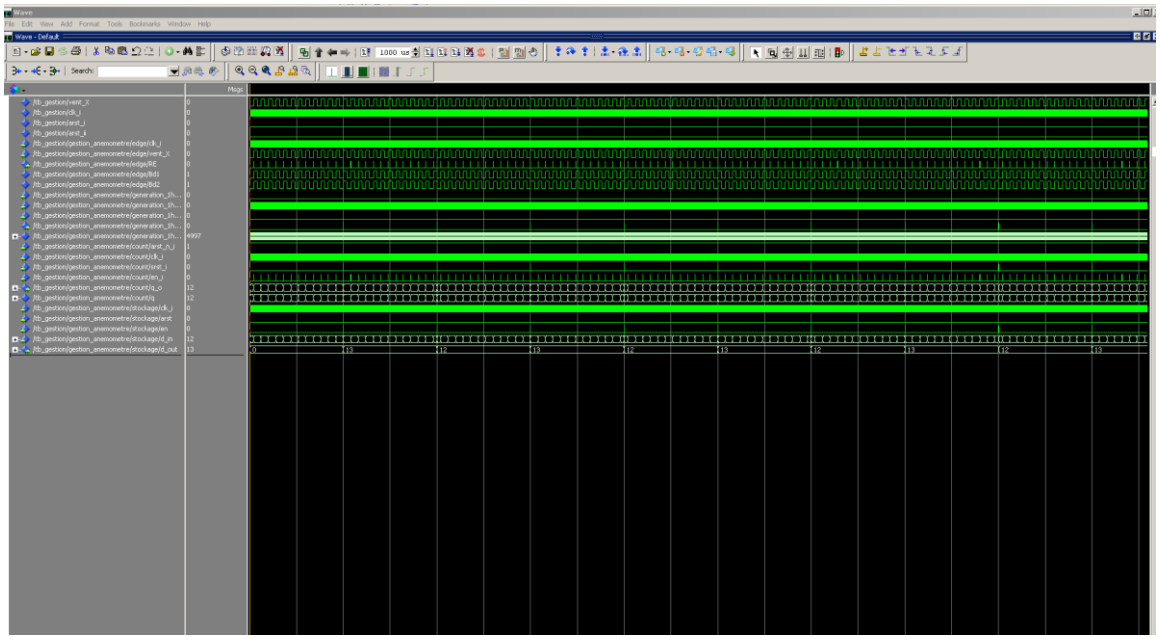

Figure 18: Simulation sur Quartus

On fait un zoom sur l'acquisition de la fréquence du vent on remarque qu'on obtient des valeurs en 12 et 13.

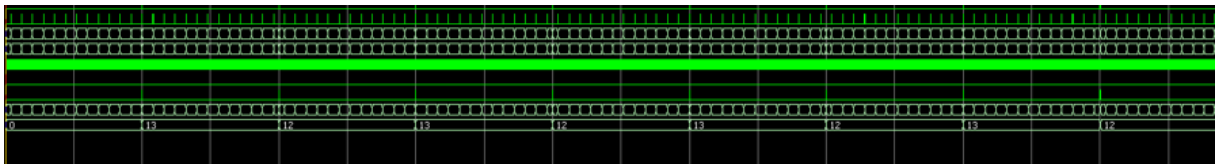

Figure 19: Fréquence du vent

4.4. Test et validation sur la carte FPGA

Pour s'assurer que l'anémomètre fonctionne bien, on va procéder à l'utilisation de GBF qui va nous permettre de faire varier les valeurs de fréquence et cette dernière on va la visualiser sur console de Quartus et sur la carte FPGA. La figure ci-dessous représente les résultats.

Pour faire cela on doit relier le canal 1 avec la broche GPIO_1 en sortie sans oubliant bien la masse sur la broche 6.

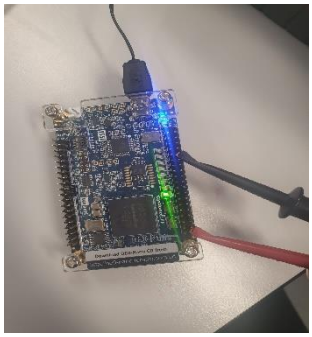

Figure 21: Simulation sur la maquette

Figure 20: Visualisation sur GBF

Node Name	Direction	Location	I/O Bank
arst_i	Input	PIN_J15	5
clk_i	Input	PIN_R8	3
d_out[7]	Output	PIN_L3	2
d_out[6]	Output	PIN_B1	1
d_out[5]	Output	PIN_F3	1
d_out[4]	Output	PIN_D1	1
d_out[3]	Output	PIN_A11	7
d_out[2]	Output	PIN_B13	7
d_out[1]	Output	PIN_A13	7
d_out[0]	Output	PIN_A15	7
vent_X	Input	PIN_B12	7

Figure 22: Pin Planner

5. Fonction Complexe "Gestion du Vérin"

5.1. Présentation du Vérin

Cette fonction sert à créer un mouvement mécanique de la barre du vérin en fonction du sens souhaité (soit à gauche, soit à droite), et en fonction de l'angle de barre il faut commander le moteur (signal PWM ici) du vérin pour contrôler le déplacement de cette barre.

Figure 23: Vérin

5.2. Analyse Fonctionnelle

Le schéma-bloc ci-dessous défini les différentes qui constituent la fonction “Vérin” ainsi qu'avec l’interface Bus Avalon :

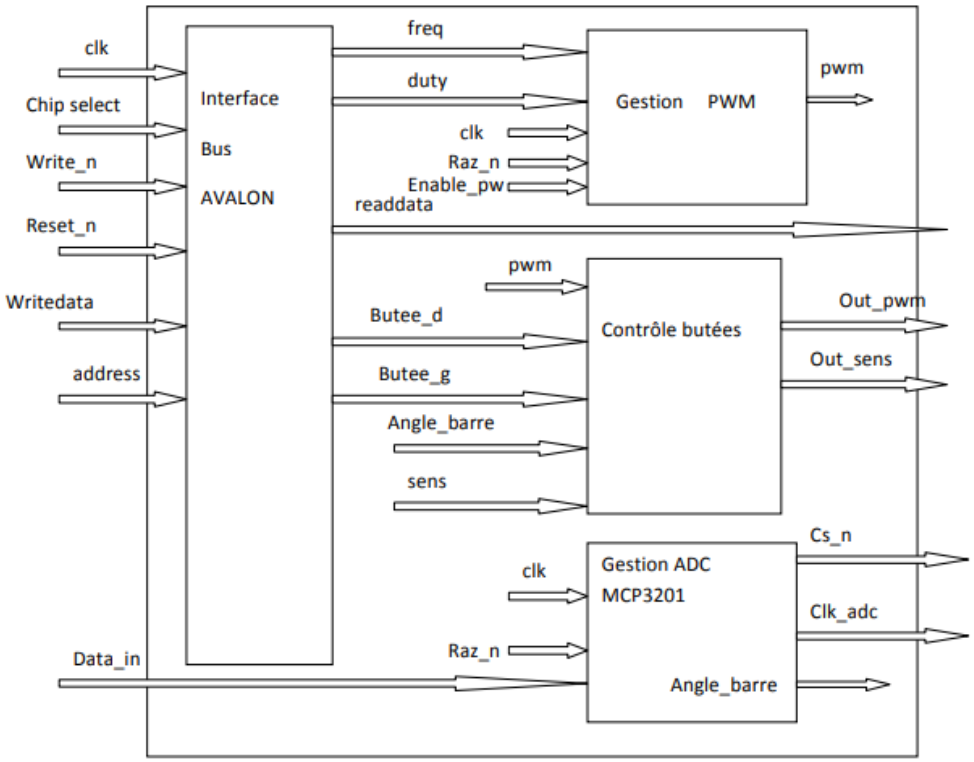

Figure 24:Bloc Fonctionnel du Vérin

- **Fonction Gestion PWM:** Cette fonction a pour but de créer un signal PWM en sortie avec une possibilité de modifier la largeur d’impulsion en changent la valeur de rapport cyclique et aussi de modifier la période en agissant sur le signal “Freq” plus le rapport cyclique est proche de 0,5 plus le vérin bouge plus vite.

- **Fonction Contrôle Butées :** Cette fonction permet de forcer le signal PWM à 0 si la barre de vérin atteint les valeurs seuil fixé pour le vérin (l“Butée_d” - “Butée_g”) ça permet aussi de contrôler le sens du déplacement de la barre et de nous indiquer l’arrivée de la tige aux extrémités en les indiquant sur les fins de course droite et gauche.

- **Fonction de Gestion ADC MCP 3201:** Cette fonction permet de récupérer la donnée de l’angle du vérin à partir du convertisseur AN MCP 3201 chaque 100 ms, puis envoyer cette valeur au bloc de contrôle butées afin de commander le vérin.

5.3. Fonctionnement du convertisseur

Pour extraire les données, le convertisseur utilise trois signaux :

- ✓ **CS:** un signal qui indique le début et la fin de la conversion. Quand il est à 0 en commence le processus d'acquisition des données et au moment où il passe à 1 ça veut dire que l'acquisition et finit.
- ✓ **CLK:** est l'horloge du convertisseur et qui est choisie de telle façon que chaque période soit équivalente à un bit numérique. Sa valeur est de 1 MHz.
- ✓ **Dout :** est le signal d'entrée qui contient l'information numérique à propos de l'angle et qui entre dans le processus de traitement pour extraire les données d'angle barre.

Au début le signal CS est en état haut, dès qu'on veut commencer l'acquisition des données, on active le CS en le remettant en état bas.

Comme il est montré dans la figure ci-dessous, en remarque qu'au niveau du signal Dout les bites utiles commencent juste après le bit de sécurité qui est le 4ᵉ bit qui doit être à la zéro logique, aussi en remarque que le début du bit commence avec le front montant et la fin du bit se fait sur le front descendant on sélectionne les 12 bits qui suit les bit de sécurité.

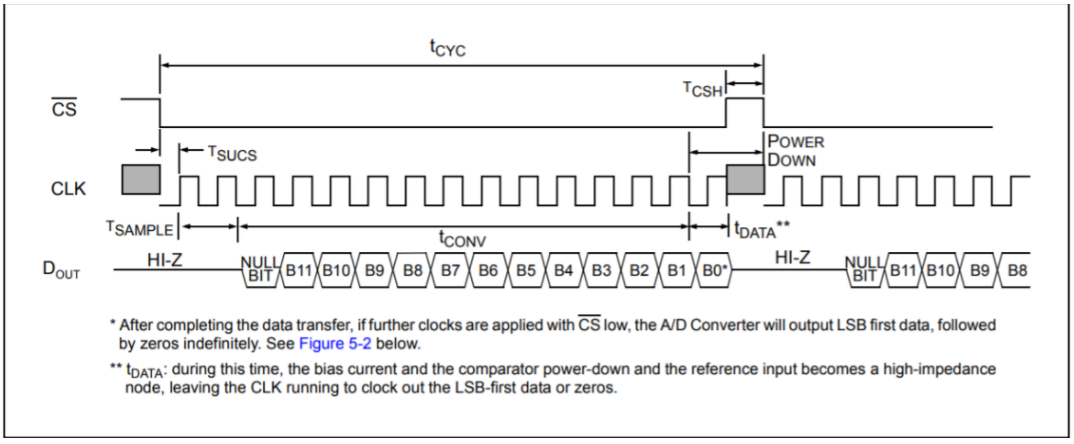

Figure 25: Fonctionnement de l'ADC

5.4. Circuit d'interface du vérin

Le bloc ci-dessous montre les différentes sous fonction du vérin commencent par la génération de la fréquence jusqu'au contrôle de butée.

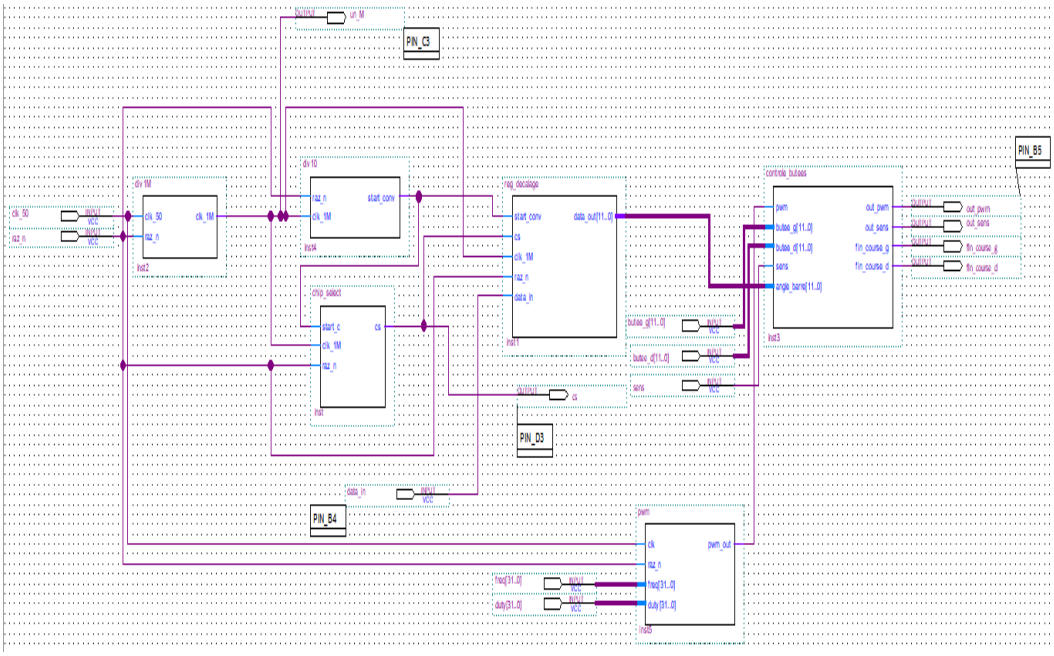

Figure 26:fichier bdf du verin

- **Bloc PWM:** son rôle est de générer un signal PWM en sortie pour pouvoir gérer la vitesse du vérin. Pour créer ce signal on doit fixer la fréquence et le rapport cyclique (cela dépendra de la vitesse choisie pour déplacer le vérin). En fixant la fréquence on va pouvoir creer une période, et on fixant le rapport cycle on va pouvoir créer le signal carré PWM(état haut pour un duty cycle inférieur à la valeur fixée)

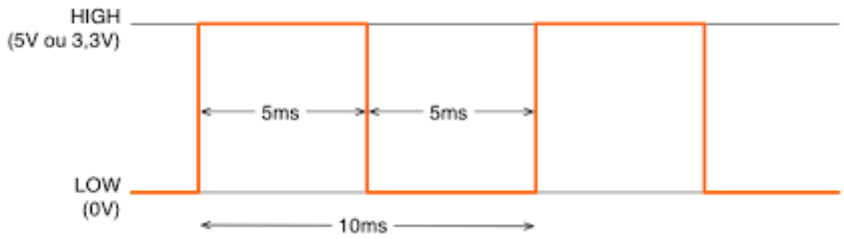

Figure 27: Signal du PWM

Dans cet exemple on a un rapport cyclique de 50% et une période de 10ms, donc durant les premières 5ms le signale sera à l'état haut et durant les 5ms qui suit il sera à l'état bas.

- **Bloc Contrôle_Butée:** Ce bloc permet de contrôler le mouvement du vérin, pour le réaliser on aura besoin du signal créer dans le bloc précédent “PWM”, de la position du vérin "angle barre", des valeurs seuils pour pouvoir comparer à chaque fois la position du vérin avec ces valeurs qui sont "butée_g ”et “butée_d” et finalement le sens pour savoir si le vérin et en mouvement vers la

gauche ou vers la droite. Si l'angle barre soit supérieur ou égale à la valeur seuil de butée (butée_g si le sens du déplacement du vérin est vers la gauche, idem pour la butée_d) il faut mettre le signal PWM en état bas, puis générer un signal de fin de course "fin_course_d" ou "fin_course_d" (selon le sens de déplacement).

- **Bloc Registre_Décalage**: Ce bloc permet d'avoir en sortie la donnée de l'angle barre qu'on doit envoyer au bloc Contrôle_Butée (pour faire la comparaison avec les valeurs seuils) pour récupérer cette donnée on aura besoin d'un convertisseur.

- Afin de créer ce convertisseur on aura besoin du **bloc Chip Select** dont la seule sortie sera le signal CS indispensable pour le fonctionnement du **Registre de décalage.** Pour générer celui-ci, on aura besoin de deux entrées : une entrée clk connectée à la sortie de l'horloge principale 1MHz; une entrée start_c représentant le signal start_conv de fréquence 10Hz (ou de période 100 ms) généré par le bloc **div100_ms** (voir code du bloc en annexe). Pour commencer la conversion des données, il faudrait que les signaux CS et start_conv soient à l'état bas (condition de démarrage) ; puis effectuer la lecture de données à partir du 4iem front montant de l'horloge avec un décalage par bit à chaque prochain front récupérant les 12 bits (0..11) et affectées au signal de sortie Dout associé à l'angle barre.

- **Bloc Horloge Principale div1M:** Permet de créer une nouvelle horloge de 1Mhertz à partir de l'horloge de base 50 MHz du FPGA. Il a été réalisé à l'aide d'un compteur diviseur de fréquence (par 50) donnant un signal de période 1 us. Tous les autres blocs dépendent de cette horloge.

5.5. Implémentation sur SOPC

Pour la partie en SOPC, nous avons créé le fichier Avalon qui permet de connecter le vérin a l'ensemble de système à l'aide de bus Avalon. Pour avoir une sortie des pin principale au niveau de bloc "Avalon vérin" il faut déplacer tous les pin dans le fichier conduit tout 'en donnant le même nom pour le type de signal.

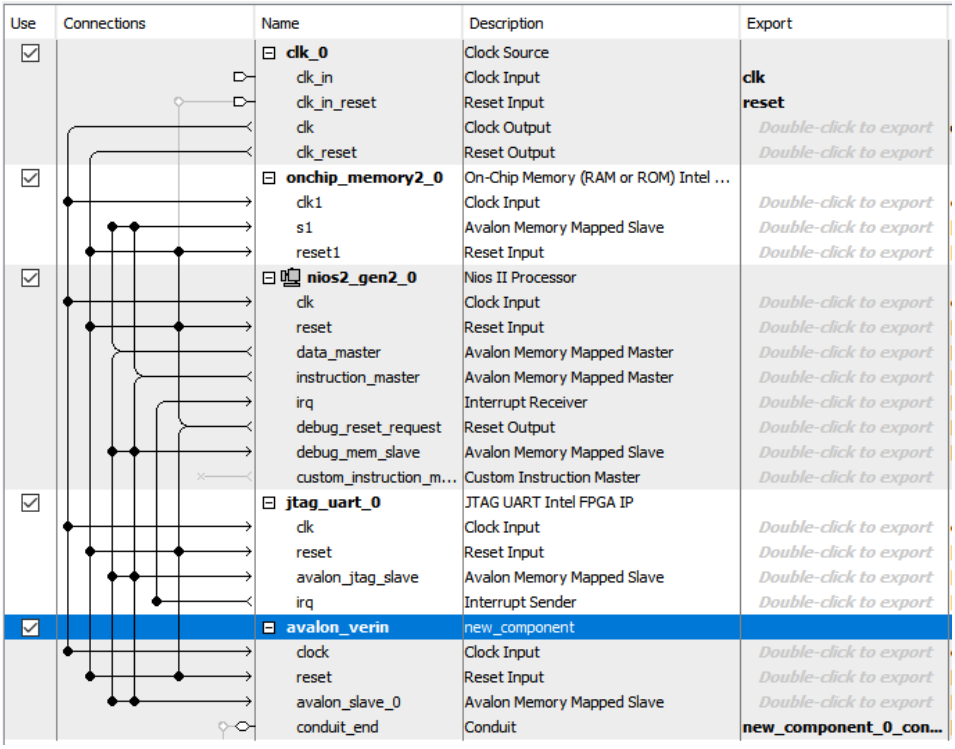

Figure 28: Connection Vérin

La figure ci-dessous montrant le fichier top qui gère la partie Avalon de Vérin.

```
 6  port (
 7      clk, reset_n, data_in  :   in std_logic;
 8      out_pwm, cs, out_sens          :   out std_logic
 9      );
10  end entity TOP;
11
12  architecture rtl_S of TOP is
13     component avalon_verin is
14        port (
15           clk_clk                              : in  std_logic := 'X'; -- clk
16           new_component_0_conduit_end_data_in  : in  std_logic := 'X'; -- data_in
17           new_component_0_conduit_end_cs       : out std_logic;        -- cs
18           new_component_0_conduit_end_out_pwm  : out std_logic;        -- out_pwm
19           new_component_0_conduit_end_out_sens : out std_logic;        -- out_sens
20           reset_reset_n                        : in  std_logic := 'X'  -- reset_n
21        );
22     end component avalon_verin;
23  begin
24     u0 : component avalon_verin
25        port map (
26           clk_clk                              => clk,                                    --
27           new_component_0_conduit_end_data_in  => data_in,  -- new_component_0_conduit_end.data_in
28           new_component_0_conduit_end_cs       => cs,       --                                .cs
29           new_component_0_conduit_end_out_pwm  => out_pwm,  --                                .out_pwm
30           new_component_0_conduit_end_out_sens => out_sens, --                                .out_sens
31           reset_reset_n                        => reset_n                          --
32        );
33
34
35  end architecture;
```

Figure 29: Code C du Vérin

Conclusion

Tout au long de ce BE, nous avons réalisé des fonctions permettant de créer des sous-systèmes (anémomètre ; compas ; gestion vérin) rentrant dans la composition générale du système d'asservissement d'un bateau. Pour cela, nous avons créé différents blocs associés l'un l'autre à cet effet ; au nombre de ceci on citera essentiellement des diviseurs de fréquences, détecteur de front montant, des compteurs, des registres, …

Nous avons néanmoins rencontré de légères difficultés quant à la compréhension du fonctionnement du bloc de conversion de données et ainsi du développement du code qui va avec. Aussi, le test de validation de la fonction complexe n'a pas pu être réalisé à cause d'une mauvaise gestion du temps attribué pour chaque fonction.
En définitive, cet exercice a été très bénéfique en termes de réalisation projet, travail collectif et une maîtrise plus poussée du langage VHDL à l'aide de l'outil QARTUS.

Annexes

Les différentes annexes comportant le code VHDL, Code C, Fichier SOPC etc. sont disponible sur GitHub sur le lien ci-dessous :

Dans ce lien, vous allez trouver un document PDF : Compte rendu du projet contient les chapitres réalisés durant le projet.

Ainsi qu'un dossier embarqué comportant les deux fonctions simple (F1 : Compas et F2 : Anémomètre) et la fonction complexe (F6 : Gestion Vérin).