

Synthèse et mise en œuvre des systèmes

Bureau d'étude VHDL



Pilote de barre franche pour voiliers

M2 SME

Rédigé par :

MBOUNGOU Frangely
BA WAZIR Ahmed

SOMMAIRE

I-Introduction.....	3
II - Conception.....	4
1 - Présentation du système.....	4
2 - Architecture.....	5
III - Développements et simulations.....	6
1 - Fonction simple : Compas.....	6
a - Analyse fonctionnelle	7
b - Codage	9
c - Simulation.....	13
2 - Fonction complexe : Interface Homme Machine.....	15
a - Analyse fonctionnelle	16
b - Codage	17
c - Simulation.....	19
IV - Intégration des fonctions sur le SOPC.....	20
1 - Conception.....	20
2 - Réalisation.....	20
V - Conclusion.....	24

I - Introduction

L'objectif de ce bureau d'études est de mettre en pratique l'automatisation d'un système embarqué à travers sa conception, la validation par la simulation et l'implémentation via des outils de développement et d'intégration.

Pour ce faire, l'application choisie est de reproduire à l'aide de capteurs (anémomètre, compas,...) et d'une maquette où l'on retrouvera un vérin piloté par un PWM via un pont en H, l'angle de barre sera donné par un CAN via un potentiomètre, les informations NMEA seront données via le port série RS232 dans le but de représenter l'acquisition d'une barre franche pour voiliers.

Ce rapport va donc mettre en lumière la conception, la réalisation et l'intégration de notre système.

II - CONCEPTION

1- Présentation du système

Un pilote de barre franche pour voiliers, également appelé pilote automatique pour barre franche, est un dispositif électronique conçu pour aider à maintenir le cap d'un voilier en utilisant une barre franche.

Ce type de pilote automatique est monté sur le côté de la barre franche du bateau et utilise généralement des capteurs notamment un anémomètre qui va mesurer la fréquence, et un compas qui va mesurer un signal pwm, pour détecter les mouvements du bateau et du gouvernail.

Le système est constitué d'un vérin, qui dans sa fonction de pilotage, va ajuster la direction de la barre franche pour maintenir le cap souhaité par le navigateur.

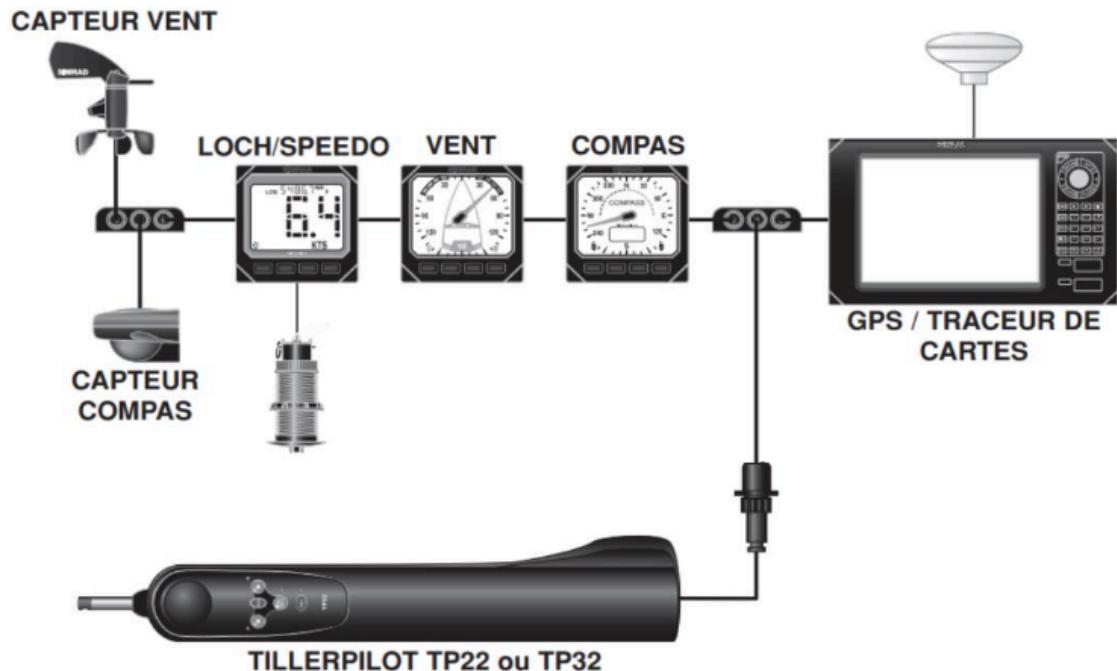


Figure 1 : Présentation de pilote de barre franche

2- Architecture

Le système à réaliser sera décomposer selon différentes fonctions suivantes :

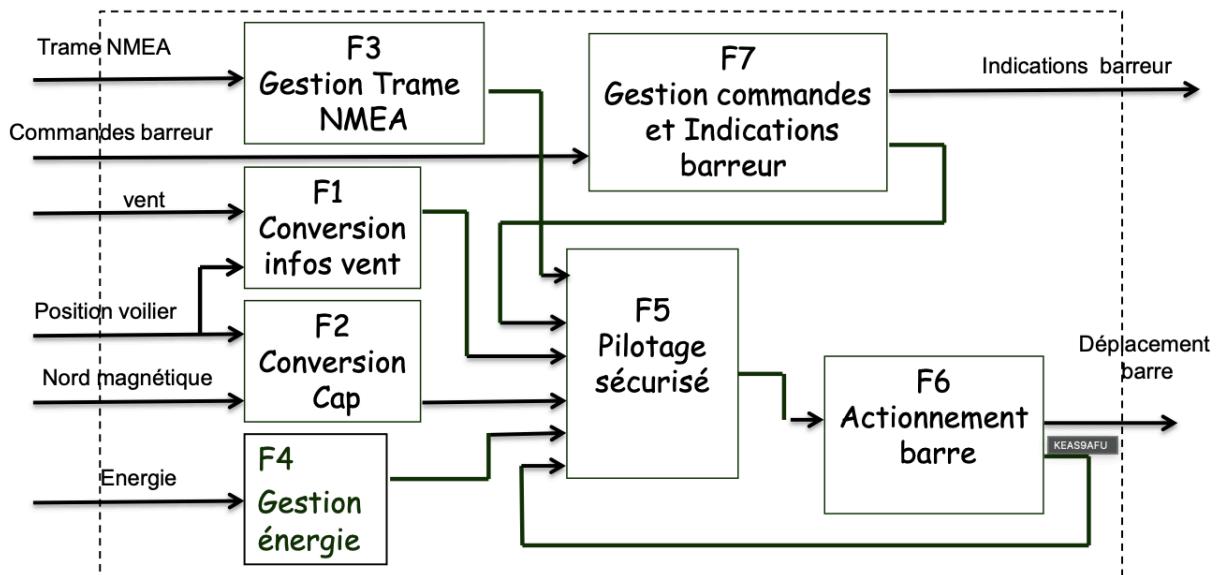


Figure 2 : Décomposition en schéma bloc

- Une fonction générant un **signal PWM** qui servira notamment de signal d'entrée.
- Une fonction qui permettra de lire la mesure de la vitesse du vent (0-250km/h) à la sortie de l'**anémomètre** qui est une sortie logique de fréquence variable (0 à 250 Hz).
- Une fonction qui utilise un **compas** pour récupérer les mesures d'angles sur le plan horizontal du voilier, permettant de donner un cap à celui-ci.
- Une fonction de **gestion vérin** gérant le pilotage de la barre franche.
- Une fonction qui permet la gestion de l'**Interface Homme Machine** (composée de différents boutons[Bâbord, Tribord, Auto/Manuel], des LEDS ainsi qu'un buzzer).

Pour notre part, le choix s'est porté de réaliser la fonction simple Compas puis la fonction complexe IHM.

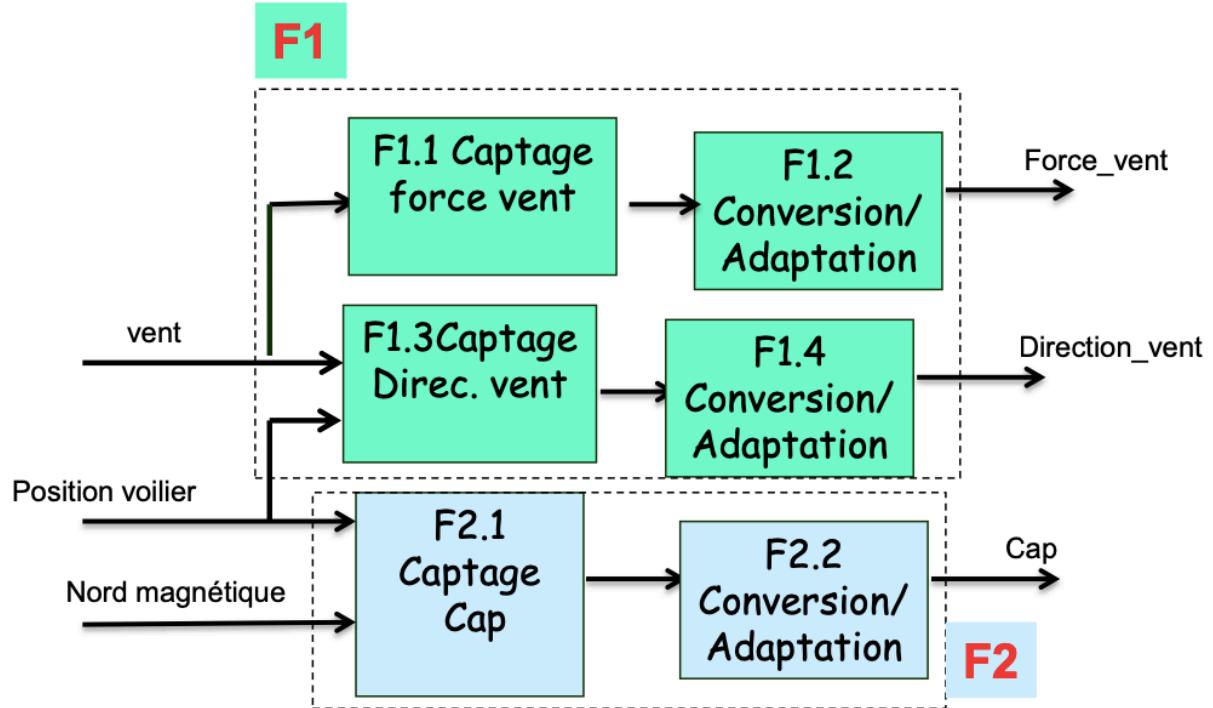


Figure 3 : Blocs fonctions simples et complexes

II - Développements et simulations

1- Fonction simple : Compas

Le module compas a pour but de récupérer des mesures d'angles afin de fixer le cap. Il permet de faire l'acquisition de données, pour en délivrer les directions suivantes : Nord, Sud, Est et Ouest.

Celui-ci utilise notamment un signal d'entrée PWM qui fait en sorte que lorsque la boussole se met en rotation, une impulsion est générée. La largeur d'impulsion varie de 1 ms pour 0° à 36,99 ms pour $359,9^\circ$, cela équivaut à 100 us/dégré avec un décalage de +1 ms pour 0° . On peut donc déduire que 1 ms est équivalent à 10° .

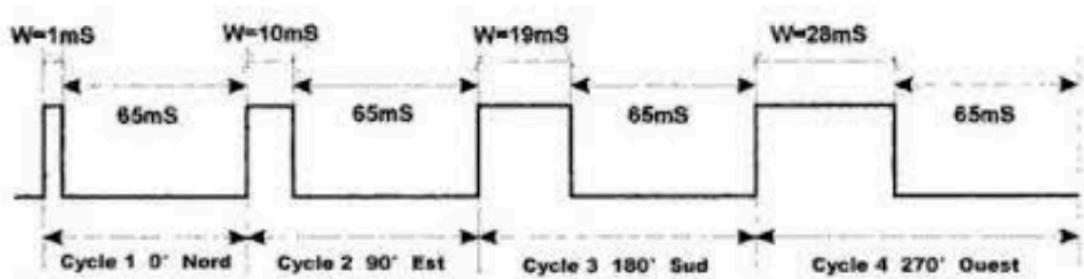


Figure 4 : Schéma d'acquisition de la boussole

La période a donc une durée minimum de 66ms (pour un angle de 0°) et de 102 ms maximum (pour un angle de 359,9°).

a- Analyse fonctionnelle

Le module compas présente les entrées et sorties suivantes :

Entrées

- clk_50M : une horloge à 50 MHZ
- raz_n : reset actif à 0 afin d'initialiser le circuit
- in_pwm_compas : signal PWM d'entrée de la boussole qui va donc varier entre 1 ms et 36,9 ms
- start_stop : elle permet de démarrer une acquisition lorsque le mode fixé est en monocoup et que c'est = 1.
- Continu : permet de fixer le mode de fonctionnement du compas, soit en mode en continu lorsque continu = 1 en rafraîchissement les données toutes les secondes. Sinon si continu = 0, on est donc en mode monocoup qui est donc activé si l'entrée start_stop = 1 pour valider la prise en compte des données.

Sorties

- data_compas : elle permet d'exprimer en degré la valeur du cap codé sur 9 bits.
- data_valid : elle permet de vérifier la validité d'une mesure. Elle est notamment nécessaire quand le mode de fonctionnement est en mode monocoup afin d'être sûr de notre valeur de sortie data_compas.
On peut donc décomposer la gestion du module compas tel que :

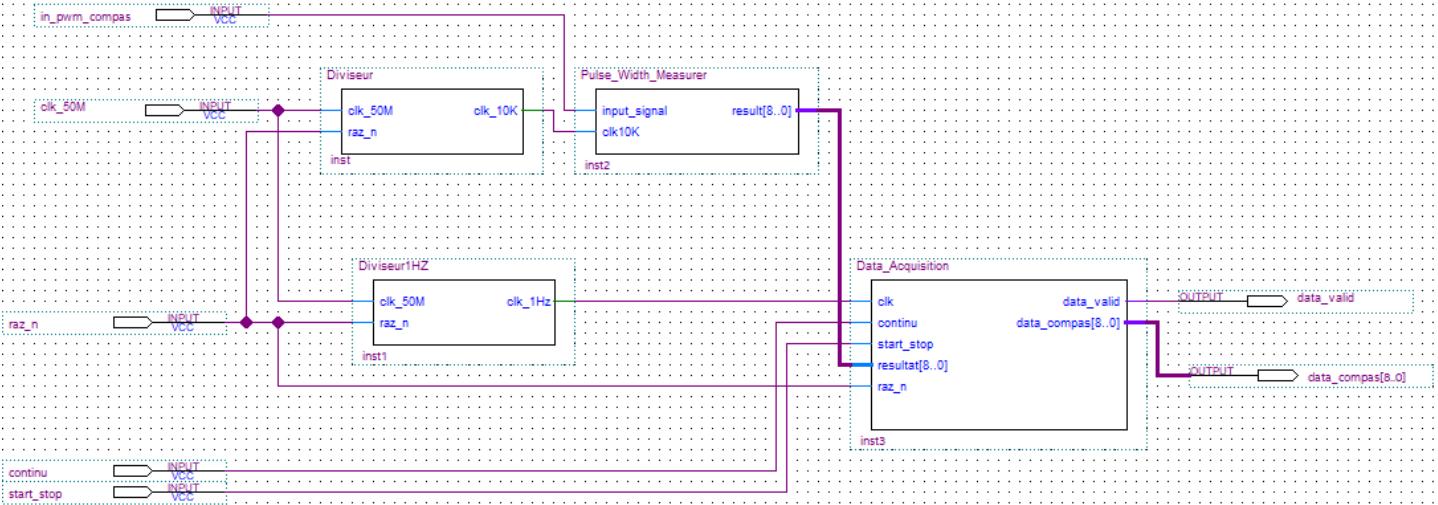


Figure 5 : Schéma fonctionnel du module compas

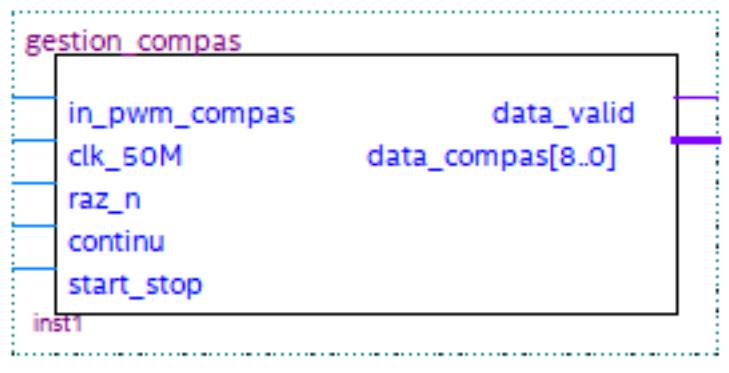


Figure 6 : Bloc fonctionnel du module compas

b- Codage

L'implémentation de notre module compas se fera à travers le logiciel Quartus 18 où nos différents blocs seront codés en langage VHDL.

Diviseur : ce bloc représente un diviseur permettant de générer un signal de fréquence à 10 kHz équivalent à un degré. Il va notamment permettre de déterminer le nombre de degrés dans chaque période du signal PWM.

```
1 —Diviseur de fréquence
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 use ieee.std_logic_unsigned.all;
5
6
7 entity Diviseur is
8     Port ( clk_50M : in STD_LOGIC;
9             raz_n : in STD_LOGIC;
10            clk_10K : inout STD_LOGIC
11        );
12 end Diviseur;
13
14 architecture Behavioral of Diviseur is
15
16 begin
17     process (clk_50M, raz_n)
18         variable count : integer range 0 to 4999 ; — Compteur pour diviser le signal
19
20     begin
21         if raz_n = '0' then
22             count := 0; — Réinitialisation du compteur lorsque raz_n est actif
23             clk_10K <= '0';
24         else if (clk_50M'event and clk_50M='1') then
25             count := count + 1;
26             if count = 4999 then
27                 count := 0;
28                 clk_10K <= not clk_10K; — Inversion de la sortie pour générer une fréquence de 10 KHz
29             else
30                 count := count ;
31                 — clk_10K <= '1';
32             end if;
33         end if;
34         end if;
35
36     end process;
37 end Behavioral;
```

Figure 7 : Code du diviseur 10KHz

Diviseur1HZ : Ce bloc va permettre de générer la seconde pour le rafraîchissement des données

```
1 --Diviseur de fréquence
2 LIBRARY ieee;
3 USE ieee.std_logic_1164.all;
4 use ieee.std_logic_unsigned.all;
5
6
7 entity Diviseur1HZ is
8     Port ( clk_50M : in STD_LOGIC;
9             raz_n : in STD_LOGIC;
10            clk_1Hz : inout STD_LOGIC
11        );
12 end Diviseur1HZ;
13
14 architecture Behavioral of Diviseur1HZ is
15 begin
16 process (clk_50M, raz_n)
17     variable count : integer range 0 to 49999999 ; -- Compteur pour diviser le signal
18
19 begin
20     if raz_n = '0' then
21         count := 0; -- Réinitialisation du compteur lorsque raz_n est actif
22         clk_1Hz <= '0';
23     else if (clk_50M'event and clk_50M='1') then
24         count := count + 1;
25         if count = 49999999 then
26             count := 0;
27             clk_1Hz <= not clk_1Hz; -- Inversion de la sortie pour générer une fréquence de 1Hz
28         else
29             count := count ;
30
31         end if;
32     end if;
33     end if;
34
35 end process;
36 end Behavioral;
```

Figure 8 : Code du diviseur 1Hz

Pulse_Width_Measurer : ce bloc va permettre la recopie du signal du diviseur de 10 kHz suivant les fronts montants du signal PWM

```
1 library IEEE;
2 use IEEE.STD_LOGIC_ARITH.ALL;
3 use ieee.numeric_std.all;
4 use ieee.std_logic_unsigned.all;
5 use ieee.numeric_std.all;
6 use ieee.std_logic_1164.all;
7
8 entity Pulse_Width_Measurer is
9     Port ( in_pwm_compas : in STD_LOGIC;
10             clk10K : in STD_LOGIC;
11             result : out STD_LOGIC_VECTOR(8 downto 0));
12 end Pulse_Width_Measurer;
13
14 architecture Behavioral of Pulse_Width_Measurer is
15     signal pulse_width : natural := 0; --compteur de temps haut de signal PWM_in = in_pwm_compas
16     -- signal prev_clk : STD_LOGIC := '0';
17
18 begin
19     process (clk10K)
20     begin
21         if (clk10K'event and clk10K='1') then
22             if in_pwm_compas = '1' then
23                 pulse_width <= pulse_width + 1;--compteur
24             else
25                 if      pulse_width/=0 then
26                     -- pulse_width<=pulse_width;
27                     result <= std_logic_vector(to_unsigned(pulse_width, 9));--mettre la valeur de compteur en resultat
28                     pulse_width <= pulse_width-pulse_width;-- on remet le compteur à 0
29                 else
30                     pulse_width <= pulse_width;--pas de changement
31                 end if;
32                 -- prev_clk <= clk10K;
33             end if;
34         end if;
35     end process;
36
37 end Behavioral;
```

Figure 9 : Code du compteur

Data_Acquisition : Ce bloc représente le traitement des données pour aboutir aux signaux de sorties attendus suivant le mode fonctionnement.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_ARITH.ALL;
4 use IEEE.STD_LOGIC_UNSIGNED.ALL;
5 use IEEE.numeric_std.ALL;
6
7
8 entity Data_Acquisition is
9     Port ( clk : in STD_LOGIC; -- Horloge 1 Hz
10            continu : in STD_LOGIC; -- Mode Continu (1) ou Monocoup (0)
11            start_stop : in STD_LOGIC; -- Démarrer une acquisition (1) ou remet à 0 data_valid (0)
12            resultat : in STD_LOGIC_VECTOR(8 downto 0); -- Valeur de degré en binaire sur 9 bits
13            data_valid : out STD_LOGIC; -- Indicateur de mesure valide
14            data_compas : out STD_LOGIC_VECTOR(8 downto 0); -- Résultat de l'acquisition
15            raz_n : in STD_LOGIC
16        );
17 end Data_Acquisition;
18
19 architecture Behavioral of Data_Acquisition is
20     signal data_compas_internal : STD_LOGIC_VECTOR(8 downto 0);
21     signal data_valid_internal : STD_LOGIC := '0';
22
23 begin
24     process (clk,start_stop,raz_n)
25     begin
26
27         if(raz_n='0') then --mettre à zéro le circuit à zero
28             data_valid_internal <= '0';
29             data_compas_internal <= "000000000";
30         else
31             if (clk'event and clk='1') then
32                 if continu = '1' then-- Mode continu
33                     data_compas_internal <= resultat;
34
35                 else
36                     data_compas_internal <= "000000000";
37                 end if;
38
39             end if;
40             if (start_stop = '1' and raz_n='1') then-- en mode monocoup
41                 data_valid_internal <= '1';
42                 data_compas_internal <= resultat;
43             else
44                 data_valid_internal <= '0';
45             end if;
46         end if;
47     end process;
48
49     data_valid <= data_valid_internal;
50     data_compas <= data_compas_internal;
51 end Behavioral;
```

Figure 10 : Code du bloc d'acquisition

c- Simulation

Une fois le développement de nos différents blocs réalisés, on vient les tester via l'interface ModelSim intégré au logiciel Quartus 18.

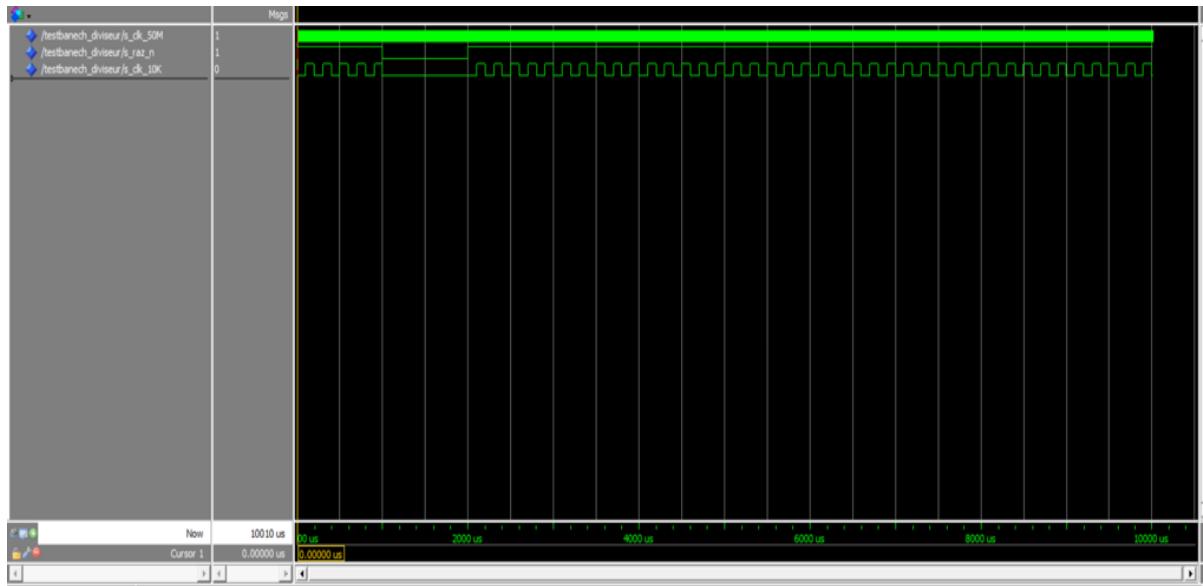


Figure 11 : Simulation du diviseur 10 khz



Figure 12 : Simulation du diviseur 1 hz

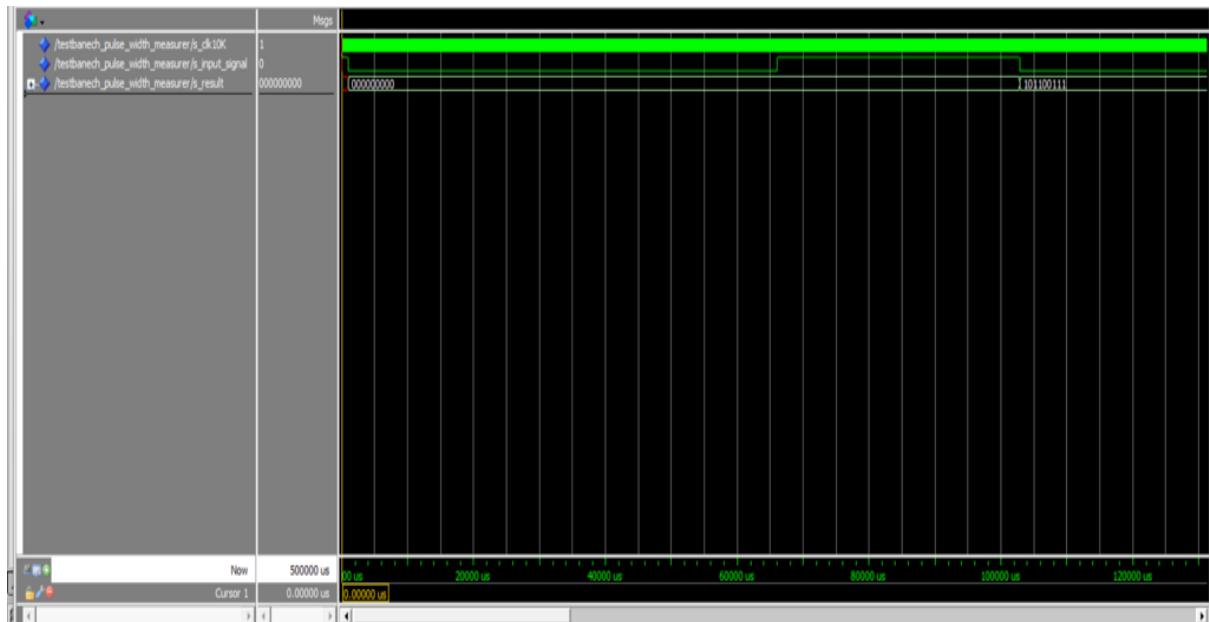


Figure 13 : Simulation du compteur

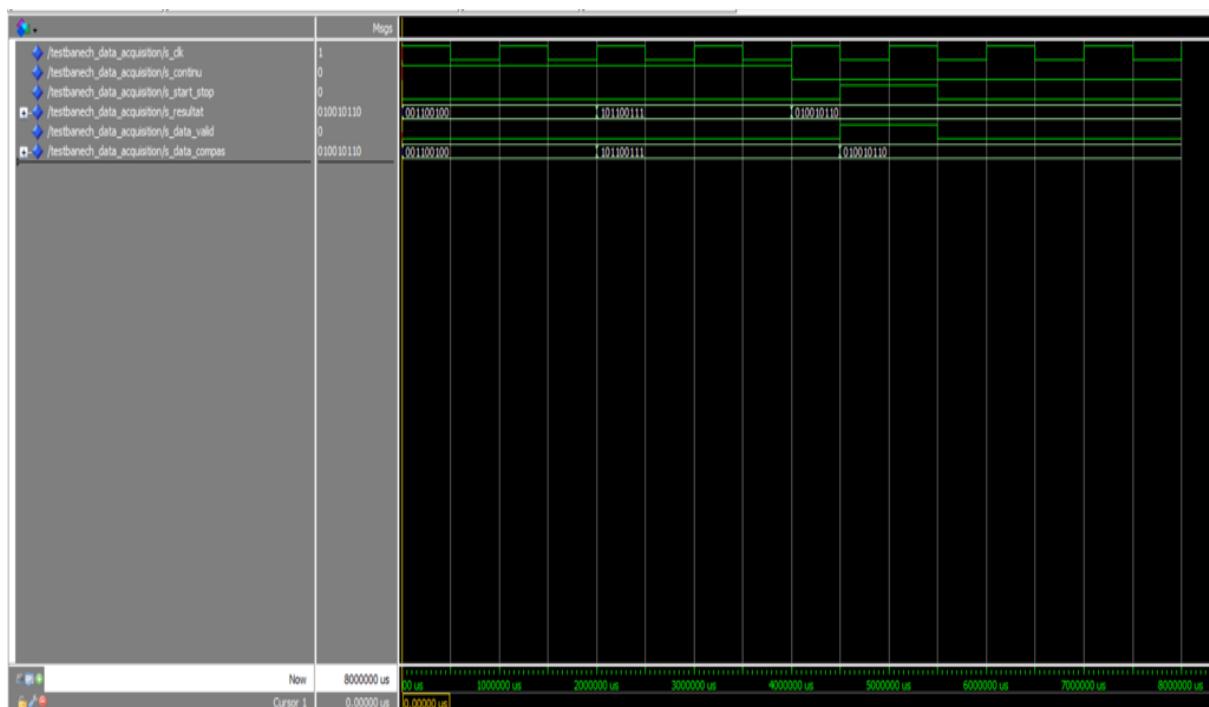


Figure 14 : Simulation du bloc d'acquisition

La fonctionnalité du module compas est tel que l'on récupère notre signal acquisitionné en sortie du module. On a ainsi testé la fonction via la carte **TERASIC DE2-C35** car elle nous permettait de visualiser tous les bits de sortie représentant la durée du rapport cyclique correspondant au cap. Ce signal est par ailleurs divisé par deux, et nous avons opté pour convertir notre

signal de sortie en degrés dans le code C lors de l'intégration de la fonction dans le NIOS.

2- Fonction complexe : Interface Homme Machine

Dans cette partie, on va représenter la mise en place du pilotage automatique de la barre franche pour les pilotes Simrad TP10, TP22 et TP32 sous forme d'une machine d'état. Celui-ci doit répondre aux besoins spécifiques de la navigation maritime, où le maintien du cap d'un bateau sans intervention constante est crucial.

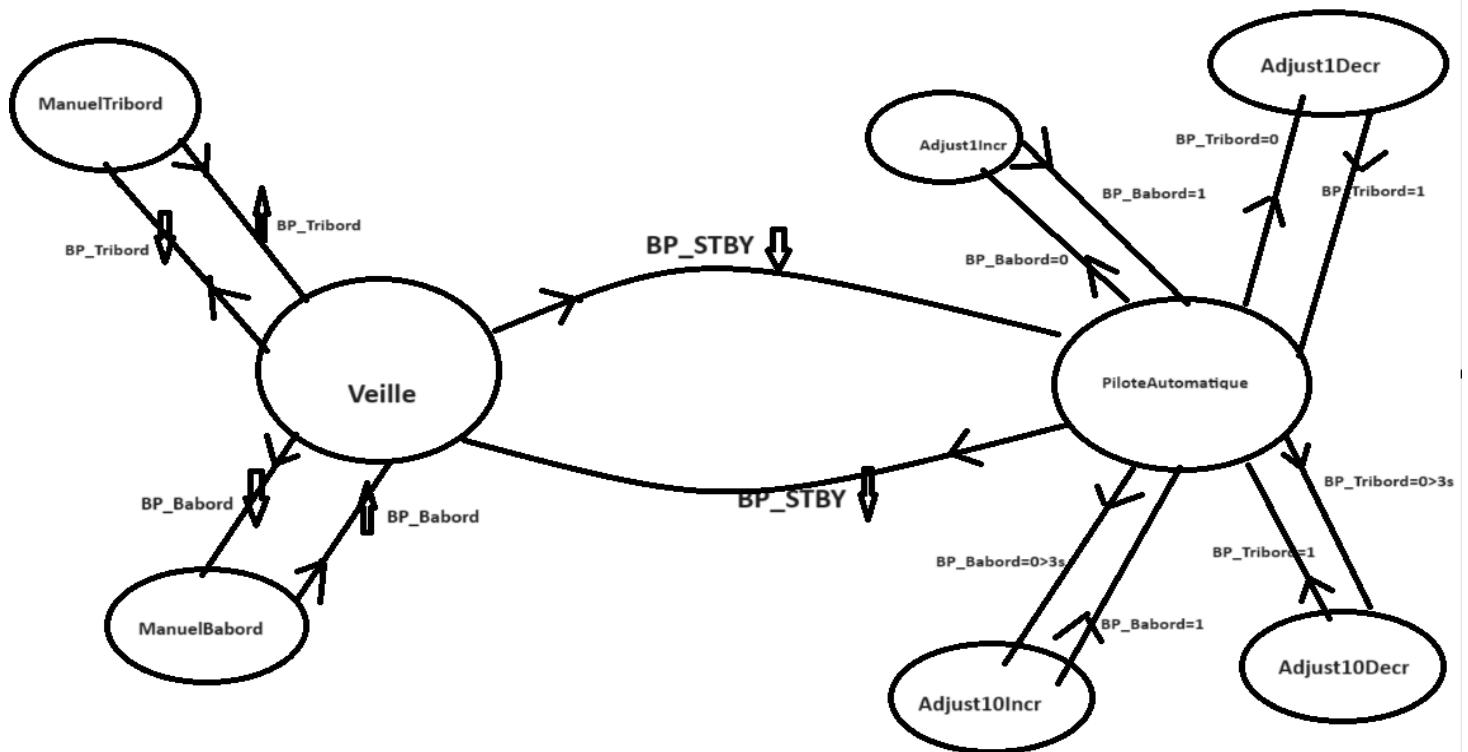


Figure 15 : Machine d'états IHM

États d'entrée : BP_Tribord, BP_Babord, BP_STBY.

Sorties : ledBabord, ledTribord, ledSTBY et le signal sonore out_bip

Mode de fonctionnement : Veille, ManuelBabord, ManuelTribord, PiloteAutomatique et Adjust (états d'ajustement)

Chaque état est associé à des actions spécifiques, telles que l'allumage de LEDs, le réglage du sonneur, et la gestion des transitions entre les états.

a- Analyse fonctionnelle

Cette fonction est composée de 3 blocs :

- PWM Generator : obtenir le signal que les Leds prennent quand elles sont en faible allumage;
- Diviseur1Hz : permettre le clignotement de la LED
- PilotStateMachine : simulation de la machine d'état

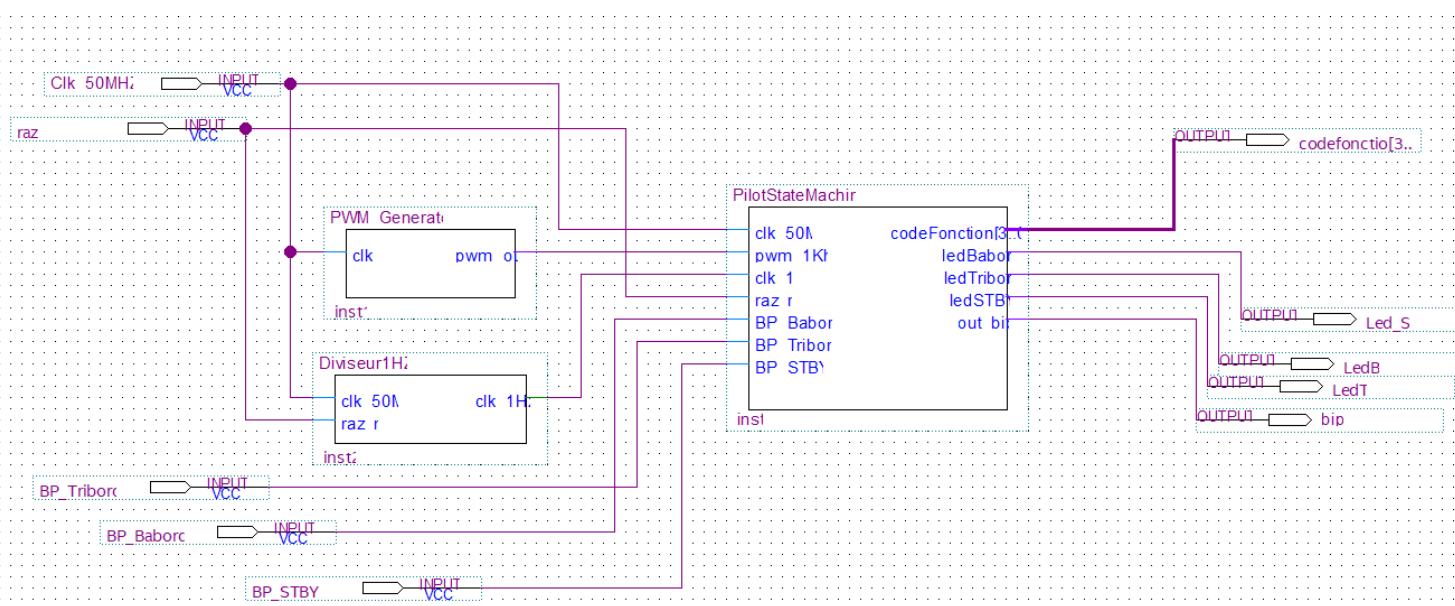


Figure 16 : Schéma fonctionnel de l'IHM

b- Codage

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.STD_LOGIC_UNSIGNED.ALL;
4 use IEEE.STD_LOGIC_ARITH.ALL;
5
6 entity PilotStateMachine is
7   Port ( clk_50M : in STD_LOGIC;
8         pwm_1Khz : in STD_LOGIC;
9         clk_1 : in STD_LOGIC;
10        raz_n : in STD_LOGIC;
11        BP_Babord, BP_Tribord, BP_STBY : in STD_LOGIC;
12        codeFonction : out STD_LOGIC_VECTOR(3 downto 0);
13        ledBabord, ledTribord, ledSTBY, out_bip : out STD_LOGIC
14      );
15 end PilotStateMachine;
16
17 architecture Behavioral of PilotStateMachine is
18   type state_type is (Veille, ManuelBabord, ManuelTribord, PiloteAutomatique, Adjust1Incr, Adjust1Decr, Adjust10Incr, Adjust10Decr);
19   signal current_state, next_state : state_type;
20   signal time_counter : integer := 350000000; -- Valeur de consigne de cap en degrés
21
22 begin
23
24   process(clk_50M, raz_n)
25   begin
26     if raz_n = '0' then
27       -- états des sorties
28       out_bip <= "0";
29       codeFonction <= "0000" ;
30       ledBabord <= '0';
31       ledTribord <= '0';
32       ledSTBY<= '0';
33
34     elsif rising_edge(clk_50M) then
35       current_state <= next_state; -- Met à jour l'état actuel avec l'état suivant
36     end if;
37   end process;
38
39   process(current_state, BP_Babord, BP_Tribord, BP_STBY)
40   variable counter : integer range 0 to 350000000 ; -- Compteur pour la durée de l'appui
41   begin
42     -- Logique de transition d'état basée sur les entrées BP_Babord, BP_Tribord, et BP_STBY
43     case current_state is
44       when Veille =>
```

```

48     ledSTBY <= clk_1;--clignoter la Led
49     ledBabord <= pwm_1Khz;--allumage faible
50     ledTribord <= pwm_1Khz;--allumage faible
51     if BP_Babord = '0' then
52         next_state <= ManuelBabord; -- changement d'état
53     elseif BP_Tribord = '0' then
54         next_state <= ManuelTribord; -- changement d'état
55     elseif BP_STBY = '0' then
56         wait until BP_STBY = '1';
57
58         next_state <= PiloteAutomatique;-- changement d'état
59     else
60         next_state <= Veille;
61     end if;
62
63     when ManuelBabord =>
64         out_bip <= '1';-- faire fonctionner le sonneur
65
66     when ManuelTribord =>
67         out_bip <= '1';-- faire fonctionner le sonneur
68
69     when PiloteAutomatique =>
70
71
72     ledTribord <= pwm_1Khz;--allumage faible
73     ledBabord <= pwm_1Khz;--allumage faible
74     ledSTBY <= '1'; -- Logique de transition d'état pour le mode pilote automatique*
75
76
77     if BP_Babord = '0' then
78
79         while BP_Babord = '0' and counter < 150000000 loop -- compter la duré de l'appui
80             counter := counter + 1;
81
82         end loop;
83
84
85
86     if counter>150000000 then --appui plus que 3 secondes
87
88         counter := 0;
89         next_state <=Adjust1Incr;
90     else --appui moins de 3 secondes
91         next_state <=Adjust1Incr;
92         counter := 0;
93
94     end if;
95     elseif BP_Tribord = '0' then
96         while BP_Tribord = '0' loop
97             -- Incrémentation du compteur
98             counter:= counter + 1;
99
100
101    -- Sortir de la boucle si une certaine condition est satisfaite
102    if BP_Tribord='1' then
103        exit;
104    end if;
105    end loop;
106    if counter>150000000 then --appui plus que 3 seconds
107
108        next_state <=Adjust1Decr;
109        counter := 0;
110    else --appui moins de 3 secondes
111        next_state <=Adjust1Decr;
112        counter := 0;
113
114    end if ;
115    elseif BP_STBY = '0' then
116        wait until BP_STBY = '1';
117        next_state <= PiloteAutomatique;
118    else
119        next_state <= Veille;
120    end if;
121
122
123    when Adjust1Incr =>
124        ledBabord <= '1';
125        wait for 1000 ms;
126        ledBabord <= '0';
127        ---réglage du sonneur
128        out_bip <= '1';
129        wait for 1000 ms;
130        out_bip <= '0';
131
132
133    when Adjust1Decr =>
134        ledTribord <= '1';
135        wait for 1000 ms;
136        ledTribord <= '0';
137        ---réglage du sonneur
138        out_bip <= '1';

```

```

138     out_bip <= '0';
139     when Adjust10Incr =>
140         ledBabord <= '1'; --clignoter la LED 2 fois
141         wait for 1000 ms;
142         ledBabord <= '0';
143         wait for 500 ms;
144         ledBabord <= '1';
145         wait for 1000 ms;
146         ledBabord <= '0';
147         out_bip <= '1';----réglage du sonneur
148         wait for 500 ms;
149         out_bip <= '0';
150         wait for 500 ms;
151         out_bip <= '1';
152         wait for 500 ms;
153         out_bip <= '0';
154     when Adjust10Decr =>
155         ledTribord <= '1';--clignoter la LED 2 fois
156         wait for 1000 ms;
157         ledTribord <= '0';
158         wait for 500 ms;
159         ledTribord <= '1';
160         wait for 1000 ms;
161         ledTribord <= '0';
162         out_bip <= '1';----réglage du sonneur
163         wait for 500 ms;
164         out_bip <= '0';
165         wait for 500 ms;
166         out_bip <= '1';
167         wait for 500 ms;
168         out_bip <= '0';
169     -----
170     when others =>
171         next_state <= Veille;
172     end case;
173 end process;
174 -- Logique de sortie basée sur l'état actuel
175 with current_state select
176     codeFonction <= "0000" when Veille, -- état de la sortie codeFonction pour le mode Veille
177                 "0001" when ManuelBabord,
178                 "0010" when ManuelTribord,
179                 "0011" when PiloteAutomatique, -- état de la sortie codeFonction pour le mode PiloteAutomatique
180                 "0100" when Adjust1Incr,
181                 "0101" when Adjust10Incr,
182                 "0111" when Adjust1Decr,
183                 "0110" when Adjust10Decr,-- D
184                 "0000" when others;
185 end Behavioral;

```

Figure 17 : Code de la machine d'état

c- Simulation

Par manque de temps, nous avons pas pu faire aboutir la simulation dû aux erreurs rencontrées sur l'implémentation du code. La dernière en date a été une erreur sur la boucle while , l'erreur nous indique qu'on est obligé de faire une boucle qui tient au plus 10000 front montants de l'horloge alors qu'on cherche à calculer la durée de l'appui sur le bouton si on a plus de 3 secondes ou non pour déterminer l'état suivant de la machine sachant qu'avec une horloge de 50Mhz , 10000 fronts montants ne permettent pas de calculer 3 secondes.

- ⚠ 10752 VHDL Process Statement error at PilotStateMachine.vhd(7), signal pilot_time is read before the process statement due to a nil process
- ⚠ 10536 VHDL Loop Statement error at PilotStateMachine.vhd(80): Loop must terminate within 10,000 iterations
- ⚠ 10441 VHDL Process Statement error at PilotStateMachine.vhd(40): Process Statement cannot contain both a sensitivity list and a Wait Statement

IV - Intégration des fonctions sur le SOPC

Dans le but d'intégrer nos composants dans la carte DE0 nano, on va concevoir un système hardware à travers l'outil Platform Designer qui va nous permettre de concevoir notre microcontrôleur comportant donc une partie processeur (softcore NIOS II) à laquelle on associe des périphériques (PIO, Timers, UART, USB, composants propriétaires, ...) et de la mémoire, que l'on va embarquer sur notre FPGA afin de la coupler à notre système complet.

1- Conception

Les différentes étapes servant à l'intégration des fonctions au sein du SOPC sont les suivantes :

- Écrire un code Avalon.vhd qui va faciliter la communication entre les différents modules matériels et logiciels en spécifiant comment les données sont transférées et interprétées dans le but de simplifier l'intégration des composants sur le FPGA.
- Générer le schéma bloc correspondant à notre composant.
- La validation se fait via le code C à travers l'environnement de développement Eclipse, en transmettant ce code à la carte et le code par debugage et en visualisant les signaux sur la carte et l'oscilloscope.

2- Réalisation

Dans le cadre de notre projet, on a donc intégré les fonctions pwm et compas dans notre FPGA grâce aux codes Avalon.vhd.

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use IEEE.std_logic_arith.all;

-- LIBRARY work;

ENTITY compas_avalon IS
  PORT
  (
    clk, chipselect, write_n, reset_n : in std_logic;
    in_compas : in std_logic;
    wridata : in std_logic_vector (31 downto 0);
    readdata : out std_logic_vector (31 downto 0);
    address: std_logic_vector (1 downto 0)
  );
END entity;

ARCHITECTURE arch_avalon_compas OF compas_avalon IS

COMPONENT gestion_compas
  PORT(in_pwm_compas : IN STD_LOGIC;
       clk_50M : IN STD_LOGIC;
       raz_n : IN STD_LOGIC;
       continu : IN STD_LOGIC;
       start_stop : IN STD_LOGIC;
       data_valid : OUT STD_LOGIC;
       data_compas : OUT STD_LOGIC_VECTOR(8 DOWNTO 0)
  );
END COMPONENT;

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

BEGIN

--écriture registres

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 <= wridata(0);
      continu <= wridata(1);
      start_stop <= wridata(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"000000"&"0"&start_stop&continu&raz_n;
when "10" => readdata <= X"00000000"&"00"&data_valid&data_compas;
when others => readdata <= (others => '0');
end case;
end process process_Read;

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

END arch_avalon_compas;

```

Figure 18 : Code de Avalon_compas.vhd

On crée par la suite notre hardware

Use	C...	Name	Description	Export	Clock	Base
<input checked="" type="checkbox"/>		clk_0	Clock Source		<i>exported</i>	
<input checked="" type="checkbox"/>		CPU	Nios II Processor		clk_0	0x0001_0800
<input checked="" type="checkbox"/>		RAM	On-Chip Memory (RAM or ROM) Intel ...		clk_0	0x0000_8000
<input checked="" type="checkbox"/>		LEDS	PIO (Parallel I/O) Intel FPGA IP			
		clk	Clock Input	<i>Double-click to export</i>	clk_0	
		reset	Reset Input	<i>Double-click to export</i>	[clk]	
		s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
		external_connection	Conduit	leds_external_connection		0x0001_1050
<input checked="" type="checkbox"/>		JTAG_UART	JTAG UART Intel FPGA IP			
		clk	Clock Input	<i>Double-click to export</i>	clk_0	
		reset	Reset Input	<i>Double-click to export</i>	[clk]	
		avalon_jtag_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
		irq	Interrupt Sender	<i>Double-click to export</i>	[clk]	
<input checked="" type="checkbox"/>		SYST_ID	System ID Peripheral Intel FPGA IP			
		clk	Clock Input	<i>Double-click to export</i>	clk_0	
		reset	Reset Input	<i>Double-click to export</i>	[clk]	
		control_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
<input checked="" type="checkbox"/>		BOUTONS	PIO (Parallel I/O) Intel FPGA IP			
		clk	Clock Input	<i>Double-click to export</i>	clk_0	
		reset	Reset Input	<i>Double-click to export</i>	[clk]	
		s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clk]	
		external_connection	Conduit	boutons_external_connection		0x0001_1040
<input checked="" type="checkbox"/>		avalon_pwm_0	avalon_pwm			
		clock	Clock Input	<i>Double-click to export</i>	clk_0	
		avalon_slave_0	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clock]	
		reset	Reset Input	<i>Double-click to export</i>	[clock]	
		conduit_end	Conduit	avalon_pwm_0_conduit_end	[clock]	
<input checked="" type="checkbox"/>		av_compas_0	av_compas			
		clock	Clock Input	<i>Double-click to export</i>	clk_0	
		avalon_slave_0	Avalon Memory Mapped Slave	<i>Double-click to export</i>	[clock]	
		reset	Reset Input	<i>Double-click to export</i>	[clock]	

Figure 19 : Création NIOS

Que l'on va par la suite générer.

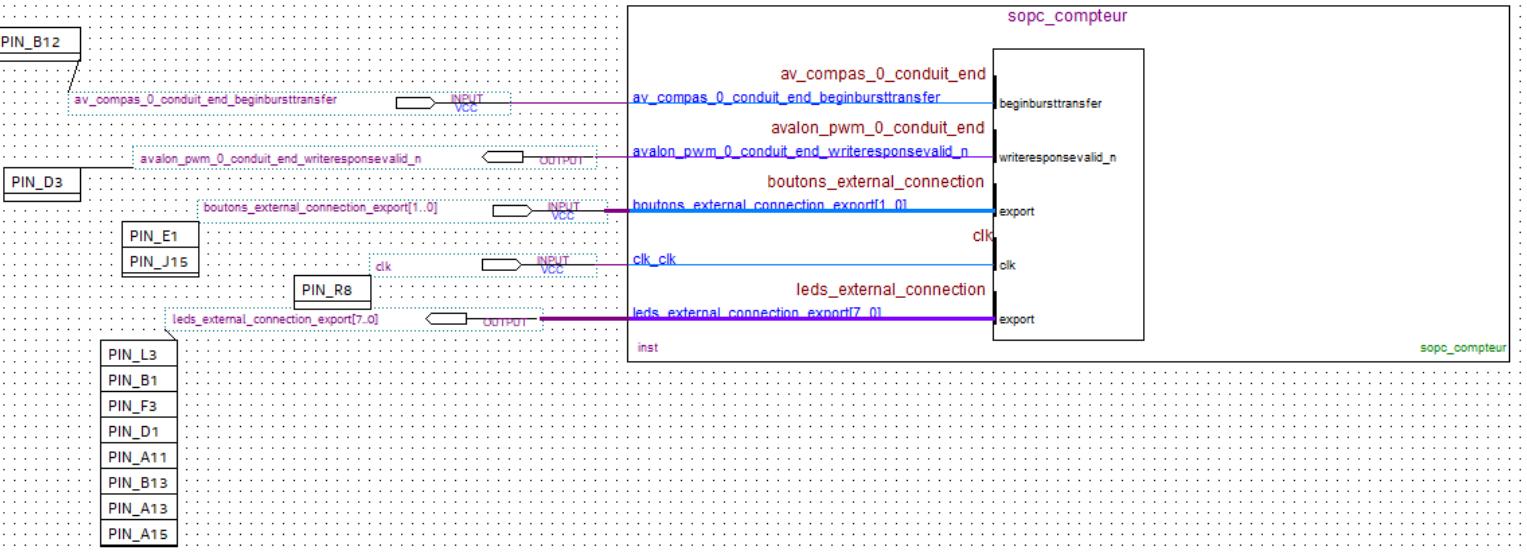


Figure 20 : Bloc du SOPC

Pour finir on fait l'implémentation du code C

```

#define duty (unsigned int *) (AVALON_PWM_0_BASE + 4)
#define control (unsigned int *) (AVALON_PWM_0_BASE + 8)

unsigned int a;
int main()
{
    *freq = 0x0400; // divise clk par 1024
    *duty = 0x0200; // RC = 50
    *control = 0x0003;
    alt_putstr("Salut ext!\n"); // test si communication OK

    while (1)
    {
        alt_putstr("Salut int!\n"); // test si communication OK
        /*leds = *boutons;
        a = *boutons & 3;
        printf("boutons = %d \n", a);
        usleep(1000000);
        switch(a)
        {
            case 0 : *leds=0; break;
            case 1 : *leds=0; break;
            case 2 : break;
            case 3 : *leds=*leds + 1; break;
            default : *leds = 0; break;
        }
        */
        return 0;
    }
}

```

Figure 21 : Code C test_avalon_pwm

Cependant, lors de l'intégration de la fonction compas pour le développement du code C il y a eu des erreurs de mapping entre la carte et le logiciel ne satisfaisant pas ainsi la transmission de la fonction vers la carte.

V - Conclusion

Pour conclure, ce projet nous a permis de réaliser les différentes étapes de développement d'un système embarqué allant de l'analyse fonctionnelle à l'intégration des composants sur le FPGA. Bien que nous ne soyons pas aller au bout du cahier des charges, cela a été très enrichissant de prendre en main différents outils numériques tel que le langage VHDL nécessaire à la description du comportement et de la structure des systèmes électroniques, mais aussi du logiciel Quartus servant à la conception des FPGA mais aussi à leur mise en œuvre.