

# Synthèse et mise en œuvre des systèmes

Bureau d'étude BE VHDL M2 SME



Université Paul Sabatier Toulouse III  
HADJ HASSEN Yosri & HAMICI Mohamed Anis

## Table des matières

Table des figures .....	2
I. Introduction .....	3
II. Analyse des besoins.....	4
III. Réalisations du BE : Pilote de Barre Franche.....	5
1. Mise en œuvre de la gestion du cap.....	5
2. Analyse fonctionnelle.....	6
3. Implémentation et simulation .....	6
IV. Réalisations du BE : Gestion commandes et indications barreur « IHM » .....	8
1. Mise en œuvre de l'IHM .....	8
2. Analyse fonctionnelle.....	8
3. Implémentation et simulation .....	12
V. Réalisations du BE : SOPC 1 <sup>er</sup> version (Intégration du PWM) .....	13
1. Mise en œuvre .....	13
2. Analyse fonctionnelle.....	13
3. Implémentation et simulation .....	13
3. Tests et simulation de la fonction SOCP .....	14
VI. Réalisations du BE : SOPC 2 <sup>ème</sup> version (Intégration du compas) .....	16
1. Mise en œuvre .....	16
2. Analyse fonctionnelle :.....	16
3. Implémentation et simulation .....	17
2. Tests et simulation de la fonction compas en SOPC .....	19
Conclusion.....	21

## Table des figures

Figure 1: Présentation de pilote de barre franche.....	3
Figure 2: Les blocs de Pilote de barre franche .....	4
Figure 3: Diagramme contexte du système .....	4
Figure 4: Diagramme du signal PWM Compas .....	5
Figure 5: Analyse Fonctionnelle du compas .....	6
Figure 6: Implémentation de la fonction Compas sur Quartus.....	7
Figure 7: Simulation compas.....	7
Figure 8: Schéma fonctionnel de IHM .....	8
Figure 9: Schéma fonctionnel détaillé.....	9
Figure 10: Machine à état IHM-gestion BP .....	9
Figure 11: Machine à état gestion des LEDs.....	11
Figure 12: Machine à état Gestion buzzer.....	11
Figure 13: Analyse Fonctionnelle du SOPC 1er version.....	13
Figure 14: Fonction sur SOPC Builder .....	13
Figure 15: Code des registres de communications.....	14
Figure 16: Bloc SOPC sur Quartus .....	14
Figure 17: Code SOPC test .....	15
Figure 18: Visualisation sur l'oscilloscope .....	15
Figure 19: Le composant Avalon compas.....	16
Figure 20: Le composant Avalon compas.....	16
Figure 21:Fonction compas sur SOPC Builder .....	17
Figure 22: Code des registres de communications pour le compas.....	18
Figure 23: Bloc SOPC sur Quartus (avec compas) .....	18
Figure 24: code de résolution du problème de mapping .....	19
Figure 25:Code c pour le test et les résultats avec le NIOS .....	20
Tableau 1: mode fonctionnement du code_fonction.....	8
Tableau 2: Structure des registres de l'interface Avalon .....	19

## I. Introduction

Notre but principal, dans le cadre de notre unité d'enseignement "Synthèse et Mise en Œuvre de Systèmes", est de développer un pilote de barre franche sous la forme d'une puce programmable SOPC (System On Programmable Chip). Cela implique l'utilisation du langage VHDL pour décrire le système, basé sur une analyse détaillée des spécifications et une structuration fonctionnelle du système. Nous allons concevoir des circuits d'interfaces numériques en VHDL pour le simuler et le valider sur une maquette. Ensuite, nous établirons des connexions avec les bus de microprocesseurs tels que NIOS, Altéra et Avalon pour vérifier la fonctionnalité du SOPC par des manipulations spécifiques.

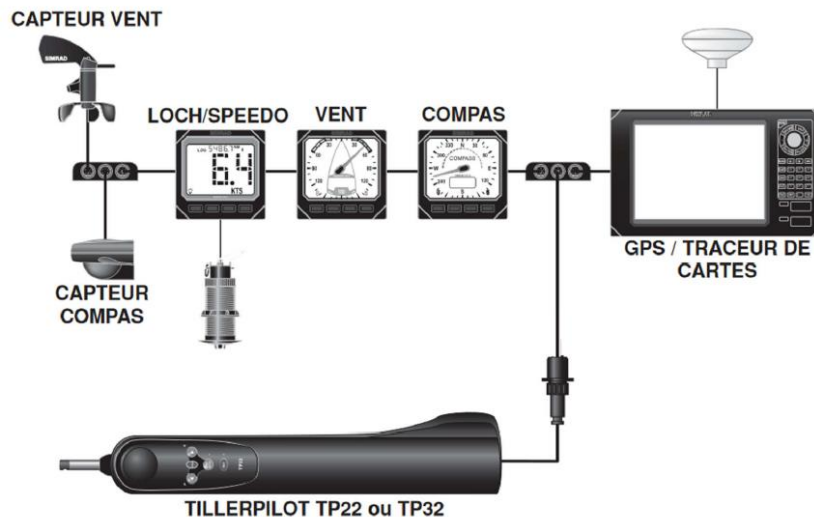


Figure 1: Présentation de pilote de barre franche.

Un pilote automatique pour voilier est un dispositif électrique ou hydraulique conçu pour maintenir le cap d'un voilier sans intervention humaine. Il s'avère d'une grande utilité pour les navigateurs solitaires ou en équipage réduit. Ce pilote se compose essentiellement de trois éléments : un compas, une unité électronique et une unité de puissance. Dans les versions récentes, le compas, électronique, transmet en continu au système de traitement le cap actuel du bateau. L'unité de traitement, quant à elle, définit le cap à suivre. En permanence, elle compare ces deux caps et si une différence est détectée, elle ordonne à l'unité de puissance d'intervenir sur la barre pour réaligner le bateau sur son cap désiré.

Pour les pilotes de barre franche, l'unité de puissance est souvent un vérin linéaire. Ce vérin, installé entre le banc du cockpit et la barre, réagit aux variations de cap en agissant sur la barre en conséquence.

Ce projet d'étude porte précisément sur la conception d'un pilote de barre franche pour un voilier, visant à automatiser sa navigation. Ce système englobe une commande de pilotage équipée d'un vérin. Les commandes de cette unité permettent de contrôler l'extension et la rétraction du vérin, influençant ainsi la barre franche du voilier. Ces commandes permettent également de choisir différents modes de navigation pour le voilier. En parallèle, ce système est connecté à une boussole (compas), un GPS avec une interface NMEA, une girouette pour déterminer la direction du vent et un anémomètre pour mesurer sa vitesse.

Toutes ces données seront utilisées pour calculer la trajectoire du voilier et lui permettre de naviguer selon divers modes tels que le Conservateur d'Allure, le mode Automatique, et bien d'autres.

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

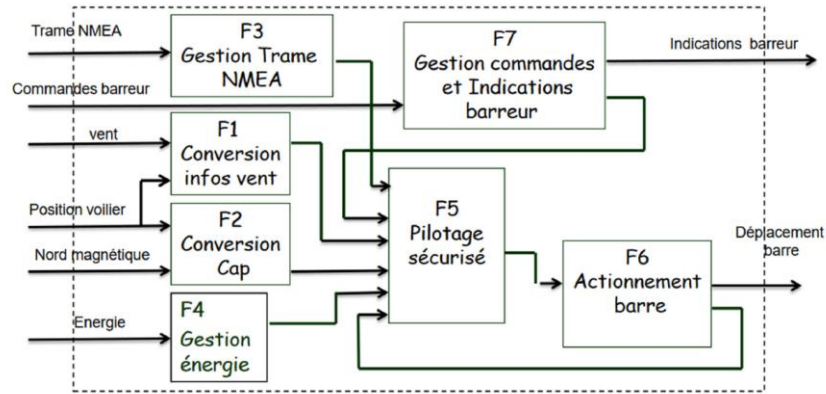


Figure 2: Les blocs de Pilote de barre franche

## II. Analyse des besoins

Pour répondre aux besoins de la barre franche, le système à développer comprendra différentes fonctions et flux d'informations essentiels pour son fonctionnement.

Il comportera :

- Une fonction dédiée à la lecture de la vitesse du vent (0-250 km/h), en interprétant la sortie logique de fréquence variable (0 à 250 Hz) de l'anémomètre.
- Une fonction de génération de signal PWM, intégrée ultérieurement dans le SOPC du FPGA, pour produire un signal utilisable via le Bus Avalon.
- Une fonction de gestion du vérin, responsable du contrôle de la barre franche.
- Une fonction exploitant un compas pour obtenir les mesures d'angle horizontales du voilier, déterminant ainsi sa direction.
- Une fonction de gestion de l'interface Homme-Système, comprenant divers boutons (Bâbord, Tribord, Auto/Manuel), des LED et un buzzer.
- L'intégration d'un MCU (Microcontrôleur) dans le FPGA via l'outil SOPC d'Altera. Ce MCU assurera le traitement et l'affichage des différentes variables du projet telles que le cap du voilier, la position du vérin, les données GPS, la gestion du signal PWM et la vitesse du vent.

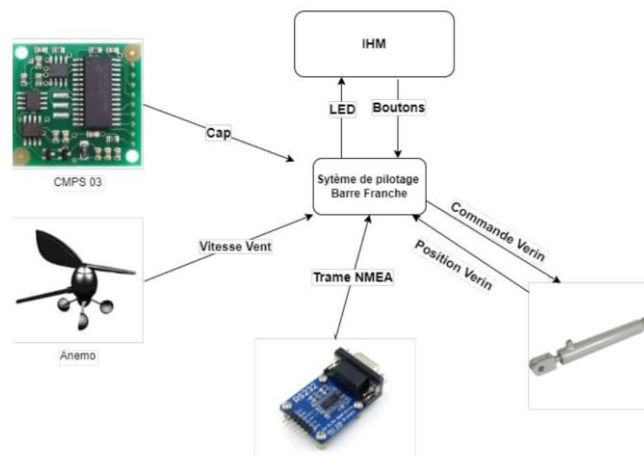


Figure 3: Diagramme contexte du système

### III. Réalisations du BE : Pilote de Barre Franche

#### 1. Mise en œuvre de la gestion du cap

Dans cette partie nous avons dû faire un module compas basé sur le module gestion compas pour boussole CMPS03 ou CMPS10

Avec les spécifications suivantes :

#### Entrées :

**Clk\_50M** : horloge 50MHz

**Raz\_n** : reset actif à 0 → initialise le circuit

**in\_pwm\_compas** : signal PWM de la boussole, durée varie de 1ms à 36,9ms

**Continu** : si=0 : mode monocoup, si=1 : mode continu -- en mode continu la donnée est rafraîchie toutes les secondes

**Start\_stop** : en monocoup si=1 démarre une acquisition, si =0 -- remet à 0 le signal data\_valid

#### Sorties :

**Data\_valid** : =1 lorsqu'une mesure est valide -- est remis à 0 quand start\_stop passe à 0

**Data\_compas** : valeur du cap réel exprimé en degré codé sur 9 bits

Ce module de boussole a été spécialement conçu pour être utilisé dans les robots comme aide à la navigation. L'objectif était de produire un nombre unique pour représenter la direction vers laquelle le robot est orienté. La boussole utilise le capteur de champ magnétique Philips KMZ51, qui est suffisamment sensible pour détecter le champ magnétique terrestre.

Nous nous intéressons à la sortie donnée par ce module sous la forme d'un signal PWM dont la largeur positive de l'impulsion représente l'angle. La largeur de l'impulsion varie de 1mS (0°) à 36,99mS (359,9°) - soit 100uS/° avec un décalage de +1mS. Le signal est en état bas pendant 65 ms entre les impulsions, de sorte que le temps de cycle est de 65 ms + la largeur de l'impulsion.

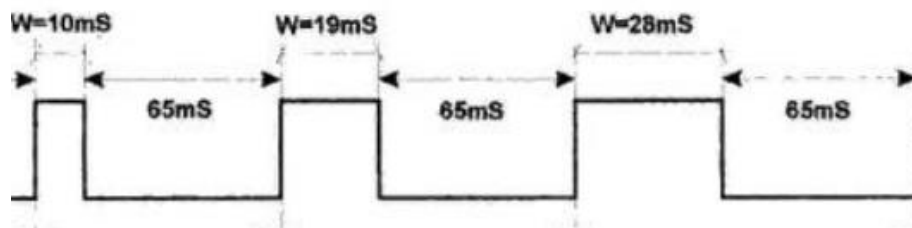


Figure 4: Diagramme du signal PWM Compas

## 2. Analyse fonctionnelle

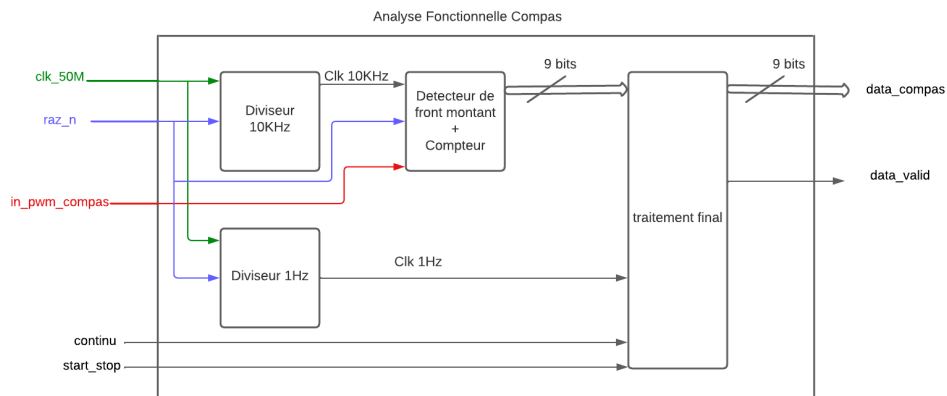


Figure 5: Analyse Fonctionnelle du compas

**Diviseur 10Khz :** Ce diviseur prend l'horloge 50Mhz native de nos cartes DE2/DE0 NANO et la divise pour créer une sortie d'horloge 10Khz, ce qui signifie que la période produite est de 100us, ce qui est conforme aux 100us/degré donnés par le compas, ensuite nous utilisons ce diviseur pour incrémenter notre compteur d'état haut pour obtenir sa durée dans les unités commodes.

**Diviseur 1Hz :** Ce diviseur prend l'horloge 50Mhz native de nos cartes DE2/DE0 NANO et la divise pour créer une sortie d'horloge 1 Hz donc 1 seconde. Il est utilisé pour mettre en œuvre le mode continu qui génère des mesures toutes les secondes.

**Détecteur de front montant/Compteur :** Ce bloc contient 2 fonctionnalités

1- Détecteur de front montant : pour repérer le moment où notre impulsion démarre et initier le comptage, puis arrêter et mémoriser la valeur du comptage lorsque le signal passe au niveau bas.

2- le compteur démarre lorsque nous détectons un front haut, il s'incrémente tous les cycles d'horloge de 10 KHz, c'est à dire tous les 100us, ce qui signifie +1 degré à chaque fois que le compteur s'incrémente. Enfin cela donne une sortie (la valeur du comptage) codée sur 9 bits.

**Traitement final :** Ce bloc contient la logique de la fonction attendue pour la boussole, l'initiation du mode continu, la vérification de data\_valid et la logique de start\_stop.

## 3. Implémentation et simulation

Après avoir effectué l'analyse fonctionnelle de notre circuit, nous avons procédé à son implémentation dans Quartus en utilisant des blocs en VHDL et en suivant le code VHDL tel que représenté dans la figure ci-dessous :



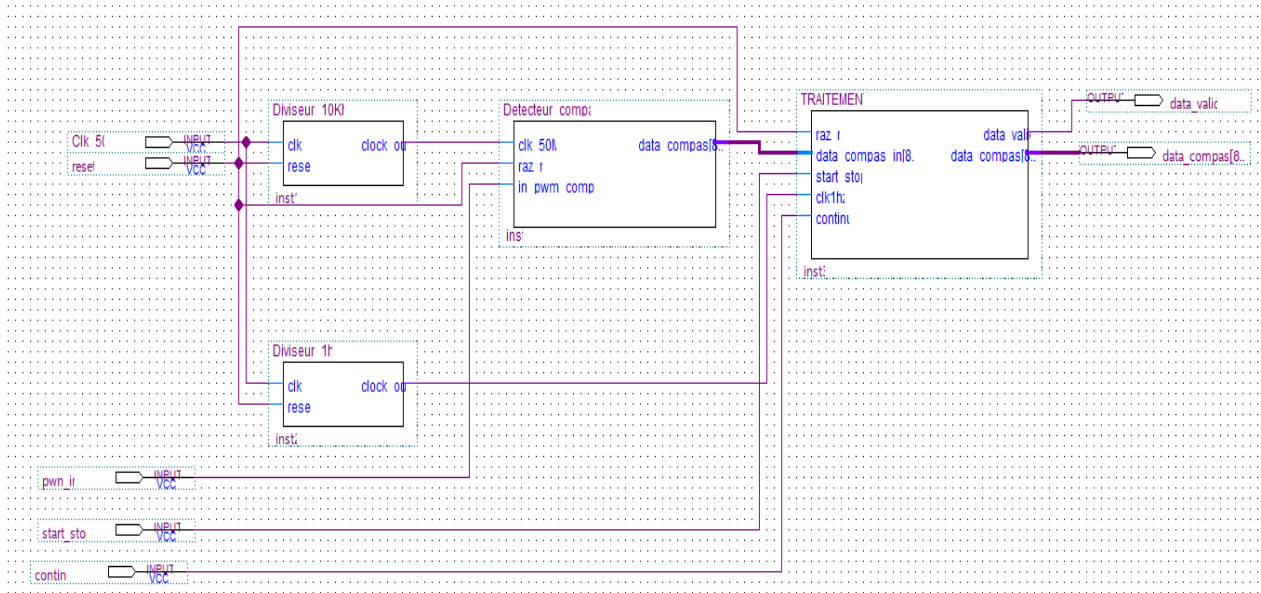


Figure 6: Implémentation de la fonction Compas sur Quartus

L'échelle de temps a été réduite pour que le simulateur soit fiable sans prendre trop de temps, afin de garantir un fonctionnement correct.

Ensuite nous avons utilisé notre carte DE2, nous avons attribué des pins à nos entrées et des leds sur les 9 bits de la sortie, généré un signal sur notre GBF avec offset pour avoir 0 à +5v pour nos états on off, 65ms + 20 ms => résultat lu sur les leds 200 degrés

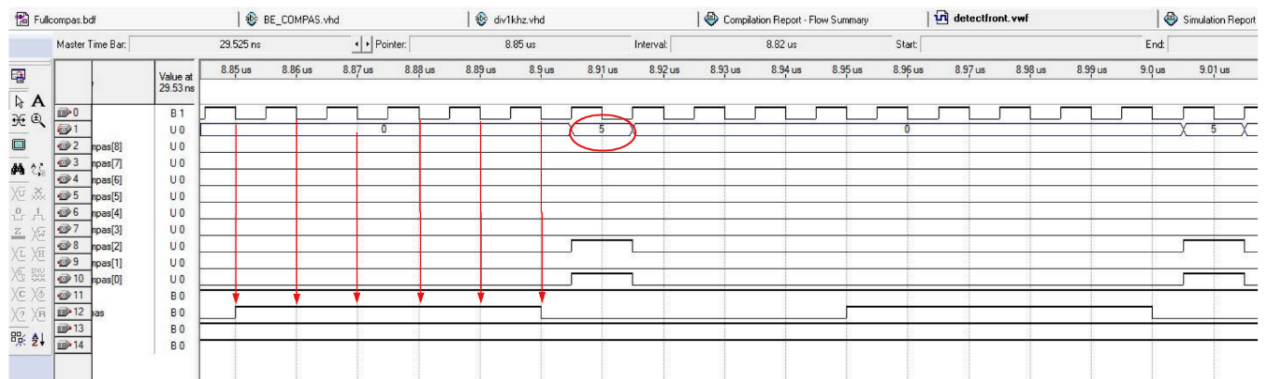


Figure 7: Simulation compas



## IV. Réalisations du BE : Gestion commandes et indications barreur « IHM »

### 1. Mise en œuvre de l'IHM

Dans le cadre de notre projet, nous souhaitons intégrer une interface homme-machine (IHM) comprenant trois boutons poussoirs, nommés "Bâbord", "Tribord", et "STBY/Auto". Chacun de ces boutons aura une fonction spécifique, dont nous expliquerons ultérieurement. En parallèle, nous allons mettre en place des LEDs et un buzzer pour fournir des indications visuelles et sonores.

L'IHM sera conçue pour permettre au pilote à barre franche d'accéder à divers modes de fonctionnement en utilisant les boutons poussoirs. Après avoir effectué des manipulations spécifiques avec ces boutons, l'IHM générera un signal de sortie appelé "code\_function", qui sera un code sur 4 bits représentant 8 modes de fonctionnement distincts.

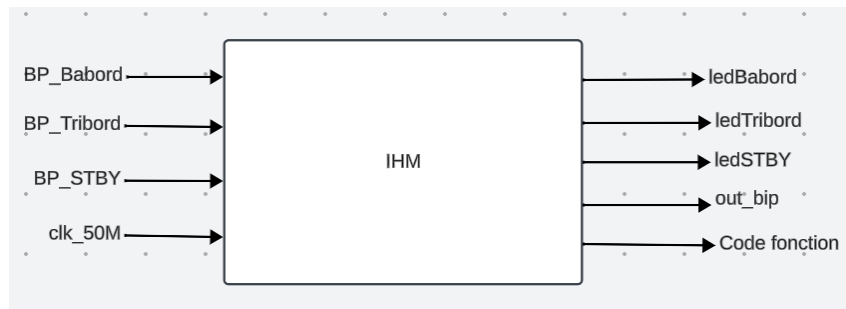


Figure 8: Schéma fonctionnel de IHM

Le code\_function qui présente les 8 mode de fonctionnement sont les suivantes :

Code fonction	Action
0000	Pas d'action, Pilote en veille
0001	Mode manuel action vérin babord
0010	Mode manuel action vérin tribord
0011	Mode pilote automatique/cap
0100	Incrément de 1° consigne de cap
0101	Incrément de 10° consigne de cap
0111	Décrément de 1° consigne de cap
0110	Décrément de 10° consigne de cap

Tableau 1: mode fonctionnement du code\_function

### 2. Analyse fonctionnelle

Pour concrétiser la fonctionnalité et faciliter la mise en œuvre, nous avons subdivisé la fonction en sous-fonctions, notamment pour la gestion des boutons-poussoirs, la gestion des LEDs, et la gestion du buzzer (out\_bip). Le schéma fonctionnel ci-dessous présente ces différentes sous-fonctions, qui seront expliquées en détail juste après le schéma. Toutes ces sous-fonctions seront regroupées dans un même composant global pour concrétiser la fonctionnalité souhaitée.

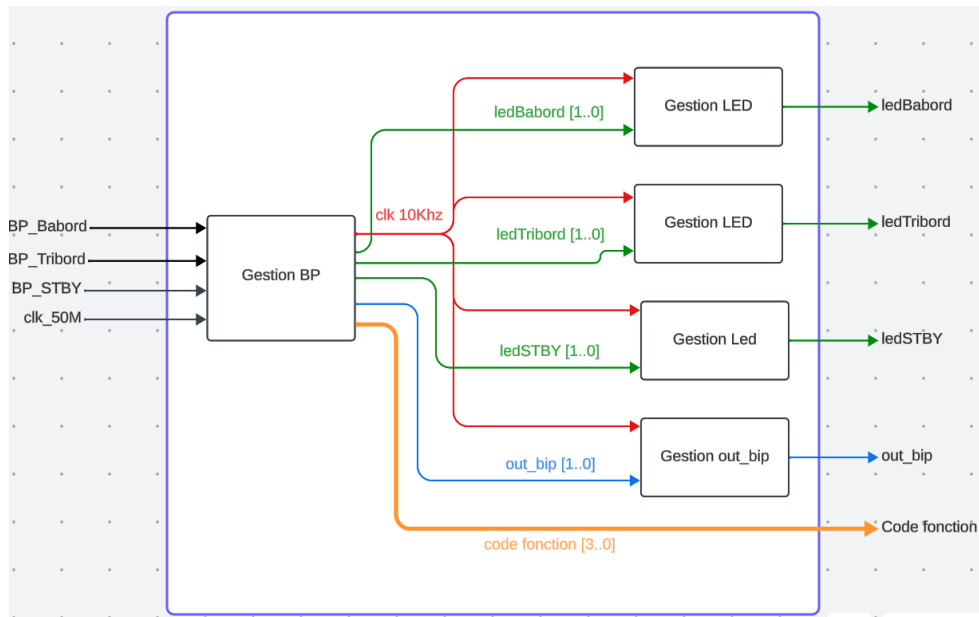


Figure 9: Schéma fonctionnel détaillé

La machine à états suivante illustre le fonctionnement de notre IHM en détaillant les différents modes et les divers appuis sur les boutons-poussoirs. Cette représentation met en évidence les transitions entre les états en réponse aux actions des utilisateurs.

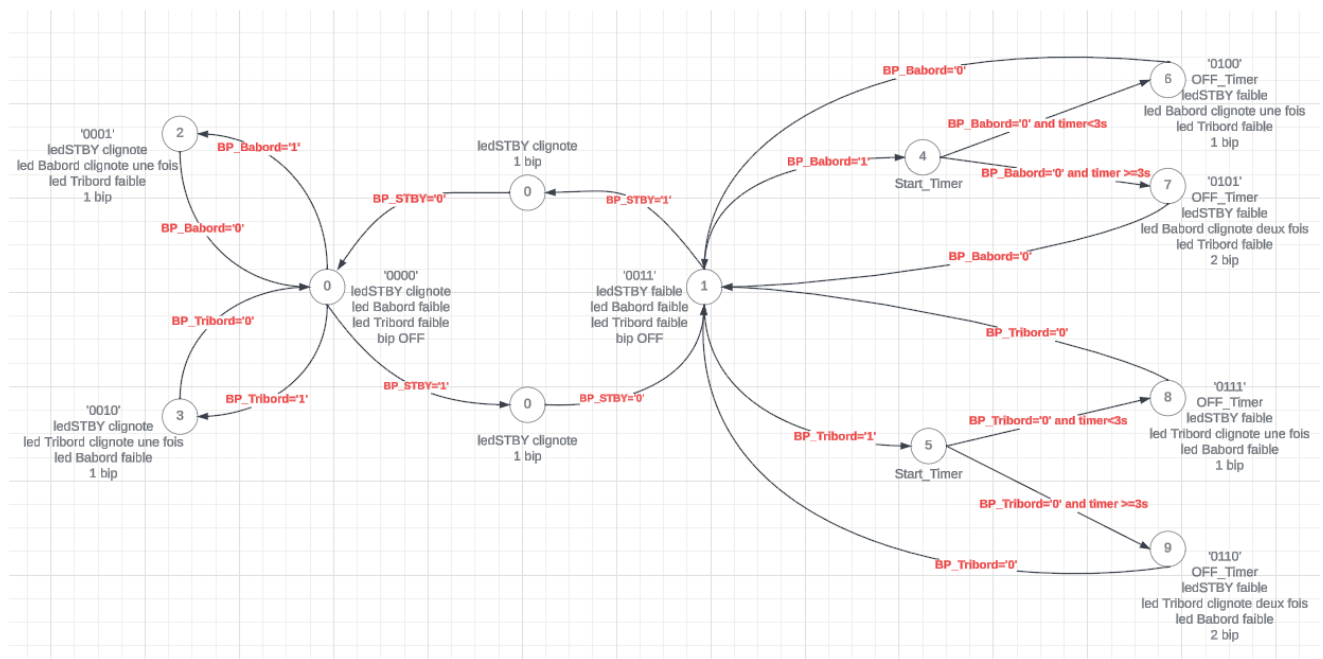


Figure 10: Machine à état IHM-gestion BP

### Gestion des boutons poussoirs :

La gestion des boutons poussoirs constitue une fonction cruciale de notre système, permettant le contrôle et la transition entre différents modes de fonctionnement du pilote à barre franche. Nous mettons en place trois BP : BP\_Babord, BP\_Tribord et BP\_STBY (Standby/Auto). Chacun de ces boutons est associé à des fonctionnalités spécifiques, et leur interaction génère un code fonction en sortie, déterminant le mode actuel du pilote.

Le BP\_STBY est utilisé pour basculer entre le mode Manuel et Auto. Un appui simple sur ce bouton permet de passer d'un mode à l'autre, avec le code fonction "0000" en mode Manuel et "0011" en mode Auto.

En mode Manuel, les BP\_Babord et BP\_Tribord sont utilisés pour des actions spécifiques. BP\_Babord génère le code fonction "0001" à chaque appui simple, indiquant une action particulière. De même, BP\_Tribord génère le code fonction "0010". Après chaque front descendant d'un appui sur ces boutons, la fonction revient automatiquement au mode Manuel avec le code fonction "0000".

En mode Auto, les BP\_Babord et BP\_Tribord sont utilisés de manière différente. Lorsqu'un de ces boutons est pressé, la fonction active un timer pour mesurer la durée d'appui. Si l'appui est maintenu pendant moins de 3 secondes, le code interprète cela comme un appui court et génère un code fonction spécifique. Si l'appui est maintenu pendant plus de 3 secondes, le système considère cela comme un appui long, et le code fonction résultant est différent. Après chaque front descendant d'un appui sur ces boutons, le système revient automatiquement au mode Auto avec le code fonction "0011".

Par conséquent, en mode Auto :

- Appui court sur BP\_Babord génère le code fonction "0100".
- Appui long sur BP\_Babord génère le code fonction "0101".
- Appui court sur BP\_Tribord génère le code fonction "0111".
- Appui long sur BP\_Tribord génère le code fonction "0110".

A chaque transition d'état ou action, chaque LED est associée à une sortie de 2 bits qui reflète son état dans le contexte actuel de la machine à états. Par exemple, en cas d'appui long sur BP\_Babord, la LED STBY et LED Tribord s'allument faiblement et la LED Babord clignote deux fois. En parallèle, un signal de 2 bits représente la sortie pour le buzzer (out\_bip). Ces signaux, exprimés en 2 bits, sont ensuite acheminés vers un autre bloc pour le traitement des différents états associés.

### Gestion des LEDs :

Pour gérer l'allumage des LEDs, un composant dédié a été créé. Il prend en entrée un signal d'horloge de 10 kHz, assurant une synchronisation avec le fonctionnement global de la barre franche, ainsi qu'un signal sur 2 bits décrivant les états des LEDs. Le signal de commande des LEDs provient de bloc de gestion des boutons. En sortie, le composant génère un signal qui dirige directement l'allumage des LEDs.

À chaque état dans le bloc de gestion des boutons-poussoirs, une sortie spécifique est produite pour chaque LED, représentant un signal sur deux bits. Ce signal est ensuite interprété dans le composant afin de régir l'allumage des LEDs. Les modes d'allumage des LEDs sont établis comme suit :

- 00 : LED éteinte
- 01 : LED clignote une fois pendant 1 seconde
- 10 : LED clignote deux fois, chaque clignotement durant 1 seconde
- 11 : LED allumée faiblement

La fonction inclut un timer qui supervise le clignotement des LEDs. Son opération est élémentaire :

Lorsqu'une LED doit clignoter une ou deux fois, le timer est enclenché. Une fois qu'il atteint 1 seconde, il éteint la LED et revient à son état initial. Pour le cas de deux clignotements, le timer éteint la LED pendant 1 seconde, puis la rallume pendant 1 seconde avant de la rééteindre. Le schéma suivant illustre la machine à état de la gestion des LEDs.

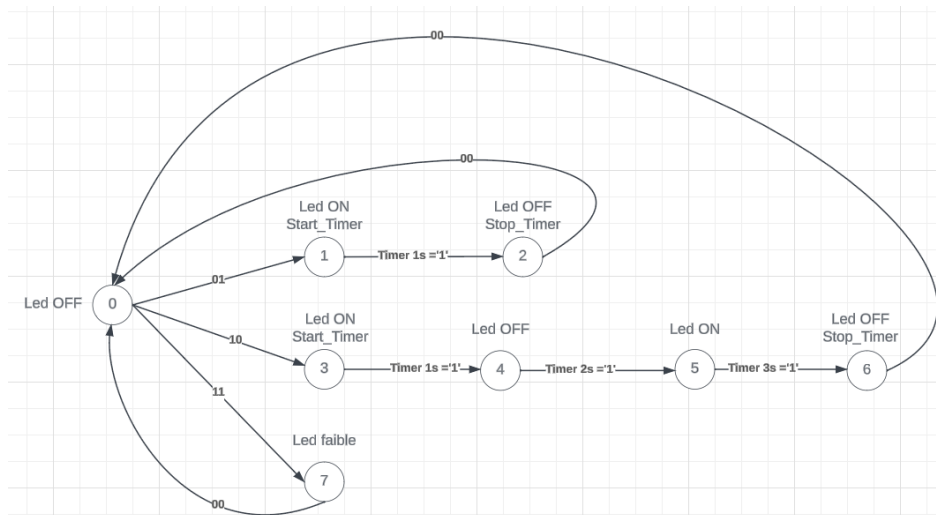


Figure 11: Machine à état gestion des LEDs

### Gestion buzzer :

Pour orchestrer l'activation du buzzer, un composant distinct a été élaboré, fonctionnant de manière similaire à la gestion des LEDs. Ce composant utilise un signal d'horloge de 10 kHz en entrée, garantissant une synchronisation avec le reste du système de barre franche. Il prend également en compte un signal sur 2 bits qui décrit l'état du buzzer. Le signal de commande du buzzer provient aussi du bloc de gestion des boutons.

Les modes de fonctionnement du buzzer sont définis comme suit :

- 00 : Buzzer éteint
- 01 : Buzzer émet un bip unique, d'une durée de 1 seconde
- 10 : Buzzer émet deux bips, chacun durant 1 seconde

Le composant intègre un timer qui régule la génération des bips. Son fonctionnement est similaire à celui de la gestion des LEDs.

En sortie, le composant génère un signal out\_bip qui contrôle directement le buzzer.

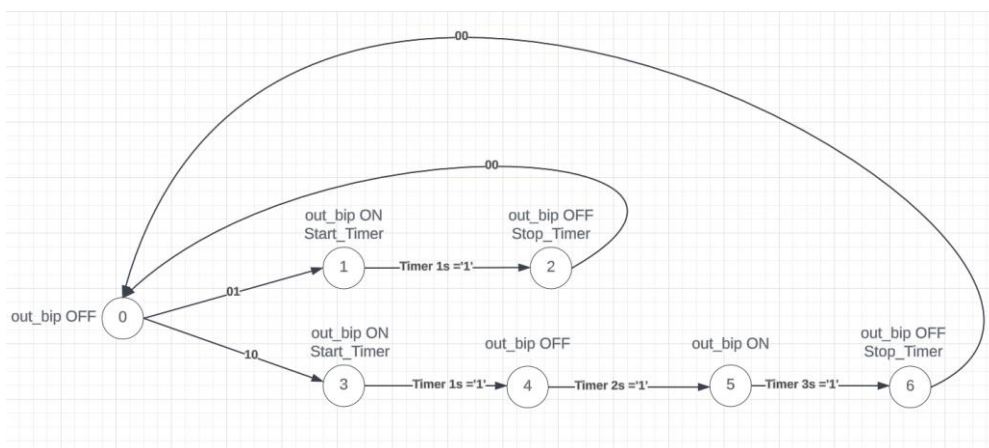


Figure 12: Machine à état Gestion buzzer

### 3. Implémentation et simulation

Dans le cadre de l'implémentation et de la simulation, nous avons élaboré le code pour la gestion des boutons poussoirs, abordant la logique nécessaire pour détecter les appuis simples et longs sur les boutons Bâbord, Tribord et STBY/Auto. Cependant, le code actuel ne parvient pas à répondre de manière satisfaisante aux différentes séquences d'appuis sur les boutons. L'objectif est de développer un code qui assure une gestion fluide des différents modes, génère correctement le code fonction et contrôle l'allumage des LEDs et du buzzer en fonction des événements déclenchés par les boutons poussoirs.

## V. Réalisations du BE : SOPC 1<sup>er</sup> version (Intégration du PWM)

### 1. Mise en œuvre

Le microcontrôleur conçu est basé sur le cœur Nios II, en suivant les spécifications du cahier des charges. Avec une mémoire RAM on-chip de 20 ko ajoutés. Deux PIO (Parallel Inputs Outputs) sont intégrés, l'un pour les boutons en entrée (2 bits) et l'autre pour les LEDs en sortie (8 bits). Un composant JTAG UART est inclus pour la communication avec le PC hôte et le débogage du programme, avec une assignation d'interruption. Un composant SysId est ajouté pour attribuer un numéro d'identification au système, renforçant la sécurité. Enfin, en générant le fichier VHDL, le SOPC (System on a Programmable Chip) est créé avec toutes les configurations définies

### 2. Analyse fonctionnelle

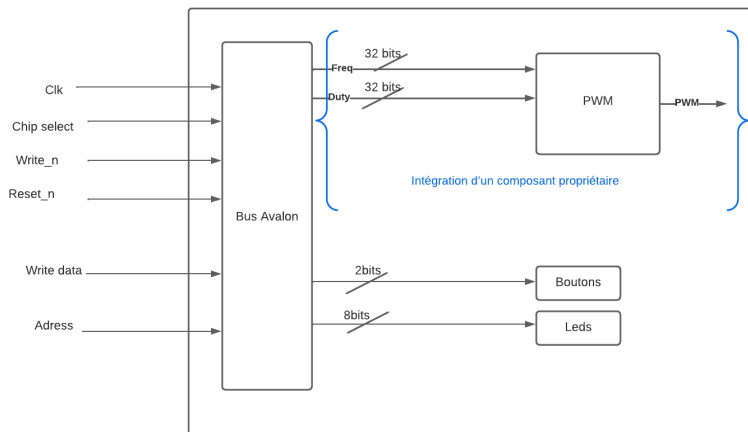


Figure 13: Analyse Fonctionnelle du SOPC 1er version

### 3. Implémentation et simulation

Pour transférer le circuit sur la carte, nous avons employé le SOPC Builder pour élaborer le microprocesseur et les divers périphériques en intégrant l'Avalon PWM, conformément aux indications figurant dans les schémas ci-dessous. Cela nous a permis de tester notre SOPC et de préparer l'intégration du même modèle dans les fonctions à réaliser au cours de notre BE

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		<b>clk_0</b> clk_in clk_reset clk clk_reset	Clock Source Clock Input Reset Input Clock Output Reset Output	clk Double-click to export Double-click to export Double-click to export	exported clk_0			
<input checked="" type="checkbox"/>		<b>RAM</b> clk1 s1 reset1	On-Chip Memory (RAM or ROM) Intel ... Clock Input Avalon Memory Mapped Slave Reset Input	Double-click to export Double-click to export Double-click to export	clk_0 [clk1] [clk1]	# 0x0000_8000	0x0000_cfff	
<input checked="" type="checkbox"/>		<b>Buttons_IO</b> clk reset s1 external_connection	PIO (Parallel I/O) Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped Slave Conduit	Double-click to export Double-click to export Double-click to export buttons_io_external_co...	clk_0 [clk] [clk]	# 0x0001_1040	0x0001_104f	
<input checked="" type="checkbox"/>		<b>LEDS_IO</b> clk reset s1 external_connection	PIO (Parallel I/O) Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped Slave Conduit	Double-click to export Double-click to export Double-click to export leds_io_external_conne...	clk_0 [clk] [clk]	# 0x0001_1030	0x0001_103f	
<input checked="" type="checkbox"/>		<b>JTAG_uart_0</b> clk reset avalon_jtag_slave irq	JTAG UART Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped Slave Interrupt Sender	Double-click to export Double-click to export Double-click to export Double-click to export	clk_0 [clk] [clk] [clk]	# 0x0001_1078	0x0001_107f	
<input checked="" type="checkbox"/>		<b>sysid_qsys_0</b> clk reset control_slave	System ID Peripheral Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped Slave	Double-click to export Double-click to export Double-click to export	clk_0 [clk] [clk]	# 0x0001_1070	0x0001_1077	
<input checked="" type="checkbox"/>		<b>CPU</b> clk reset data_master instruction_master irq debug_reset_request debug_mem_slave custom_instruction_ma...	Nios II Processor Clock Input Reset Input Avalon Memory Mapped Master Avalon Memory Mapped Master Interrupt Receiver Reset Output Avalon Memory Mapped Slave Custom Instruction Master	Double-click to export Double-click to export Double-click to export Double-click to export Double-click to export Double-click to export Double-click to export	clk_0 [clk] [clk] [clk] [clk] [clk] [clk]		IRQ 0	IRQ 31
<input checked="" type="checkbox"/>		<b>Avalon_pwm_0</b> clk avalon_slave_0 reset conduit_end	Avalon_pwm Clock Input Avalon Memory Mapped Slave Reset Input Conduit	Double-click to export Double-click to export Double-click to export avalon_pwm_0_conduit...	clk_0 [clock] [clock]	# 0x0001_1050	0x0001_105f	

Figure 14: Fonction sur SOPC Builder

Après avoir intégré le code et la logique pour le PWM avec 32 bits de service et de fréquence, nous avons besoin d'un code des registres pour établir la communication et récupérer les valeurs, nous utilisons :

```
--écriture registres
process_write: process (clk, reset_n)
begin
    if reset_n = '0' then
        freq <= (others => '0');
        duty <= (others => '0');
        control <= (others => '0');
    elsif clk'event and clk = '1' then
        if chipselect = '1' and write_n = '0' then
            if address = "00" then
                freq <= writedata;
            end if;
            if address = "01" then
                duty <= writedata;
            end if;
            if address = "10" then
                control <= writedata (1 downto 0);
            end if;
        end if;
    end process;

-- lecture registres
process_Read: process (address, freq, duty, control)
begin
    case address is
        when "00" => readdata <= freq;
        when "01" => readdata <= duty;
        when "10" => readdata <= X"00000000"&"00"&control;
        when others => readdata <= (others => '0');
    end case;
end process process_Read;

end arch_avalon_pwm;
```

Figure 15: Code des registres de communications

Une fois notre système assemblé via SOPC Builder, avec l'intégration de l'Avalon PWM, des boutons et des LEDs, nous avons généré l'interface présentée dans la figure ci-dessous. Celle-ci représente le bloc SOPC actuel qui évoluera au fil de notre BE pour intégrer les fonctionnalités spécifiques à la Barre-Franche.

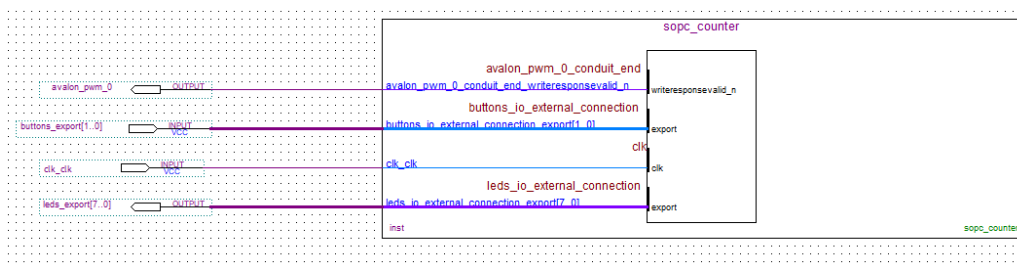


Figure 16: Bloc SOPC sur Quartus

### 3. Tests et simulation de la fonction SOCP

Notre objectif avec le code est de diviser la fréquence interne de 50 MHz par 1000, tout en maintenant un rapport cyclique de 50%. Après compilation et exécution du code sur notre SOPC, nous anticipons des résultats de 50 MHz/1024 avec un rapport cyclique de 50%, conformes à ce qui est affiché dans la visualisation sur un oscilloscope. Ces résultats correspondent aux attentes fixées.



```
#define freq (unsigned int *) AVALON_PWM_0_BASE
#define duty (unsigned int *) (AVALON_PWM_0_BASE + 4)
int main()
{
    ...
    *freq = 0x0400; // divide clk par 1024
    *duty = 0x0200; // RC = 50%
    ...
}
```

Figure 17: Code SOPC test

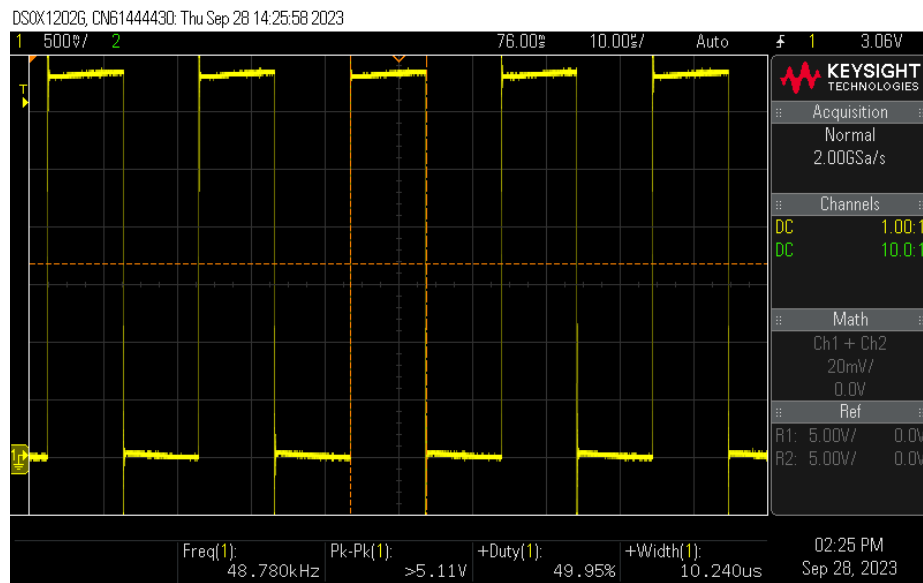


Figure 18: Visualisation sur l'oscilloscope

## VI. Réalisations du BE : SOPC 2<sup>ème</sup> version (Intégration du compas)

### 1. Mise en œuvre

Après avoir intégré avec succès une fonction sur le soc (pwm), il est temps d'ajouter notre module compas sur le microcontrôleur basé sur le cœur Nios II. Tout d'abord, nous prenons notre fichier bdf du compas et nous générons un fichier vhdl pour celui-ci (automatiquement par quartus), ce code initialise les blocs, établit les connexions entre eux en port map et rassemble le tout en une seule unité avec les entrées et sorties souhaitées, nous chargeons ce code dans notre composant sur platform designer" et analysons sa synthèse pour générer les signaux et interfaces créés.

### 2. Analyse fonctionnelle

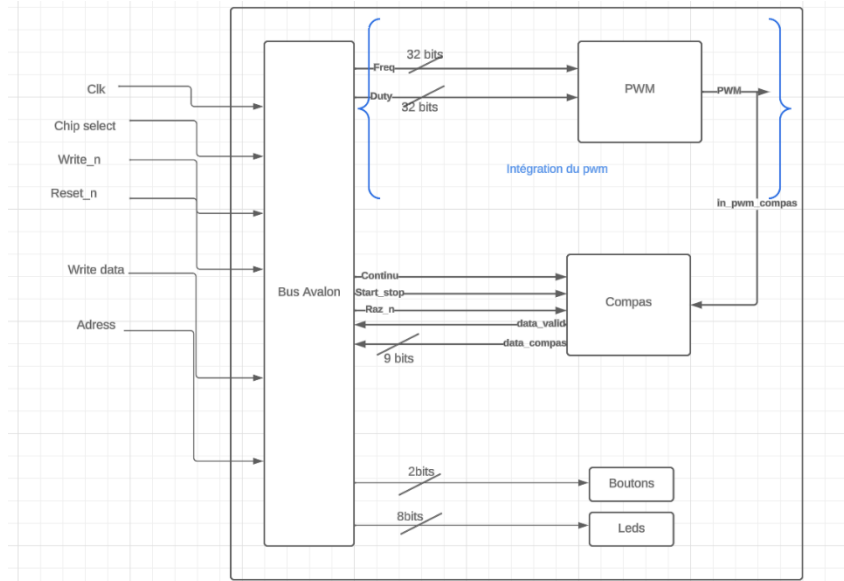


Figure 19: Le composant Avalon compas

Nous créons ensuite un nouveau signal à l'aide de "conduit end" et y plaçons notre pwm, qui servira d'entrée dans notre circuit généré, cette entrée contiendra l'impulsion pwm dont le module compas calculera le degré.

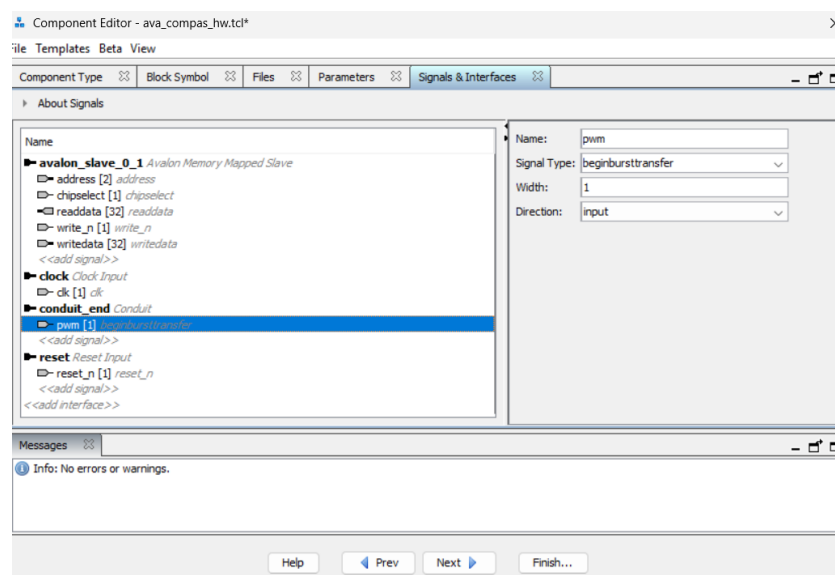


Figure 20: Le composant Avalon compas

### 3. Implémentation et simulation

Pour transférer le circuit sur la carte, nous avons employé le SOPC Builder pour élaborer le microprocesseur et les divers périphériques en intégrant l'Avalon PWM, conformément aux indications figurant dans les schémas ci-dessous. Cela nous a permis de tester notre SOPC et de préparer l'intégration du même modèle dans les fonctions à réaliser au cours de notre BE

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
		clk_0	Lock input	Double-click to export	clk_0			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk1]			
		reset1	Reset Input	Double-click to export	[clk1]			
<input checked="" type="checkbox"/>		<b>Buttons_IO</b>	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]			
		external_connection	Conduit	buttons_io_external_co...		0x0000_8000	0x0000_cfff	
<input checked="" type="checkbox"/>		<b>LEDS_IO</b>	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		s1	Avalon Memory Mapped Slave	Double-click to export	[clk]			
		external_connection	Conduit	leds_io_external_conne...		0x0001_1050	0x0001_105f	
<input checked="" type="checkbox"/>		<b>jtag_uart_0</b>	JTAG UART Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]			
		irq	Interrupt Sender	Double-click to export	[clk]	0x0001_1098	0x0001_109f	
<input checked="" type="checkbox"/>		<b>sysid_qsys_0</b>	System ID Peripheral Intel FPGA IP					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1090	0x0001_1097	
<input checked="" type="checkbox"/>		<b>CPU</b>	Nios II Processor					
		clk	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		data_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		instruction_master	Avalon Memory Mapped Master	Double-click to export	[clk]			
		irq	Interrupt Receiver	Double-click to export	[clk]			
		debug_reset_request	Reset Output	Double-click to export	[clk]			
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_0800	0x0001_0fff	
		custom_instruction_m...	Custom Instruction Master	Double-click to export	[clk]			
<input checked="" type="checkbox"/>		<b>new_component_0</b>	new_component					
		clock	Clock Input	Double-click to export	clk_0			
		avalon_slave_0	Avalon Memory Mapped Slave	Double-click to export	[clk]			
		reset	Reset Input	Double-click to export	[clk]	0x0001_1070	0x0001_107f	
		conduit_end	Conduit	new_component_0_con...	[clk]			
<input checked="" type="checkbox"/>		<b>ava_compas_0</b>	ava_compas					
		clock	Clock Input	Double-click to export	clk_0			
		reset	Reset Input	Double-click to export	[clk]			
		conduit_end	Conduit	ava_compas_0_conduit...	[clk]			
		avalon_slave_0_1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0001_1060	0x0001_106f	

Figure 21: Fonction compas sur SOPC Builder

Il est également important de double cliquer sur la section "export" du "conduit end" pour définir le nœud qui sera utilisé comme entrée ou sortie.

Comme nous l'avons fait précédemment, nous créons maintenant le code de lecture/écriture qui sera l'interface entre l'Avalon et notre programme vhd.

```

34 --registres d'écriture
35
36 process_write : process (clk, reset_n)
37 begin
38 if reset_n = '0' then
39 reset <= '0';
40 continu <= '0';
41 start_stop <= '0';
42 elsif clk'event and clk = '1' then
43 if chipselect = '1' and write_n = '0' then
44 if address = "00" then
45 reset <= writedata(0);
46 continu <= writedata(1);
47 start_stop <= writedata(2);
48
49
50
51 end if;
52 end if;
53 end if;
54 end process;
55
56 -- registres de lecture
57
58
59
60 process_Read : process(address, start_stop, continu, reset, data_compas, data_valid)
61 BEGIN
62 if address = "00" then
63 readdata <= X"00000000"&"0"&start_stop&continu&reset;
64 else
65 readdata <= X"00000"&"00"&data_valid&data_compas;
66
67 end if;
68 END PROCESS process_Read ;
69 -- Instance du composant "Avalon_compas"
70 C1 : Avalon_compas port map(reset, clk, start_stop, continu, pwm_in, data_compas, data_valid);
71
72 END arch_compas;

```

Figure 22: Code des registres de communications pour le compas

Le code assemble nos entrées et les place dans un registre (que nous avons appelé config) et les sorties dans le registre voisin (config +4), ceci nous permet de tester complètement notre code car nous pouvons contrôler les entrées (continu, start\_stop et reset) en réglant les valeurs de notre registre config.

(Reset est le LSB nous le réglons à 1 car il est actif bas, nous réglons continu à 1 et enfin start\_stop à 0) ce qui signifie que config contient 0x0003 en hexadécimal.

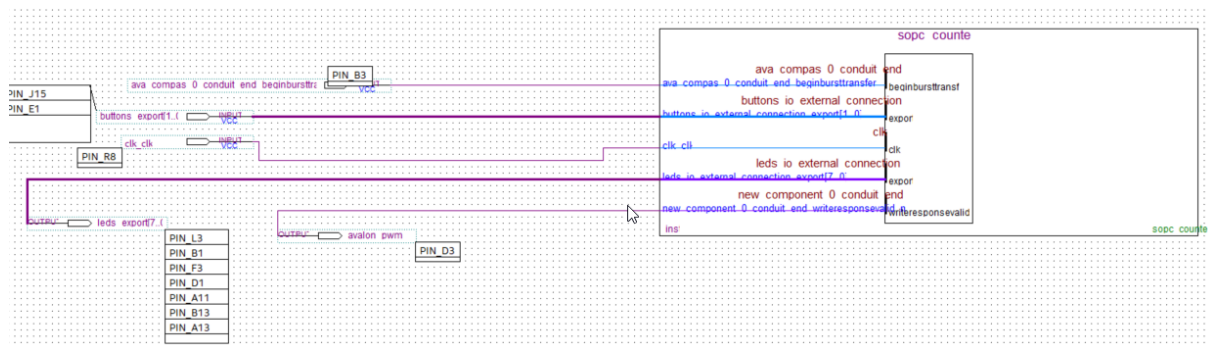


Figure 23: Bloc SOPC sur Quartus (avec compas)

Nous avons obtenu notre bloc finalisé à tester, ensuite nous avons relié notre horloge à l'entrée horloge de notre carte DE0 nano et les extrémités des conduits pour les broches pwm et compas à une broche de sortie et d'entrée respectivement, nous les connectons extérieurement de sorte que le pwm généré agit comme une entrée à notre module compas et que nous pouvons régler n'importe quel signal pwm que nous avons besoin d'utiliser pour tester le compas sans avoir besoin d'un équipement externe.

**Erreur: Port "ava\_compas\_0\_conduit\_end\_beginbursttransfer" does not exist in macrofunction "u0":**

Ce message d'erreur indique un problème avec le mappage des ports, par rapport au port "ava\_compas\_0\_conduit\_end\_beginbursttransfer" (celui crée par le conduit end du module compas). Nous avons donc défini une entité nommée "top" avec trois ports d'entrée : "clk\_50", "pwm" et "reset"

: "Nousinstancions notre composant "sopc\_counter" et mappions ses ports d'entrée aux signaux de l'entité "top".

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity top is
6  port (
7
8      clk_50 : in std_logic;
9      pwm : in std_logic;
10     reset : in std_logic
11 );
12
13 end entity top;
14
15 ARCHITECTURE arch of top IS
16
17     component sopc_counter is
18     port (
19         ava_compas_0_conduit_end_beginbursttransfer : in std_logic := 'X';
20         clk_clk : in std_logic := 'X';
21         reset_n : in std_logic := 'X' |
22     );
23 end component sopc_counter;
24
25
26
27
28
29 begin
30
31     u0 : component sopc_counter
32     port map (
33         ava_compas_0_conduit_end_beginbursttransfer => pwm,
34         clk_clk => clk_50
35         reset_n => reset
36     );
37
38
39 -- cette erreur (12002): Port "ava_compas_0_conduit_end_beginbursttransfer" does not exist in macrofunction "u0"
40
41
42 end arch;
43

```

Figure 24: code de résolution du problème de mapping

## 2. Tests et simulation de la fonction compas en SOPC

Nous créons ainsi le code C pour tester notre avalon compas , nous définissons les registres en fonction de leurs noms dans system.h en respectant la structure (**figure 20**) et nous fixons une valeur pour la frequence et un duty pour notre pwm généré de 85 ms et 23% (20 ms en état haut) respectivement , cela signifie que le résultat attendu est  $20-1 \text{ (offset)} * 10 = 190$  degrés , nous mettons également en place des masques parce que le registre data\_compas contient 2 valeurs différentes , les 9 premiers bits (b0..b8) sont la valeur du degré en binaire et le 10e bit (b9) est data\_valid.

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
Compas	1 (4)	R/W	b9=valid, b8..b0= data_compas

Tableau 2: Structure des registres de l'interface Avalon

```

100
101 #define freq (unsigned int *) NEW_COMPONENT_0_BASE
102 #define duty (unsigned int *) (NEW_COMPONENT_0_BASE + 4)
103 #define control (unsigned int *) (NEW_COMPONENT_0_BASE + 8)
104
105 #define boutons (volatile char *) BUTTONS_IO_BASE
106 #define leds (unsigned int*) LEDS_IO_BASE
107 unsigned int a,e,f;
108 #define config (volatile int *) AVA_COMPAS_0_BASE
109 #define data_compas (volatile int *) (AVA_COMPAS_0_BASE+4)
110
111 int main()
112 {
113     alt_putstr("Salut ext!\n"); // test si communication OK
114     *freq = 0x40D990; // diviser la clk de 50 M par 4250000 => periode de 85 ms
115     *duty = 0xF4240; // RC = 23% => duty de 20 ms
116     *control = 0x0003;
117     *config = 0x0003;
118
119     while (1)
120     {
121
122         alt_putstr("Salut int!\n");
123
124         e=*data_compas & 0x0040; //0000 0000 0100 0000
125         f=*data_compas & 0xFF80 ; //1111 1111 1000 0000
126         printf("data_valid = %d ",e);
127         printf("num_out= %d\n", (f-1)); usleep(100000);
128
129         a = *boutons & 3;
130         printf("boutons = %d\n", a);
131     }
132 }

```

Problems Tasks Console Nios II Console Properties

compas\_avalon Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtag\_uart\_0.jtag

```

boutons = 3
Salut int!
data_valid = 1 num_out= 190
boutons = 3
Salut int!
data_valid = 1 num_out= 190
boutons = 3
Salut int!
data_valid = 1 num_out= 190
boutons = 3

```

Figure 25: Code c pour le test et les résultats avec le NIOS

## Conclusion

Le projet que nous avons mené tout au long de ce semestre a constitué une expérience enrichissante dans le domaine du VHDL et de l'électronique numérique. Ce bureau d'études nous a offert l'opportunité de développer nos compétences en matière de description matérielle, permettant ainsi la représentation et la conception du comportement d'un système électronique numérique. Au cours de cette expérience, nous avons été confrontés à des défis liés à la limitation matérielle, nécessitant une compréhension approfondie de chaque composant numérique pour manipuler efficacement certains blocs fonctionnels.

Il est important de souligner que la programmation VHDL offre une approche différente, axée sur la conception matérielle, ce qui a élargi notre compréhension de l'architecture des systèmes électroniques. La nécessité de comprendre le matériel a été cruciale pour surmonter ces défis et optimiser la conception de certaines fonctions.

Par ailleurs, l'outil de simulation disponible dans les logiciels de programmation VHDL s'est avéré être un atout puissant. Il a facilité l'implémentation et la validation des systèmes électroniques en offrant une plateforme de conception assistée par ordinateur. Cette expérience a également souligné l'importance de la simulation dans le processus de développement, permettant de tester et de valider les fonctionnalités du système avant une éventuelle implémentation matérielle.

En conclusion, ce projet a été une occasion unique de mettre en pratique nos connaissances théoriques et d'acquérir une expérience concrète dans le domaine de l'électronique numérique et du VHDL. Les défis rencontrés ont renforcé notre compréhension des aspects pratiques de la conception matérielle, tout en soulignant l'importance des outils de simulation dans le processus de développement.