



Département EEA - Faculté Sciences et Ingénierie

Master 2 SME

Synthèse et mise en œuvre des systèmes

Année 2023-2024

**Compte rendu :
Pilote de barre franche pour voiliers**

Rédigé par : HAMDAN feras
et SALAH Hichem

Encadré par : SAL Y ROSAS Damian
et Perisse Thierry

19 décembre 2023

Table des matières

1	Introduction	3
2	Compas	4
2.1	Compas sans NiosII	4
2.2	Compas avec NiosII	7
2.3	Test et Validation	9
3	Gestion du convertisseur	12
3.1	Machine à état	12
3.2	Gestion du convertisseur avec NiosII	13
3.3	Test et validation	14
4	Compas avec gestion du convertisseur	15
4.1	Test et validation	17
5	Travail restant	18
5.1	Gestion du vérin	18
5.2	Bus Avalon	18
5.3	Mise en relation de tous les blocs	18
6	Conclusion	19
7	Annexe	20
7.1	Compas	20
7.2	Compas avec NiosII	29
7.3	Gestion du convertisseur	36

1 Introduction

Dans le cadre de notre unité d'enseignement intitulée "Synthèse et Mise en Œuvre de Systèmes", nous avons entrepris la réalisation, au cours de ce projet d'étude, d'un pilote à barre franche pour un voilier. L'objectif de ce projet est d'automatiser la navigation du voilier en utilisant un système embarqué programmable appelé SOPC (System On Programmable Chip), qui est décrit à l'aide du langage de description matérielle VHDL (Very High Speed Hardware Description Language).

Ce système comprend diverses fonctions, parmi lesquelles une fonction qui fait usage d'un compas pour obtenir les mesures d'angles relatives au plan horizontal du voilier, permettant ainsi de définir sa direction, et une fonction qui contrôle un convertisseur analogique numérique et qui communique avec grâce au protocole SPI. Ce rapport se concentre sur la présentation de la mise en œuvre de ces fonctions spécifiques. Nous décrirons l'architecture de notre solution, les composants individuels qui la composent, les éventuelles machines à états qui décrivent leur comportements, ainsi que les tests que nous avons effectués pour valider son bon fonctionnement.

2 Compas

2.1 Compas sans NiosII

Le module compas que nous utilisons est une boussole équipée d'une compensation d'inclinaison qui nous permet de recueillir des données sur l'angle d'inclinaison. Pour mieux expliquer le fonctionnement de notre module, imaginons que le compas soit initialement orienté vers le nord, qui sert de point de référence dans notre système. À mesure que l'angle d'inclinaison augmente, nous nous déplaçons successivement vers les points cardinaux suivants : Nord \rightarrow Est \rightarrow Sud \rightarrow Ouest.

Le module Compas fonctionne en prenant en entrée un signal PWM (Modulation en Largeur d'Impulsion) : lorsque la boussole tourne, elle génère une impulsion élevée qui est proportionnelle à l'angle actuel. La largeur de cette impulsion varie de 1 ms (0°) à 36,99 ms ($359,9^\circ$), avec un incrément de 100 μ s par degré et un offset de 1 ms. Entre chaque impulsion, le signal reste bas pendant 65 ms, ce qui signifie que le cycle de temps total est de 65 ms plus la largeur de l'impulsion.

Le fonctionnement de ce système repose sur deux modes opérationnels distincts :

Le mode continu : Ce mode est activé lorsque l'entrée continu est réglée à 1. En mode continu, le circuit de gestion de la boussole effectue une acquisition de données toutes les secondes de manière régulière.

Le mode mono-coup : ce mode est activé lorsque l'entrée continu est réglée à 0. En mode mono-coup, une valeur de 1 sur l'entrée Start Stop est nécessaire pour initier une acquisition de données.

En ce qui concerne les sorties du système, il y a deux sorties distinctes : data valid et degré, cette dernière étant codée sur 9 bits. La sortie data valid permet de vérifier si la valeur en sortie du circuit est valide ou non. Dans le cas du mode continu, puisque l'acquisition se répète chaque seconde, il n'y a pas de problème de validité des données en sortie. Cependant, en mode mono-coup, il est essentiel de garantir la validité des données en sortie. C'est pourquoi la sortie data valid est positionnée à 1 une fois que le calcul d'acquisition est terminé, et elle revient à 0 une fois que l'entrée Start Stop est remise à 0, indiquant ainsi la fin de l'acquisition de données.

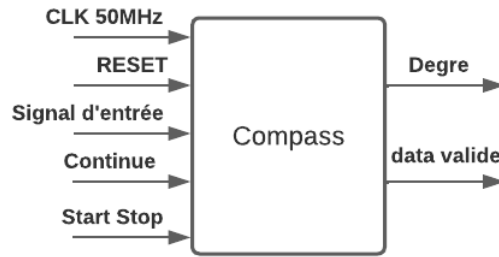


FIGURE 1 – Entrée/Sortie du compas

On peut observer sur la figure 1 les entrées et les sorties du compas. En entrée on retrouve l'horloge de 50 MHz, reset, le signal d'entrée contenant l'information sur le cap, continue et Start Stop. En sortie on retrouve Degre qui correspond au cap en degré et data valid qui indique si la sortie peut être prise en compte.

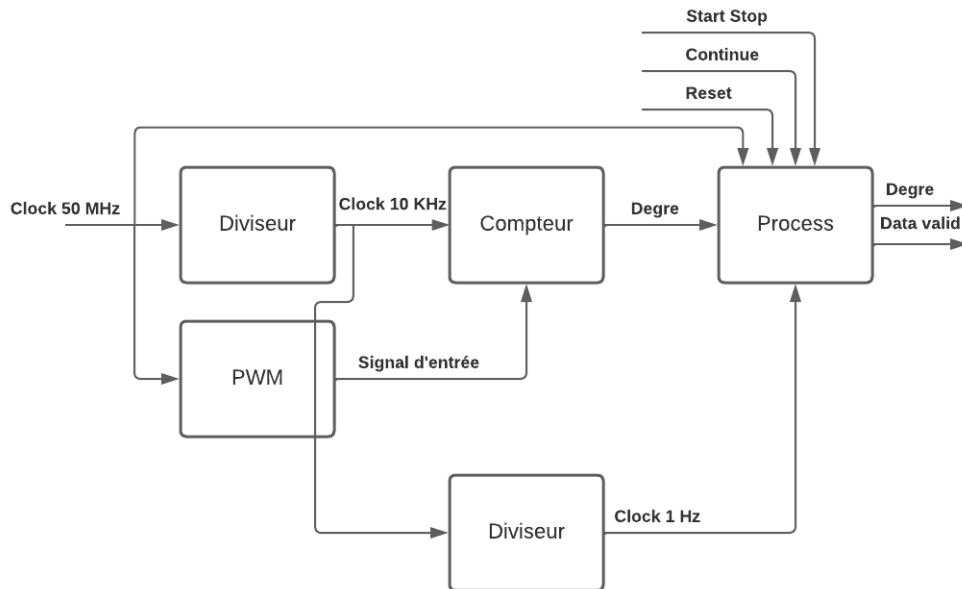


FIGURE 2 – Architecture du compas

On peut observer sur la figure 2 l'architecture du compas sous la forme d'un schéma bloc. Le diviseur transforme l'horloge de 50 MHz en une horloge de 10 kHz. Le compteur s'incrémente de 1 chaque front montant et donc s'incrémente chaque 100 μ s, si le signal d'entrée est à l'état haut. Autrement dit le compteur compte le nombre de μ s que le signal d'entrée est à l'état haut. Sur cette figure le signal d'entrée est généré par le composant PWM qui ne fait pas parti du compas mais qui est implémenter pour la réalisation de test à l'étape de validation. Le compteur donne donc directement le résultat en degré en sortie. Le Process gère donc, en fonction de l'état des entrées continu et Start Stop, les sortie du compas, Degre et Data valide.

Le diviseur, (annexe 5.1.2) compte jusqu'à 2500 avant de changer l'état de la sortie, la sortie est donc une horloge créer en divisant par 5000 l'horloge d'entrée, la fréquence passe donc de 50 MHz à 10 kHz. Le diviseur 2 compte jusqu'à 5000 avant de changer l'état de la sortie, la sortie est donc une horloge créer en divisant par 10000 l'horloge d'entrée, la fréquence

passé donc de 10 kHz à 1 Hz.

Le compteur, (annexe 5.1.4) contient un process qui est sensible à l'horloge de 10 kHz, à chaque front montant de l'horloge, chaque 100 μ s, le compteur s'incrémente de 1 si le signal d'entrée est à l'état haut, or d'après la datasheet du compas, chaque 100 μ s correspond à un degré, le compteur compte donc directement les degrés que l'on récupère en sortie.

Le composant `continu_component` gère la temporisation de la sortie en mode continu. Dans ce mode l'acquisition du cap se fait chaque seconde. Le process est sensible à l'horloge produite par le composant `div_compa2`, de période 1 seconde. Autrement dit si on est dans le mode continu la sortie est rafraîchie chaque seconde.

Le composant `ss_component` gère la sortie en mode monocoup. Le process est sensible sur l'entrée Start Stop, la sortie est donc rafraîchie sur chaque front montant de Start Stop.

2.2 Compas avec NiosII

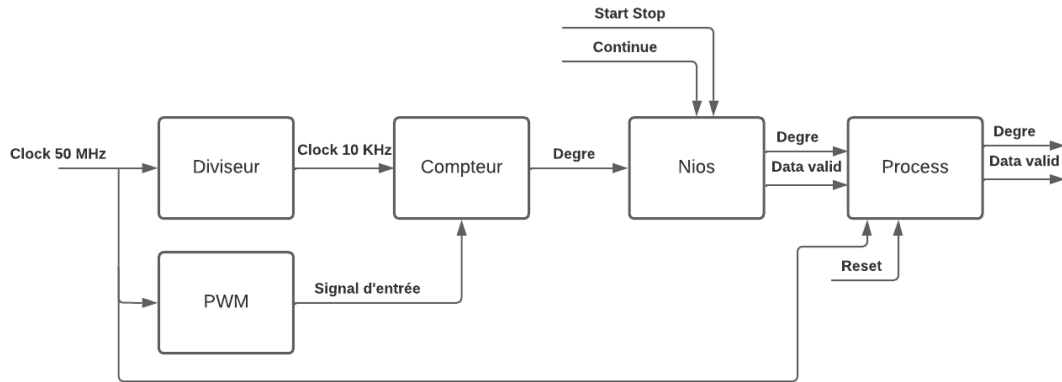


FIGURE 3 – Architecture du compas avec NiosII

On peut observer sur la figure 3 l'architecture du compas avec l'ajout du nios sous la forme d'un schéma bloc. Le nios est placé entre le compteur et le process. On peut voir dans le programme `nios_C.c` (annexe 5.2.3) que le nios gère le déclenchement de l'acquisition du cap dans les modes continu et monocoup. En sortie nios envoie au process la valeur du cap en degré en fonction du mode, en mode continu l'acquisition se produit chaque seconde grâce à la fonction `usleep()`. Les entrées continu et Start Stop sont donc directement reliées au nios et non pas au process.

2.3 Test et Validation

Pour la validation du compas on test en mettant en entrée une PWM avec un temps à l'état haut pré détermine, on observe alors la sortie, sur les LEDs pour l'architecture sans nios (figure 2) et sur la console pour l'architecture avec nios (figure 3), on a également observer data valide sur une LED ou sur un pin GPIO.

La figure 7 montre la sortie du compas pour le signal en entrée de la figure 6. On observe que pour un temps à l'état haut de 16.3 ms soit 163 μ s on obtient un cap de 162°. On effectue un deuxième test pour montrer que le compas donne un cap correct pour différentes entrées.

La figure 9 montre la sortie du compas pour le signal en entrée de la figure 8. On observe que pour un temps à l'état haut de 34.9 ms soit 349 μ s on obtient un cap de 349°.

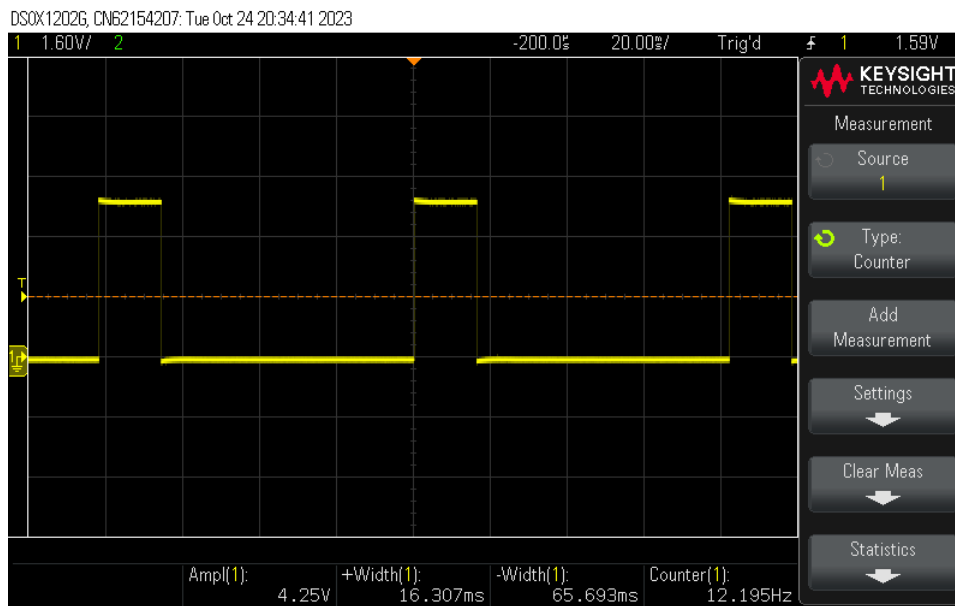


FIGURE 6 – Signal Pwm en entrée pour le test 1

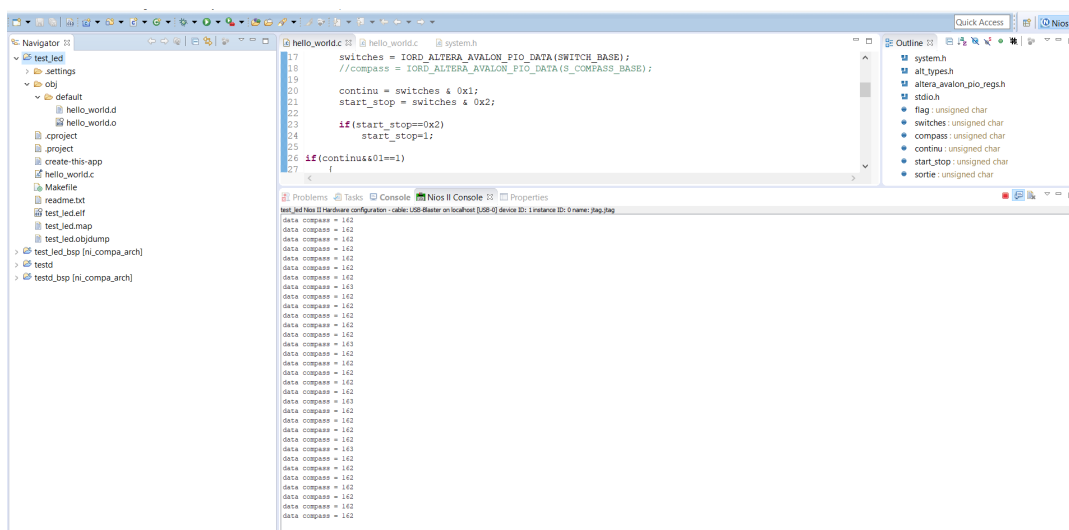
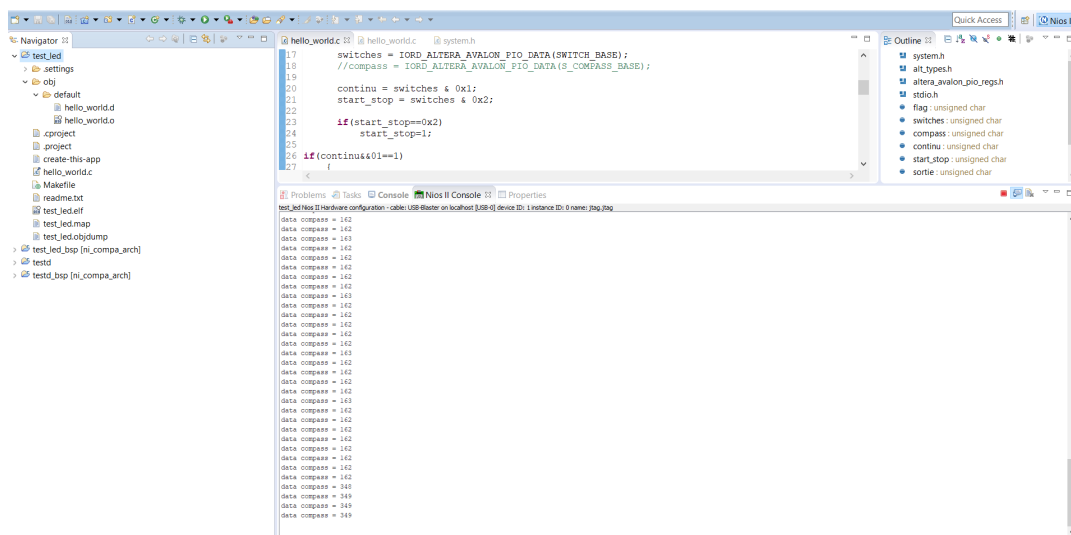
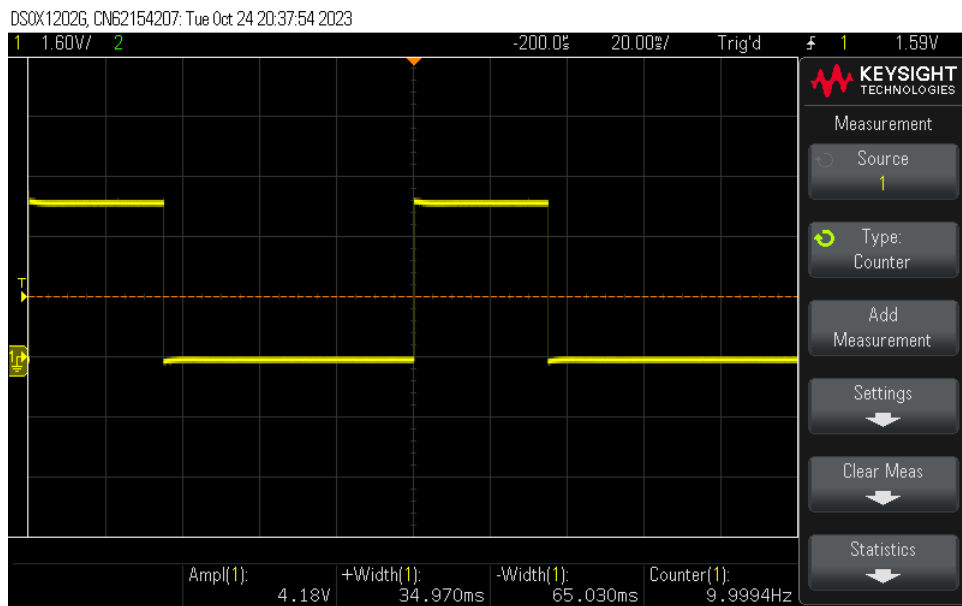


FIGURE 7 – Résultat du test 1 sur console



Lors de l'étape de validation, on test également le comportement du compas dans les modes continu et monocoup. Pour le mode continu on s'attend à avoir le cap s'afficher sur la console toutes les secondes et d'avoir data valid toujours à 1. Pour le mode monocoup on cherche à ce que le cap s'affiche sur la console qu'un front montant de Start Stop, autrement dit lorsqu'on mets Start Stop à 1 on s'attend à avoir une seule acquisition du cap, on s'attend également à data valid à 1 que lorsque Start Stop est à 1.

3 Gestion du convertisseur

3.1 Machine à état

A partir de la datasheet du convertisseur on peut en déduire la machine à état de la figure 10 :

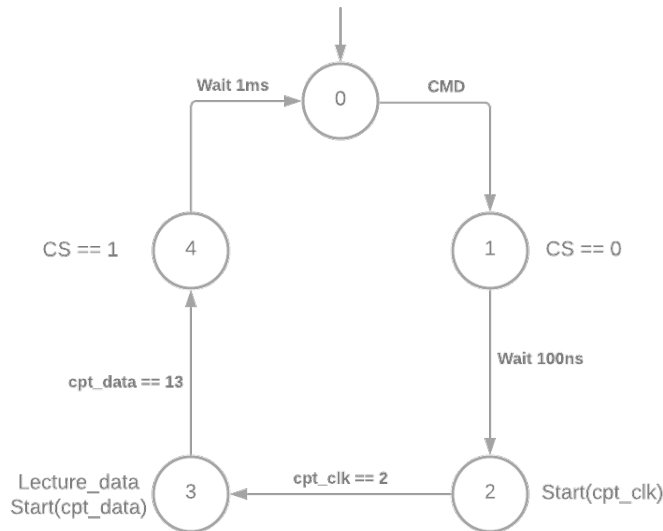


FIGURE 10 – Machine d'état pour la commande du convertisseur

Le rôle de la machine d'état est de contrôler Cs, le signal de commande du convertisseur et de lire la trame envoyée par le convertisseur au bon moment. Initialement on se trouve dans l'état 0, lorsqu'on le signal de commande cmd passe à 1, on passe à l'état 1 où on met Cs à 0 pour commencer la conversion, d'après la datasheet il faut attendre 100ns, on passe alors à l'état 2, où on déclenche un compteur de front d'horloge cpt_clk. Après deux front d'horloge on passe à l'état 3 où on commence la lecture de la trame et où le compteur cpt_data est déclenché, il compte les front montant d'horloge correspondant à un bit de donnée reçue. Lorsque le compteur arrive à 13, cela signifie que toute la donnée a été reçue, on passe alors à l'état 4 où on met Cs à 1 pour terminer la conversion, on attend 100ms pour repasser à l'état initial pour respecter le cahier des charges.

Comme on peut voir sur le code correspondant (annexe 6.3.1), on retrouve 3 process liés à la machine à état, un pour mettre en place le reset et synchroniser le passage des états avec l'horloge, un pour définir les conditions de passage entre les états et un pour définir les sorties. On retrouve également dans le programme un process qui crée une horloge de 1MHz, un process qui compte 100ns, un process qui compte 1µs, un process qui compte les fronts montants d'horloge (1MHz), un process qui compte le nombre de bits reçus (envoyé par le convertisseur) en comptant les fronts montants d'horloge (1MHz) et un process qui lit les données reçues, on utilise ici un registre à décalage.

3.2 Gestion du convertisseur avec NiosII

On peut observer sur la figure 11 l'architecture de la gestion du convertisseur avec l'ajout du nios sous la forme d'un schéma bloc. Le Nios prend le rôle de lancer la conversion et d'afficher les résultat. Il fait en sorte que la conversion ne démarre que toutes les 100ms.

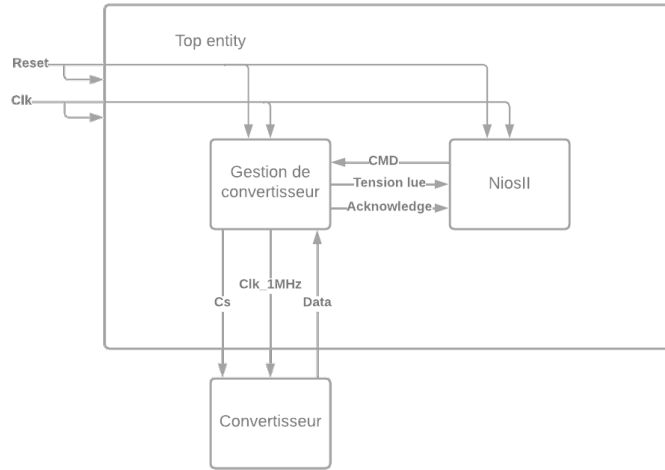


FIGURE 11 – Machine d'état pour la commande du convertisseur

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ	Tags	Opcode Name
<input checked="" type="checkbox"/>		clk_0	Clock Source							
		clk_in	Clock Input	clk	exported					
		clk_in_reset	Reset Input	reset						
		clk	Clock Output	Double-click to export	clk_0					
		clk_reset	Reset Output	Double-click to export						
<input checked="" type="checkbox"/>		NiosII	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]			IRQ 0	IRQ 31	
		debug_reset_requ...	Reset Output	Double-click to export	[clk]					
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	# 0x0002_0800	0x0002_0fff			
		custom_instructio...	Custom Instruction Master	Double-click to export	[clk]					
<input checked="" type="checkbox"/>		memory	On-Chip Memory (RAM or ROM)...							
		clk1	Clock Input	Double-click to export	clk_0					
		sl	Avalon Memory Mapped Slave	Double-click to export	[clk1]	# 0x0001_0000	0x0001_9c3f			
		reset1	Reset Input	Double-click to export	[clk1]					
<input checked="" type="checkbox"/>		JTAG	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]	# 0x0002_1038	0x0002_103f			
		irq	Interrupt Sender	Double-click to export	[clk]					
<input checked="" type="checkbox"/>		systeme_ID	System ID Peripheral Intel FPGA...							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
<input checked="" type="checkbox"/>		control_slave	Avalon Memory Mapped Slave	Double-click to export	[clk]	# 0x0002_1030	0x0002_1037			
		clk	PID (Parallel I/O) Intel FPGA IP	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		sl	Avalon Memory Mapped Slave	Double-click to export	[clk]	# 0x0002_1020	0x0002_102f			
<input checked="" type="checkbox"/>		external_connection	Conduit							
		clk	PID (Parallel I/O) Intel FPGA IP	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		sl	Avalon Memory Mapped Slave	Double-click to export	[clk]	# 0x0002_1010	0x0002_101f			
		external_connection	Conduit	Double-click to export	command_external...					
<input checked="" type="checkbox"/>		acknowledge	PID (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		sl	Avalon Memory Mapped Slave	Double-click to export	[clk]					
		external_connection	Conduit	Double-click to export	acknowledge_extern...					
<input checked="" type="checkbox"/>		data	PID (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	[clk]					
		sl	Avalon Memory Mapped Slave	Double-click to export	[clk]	# 0x0002_1000	0x0002_100f			
		external_connection	Conduit	Double-click to export	data_external_conn...					

FIGURE 12 – Architecture du NiosII

On peut voir sur la figure 12 l'architecture du nios.

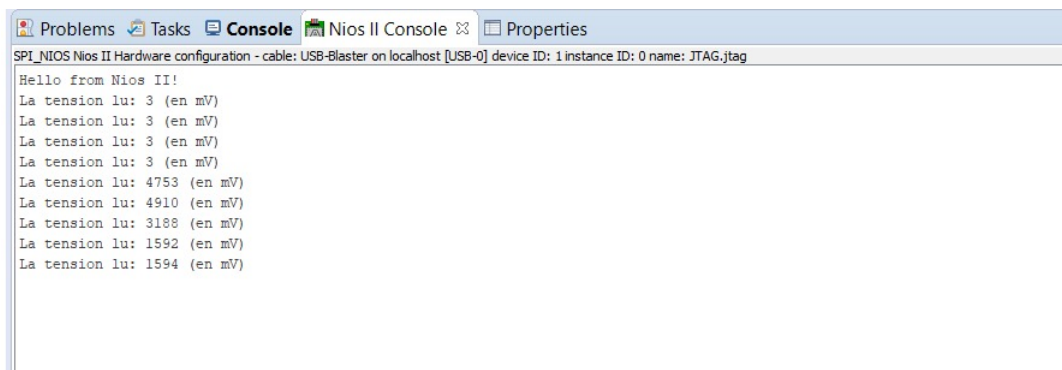
3.3 Test et validation

Pour la validation de notre programme, on test d'abord sur le banc de test où on visualise la la sortie sur les LEDs de la carte DE0 nano. La sortie est sur 12 bits et la carte en contient 8 on affiche donc que les 8 bits de poix forts.

Une fois que nous observons le comportement attendu, on test la version du programme qui intègre le NiosII, on peut alors afficher directement la sortie sur la console. On peut voir sur la figure 13.

Lorsqu'on tourne le potentiomètre correspondant à la position du vérin, on mesure la tension au borne du convertisseur que nous comparons à la valeur donner par les LEDs (après avoir converti la valeur binaire en décimale) ou par la console. Dans le cas où le Nios n'est pas intégré le signal CMD est donné par un bouton poussoir, dans le cas contraire, il est donné par le Nios.

On considère le programme validé lorsque les valeurs comparées sont cohérentes.



```
SPI_NIOS Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: JTAG.jtag
Hello from Nios II!
La tension lu: 3 (en mV)
La tension lu: 3 (en mV)
La tension lu: 3 (en mV)
La tension lu: 3 (en mV)
La tension lu: 4753 (en mV)
La tension lu: 4910 (en mV)
La tension lu: 3188 (en mV)
La tension lu: 1592 (en mV)
La tension lu: 1594 (en mV)
```

FIGURE 13 – Résultat du test sur console

4 Compas avec gestion du convertisseur

Sur la figure 14 est représenté en schéma bloc l'assemblage de la fonction gestion du convertisseur et de la fonction compas. Le Nios commande ces fonctions, compas en fonction des entrées continu et start_stop et gestion de convertisseur en fonction du temps. Les données entant affichées sur la console du Nios, la seule sortie du système est data_valide. La commande de la gestion du compas et la commande du compas en mode continu sont géré par des timers d'interruptions. Idéalement il faudrait deux interruptions avec deux timer différents, un de 100mset un de 1s. Il n'est néanmoins pas possible d'implémenter cette solution sans la licence de QuartusII, on utilise donc une seule interruption de 100mset on compte 10 utilisations de la fonction d'interruption pour lancer une acquisition du compas en mode continu et on lance une commande de conversion à chaque utilisation de la fonction.

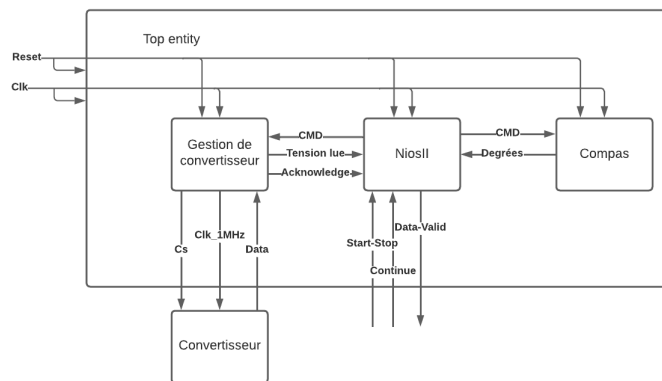


FIGURE 14 – Machine d'état pour la commande du convertisseur

Use	Connections	Name	Description	Expert	Clock	Base	End	IRQ	Tags	Opcode Name
<input checked="" type="checkbox"/>		clk_0	Clock Source	clk	external					
		clk_in	Clock Input	Double-click to export	clk_0					
		clk_in_reset	Reset Input	Double-click to export	clk_0					
		clk	Clock Output	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		nios2	Nios II Processor							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	clk_0					
		data_master	Avalon Memory Mapped Master	Double-click to export	clk_0					
		instruction_master	Avalon Memory Mapped Master	Double-click to export	clk_0					
		irq	Interrupt Receiver	Double-click to export	clk_0					
		debug_reset_req	Reset Output	Double-click to export	clk_0					
		debug_mem_slave	Avalon Memory Mapped Slave	Double-click to export	clk_0					
		custom_instruction...	Custom Instruction Master	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		memory	On-Chip Memory (RAM or ROM)...							
		clk1	Clock Input	Double-click to export	clk_0					
		s1	Avalon Memory Mapped Slave	Double-click to export	clk_0					
		reset1	Reset Input	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		sysid_qsys_0	System ID Peripheral Intel FPGA...							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	clk_0					
		control_slave	Avalon Memory Mapped Slave	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART Intel FPGA IP							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	clk_0					
		avalon_jtag_slave	Avalon Memory Mapped Slave	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		irq	Interrupt Sender							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	clk_0					
		external_connection	Avalon Memory Mapped Slave	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		position_in	PID (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	clk_0					
		s1	Avalon Memory Mapped Slave	Double-click to export	clk_0					
		external_connection	Conduit	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		ss	PID (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	clk_0					
		reset	Reset Input	Double-click to export	clk_0					
		s1	Avalon Memory Mapped Slave	Double-click to export	clk_0					
		external_connection	Conduit	Double-click to export	clk_0					
<input checked="" type="checkbox"/>		continue	PID (Parallel I/O) Intel FPGA IP							
		clk	Clock Input	Double-click to export	clk_0					

FIGURE 15 – Architecture du NiosII

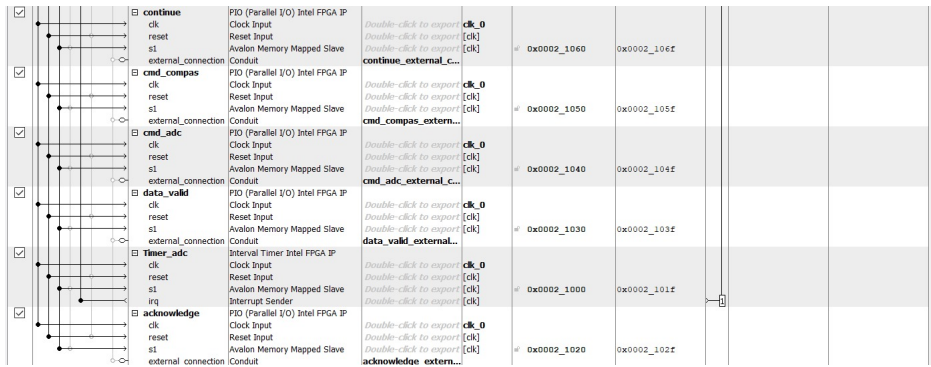


FIGURE 16 – Architecture du NiosII

On peut voir sur les figures 15 et 16 l'architecture du nios.

4.1 Test et validation

Pour cette phase de test, on ne test que en visualisant les données sur la console du Nios. On utilise le code (annexe 6.3.7) à la place du code (annexe 6.3.8) pour pouvoir afficher les données du compas et du convertisseur à la suite.

Comme on peut le voir sur la figure 17, notre système à le comportement souhaité.

```
mmd_adc_comp Nios II Hardware configuration - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtag_uart_0.jtag
La tension lu: 3501 (en mV)
data compas = 22
La tension lu: 3501 (en mV)
data compas = 22
La tension lu: 3555 (en mV)
data compas = 22
La tension lu: 4032 (en mV)
data compas = 21
La tension lu: 3516 (en mV)
data compas = 21
La tension lu: 4023 (en mV)
data compas = 18
La tension lu: 3749 (en mV)
data compas = 20
La tension lu: 3516 (en mV)
data compas = 22
La tension lu: 4037 (en mV)
data compas = 19
La tension lu: 3516 (en mV)
data compas = 21
La tension lu: 4037 (en mV)
data compas = 21
La tension lu: 4037 (en mV)
data compas = 21
La tension lu: 3652 (en mV)
```

FIGURE 17 – Machine d'état pour la commande du convertisseur

5 Travail restant

5.1 Gestion du vérin

Le fonction gestion du convertisseur fait parti de la fonction gestion du vérin. Cette fonction comprend la gestion du convertisseur, la gestion de la pwm contrôlant le moteur et le contrôle de buté.

Sur la figure 18 on peut voir l'architecture complète du travail à faire, c'est à dire le compas avec la gestion du vérin. On y retrouve en plus de l'architecture de la figure 14 la fonction pwm qui génère un signal pwm, avec un rapport cyclique défini par le Nios, qui contrôle le moteur qui modifiera la tension au bornes du convertisseur en fonction de sa position que le convertisseur enverra à la fonction gestion du convertisseur qui l'enverra au Nios qui l'utilisera pour ajuster la valeur du rapport cyclique, créant ainsi une boucle d'asservissement.

La fonction gestion de buté à pour but de gérer positions limites, max et min, du vérin en mettant à 0 la pwm si ces positions ont atteintes. Cette fonction peut être implémenté directement en c sur le Nios ou être inclus dans le composant pwm. L'implémentation sur le Nios sera à privilégier.

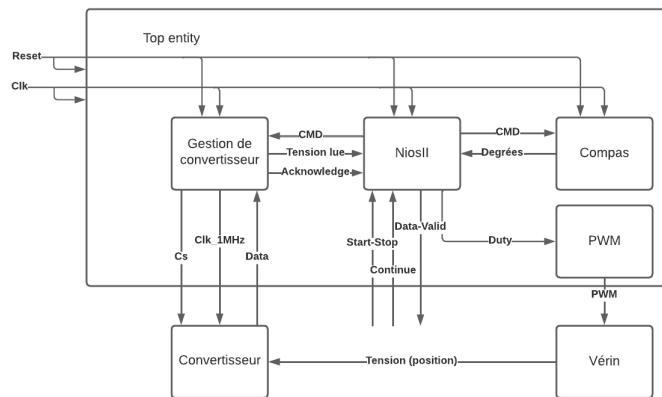


FIGURE 18 – Machine d'état pour la commande du convertisseur

5.2 Bus Avalon

Le bus Avalon permet la communication entre les blocs VHDL et le bloc Nios. Pour gérer cette communication nous avons créé des périphériques PIO lors de l'étape plateforme designer. Il nous reste donc à modifier cette partie du projet pour mettre explicitement en place le bus Avalon.

5.3 Mise en relation de tous les blocs

Même si nous avons mis en commun les blocs de gestion de convertisseur et de compas, le but du BE était d'assembler la totalité des blocs à implémenter pour réaliser le pilote de barre franche.

6 Conclusion

Ce projet, même incomplet, nous a permis d'obtenir les bases du développement sur FPGA et d'avoir une idée de ce que cette technologie peut permettre. Cette expérience en matière nous a également permis d'acquérir des compétences dans un nouveau langage de programmation, le VHDL.

Lors de ce BE nous avons été amené à réaliser plusieurs composants complexes, notamment d'une boussole et d'un pilote SPI pour communiquer avec un convertisseur analogique numérique.

7 Annexe

7.1 Compas

7.1.1 Compass

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity compass is port(
clk, razn, in_pwm, continu, start_stop :in std_logic;
data_valid: out std_logic:='0';
  out_ls : out std_logic;
data_compas : out std_logic_vector(8 downto 0);
out_pwm : out std_logic

);
end entity;

architecture C of compass is

signal sig1,sig5,pwm_sig,sig_data_valid : std_logic;
signal sig2,sig3,sig6,sig7,sig8,sig_continu,sig_ss : std_logic_vector(8 downto 0);
signal sig4 : std_logic_vector(1 downto 0);

component div_compa is
port (
  clk_div : in std_logic;
  S_div: out std_logic);
end component;

component div_compa2 is
port (
  clk_div : in std_logic;
  S_div: out std_logic);
end component;

component deg_compa is
port(
  deg_in,in_pwm : in std_logic;
  deg_s, deg_s2: out std_logic_vector(8 downto 0));
end component;

component pwm_q18 is
port(
  clk_pwm, reset_pwm: in std_logic;
  --freq_pwm, duty_pwm: in std_logic_vector(7 downto 0);
  s_pwm : out std_logic

);
```

```

end component;

component continu_component is
port(
    clk_continu : in std_logic;
    sig_entree,sig_entree2: in std_logic_vector(8 downto 0);
    --sorite_valid: out std_logic;
    sortie: out std_logic_vector(8 downto 0));
end component;

component ss_component is
port(
    start,raz : in std_logic;
    sig_entree,sig_entree2: in std_logic_vector(8 downto 0);
    --sorite_valid: out std_logic;
    sortie: out std_logic_vector(8 downto 0));
end component;

begin
dv_compas : div_compa
port map (
    clk_div=>clk ,
    S_div=>sig1 );
dv_compas_2 : div_compa2
port map (
    clk_div=>sig1 ,
    S_div=>sig5 );
dv_pwm : pwm_q18
port map(
    clk_pwm=>clk,
    reset_pwm=>'0',
    s_pwm=>pwm_sig);
dv_deg : deg_compa
port map(
    deg_in=>sig1,
    in_pwm=>pwm_sig,
    deg_s=>sig6,
    deg_s2=>sig8);
dv_continu: continu_component
port map(
    clk_continu=>sig5,
    sig_entree=>sig6 ,
    sig_entree2=>sig8,
    sortie=>sig_continu);
dv_ss: ss_component
port map(
    start=>start_stop,
    sig_entree=>sig6, raz=>razn ,
    sig_entree2=>sig8,
    sortie=>sig_ss);

```

```

process(clk)
begin

if clk 'event and clk = '1' then
if razn = '1' then
if continu='0' and start_stop='1' then
sig7<=sig_ss;
sig_data_valid<='1';
elsif continu='0' and start_stop='0' then
sig_data_valid<='0';
elsif continu='1' then
sig7<=sig_continu;
sig_data_valid<='1';
end if;
elsif razn = '0' then
sig7<="00000000";
end if;
end if;

end process;

out_1s<=sig5;
data_compas<=sig7;
out_pwm<=pwm_sig;
data_valid<=sig_data_valid;

end architecture;

```

7.1.2 div_compas

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity div_compa is port(
    clk_div : in std_logic;
    S_div: out std_logic);
end entity;

architecture dv of div_compa is
    signal sig : std_logic_vector(11 downto 0);
    signal S2: std_logic;
begin
    process(clk_div)
    begin

        IF clk_div 'event and clk_div = '1' THEN
            sig <= sig + 1 ;

            IF sig = "100111000011" THEN
                S2 <= (not S2) ;
                sig <= "000000000000";
            end if;
        end if;
        S_div <= S2;
    end process;

end architecture dv ;
```

7.1.3 pwm_q18

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity pwm_q18 is port(
    clk_pwm, reset_pwm: in std_logic;
    --freq_pwm, duty_pwm: in std_logic_vector(7 downto 0);
    s_pwm : out std_logic
);
end entity;

architecture pm of pwm_q18 is

    signal sig1,clk_pwm2 : std_logic;
    signal sig2 : std_logic_vector (7 downto 0);
    signal sig3 : std_logic;
    signal sig4 : std_logic;
    component div8 is
    port ( clk_div8 : in std_logic;
          e_div8: in std_logic_vector(7 downto 0);
          S_div8: out std_logic);
    end component;

    component cmpt8 is
    port ( E_cmpt8 : in std_logic;
          S_cmpt8: out std_logic_vector (7 downto 0));
    end component;

    component compara is
    port ( e_compara,duty_compara: in std_logic_vector(7 downto 0);
          S_compara: out std_logic );
    end component;

begin

    dv_pwm : div8
    port map ( clk_div8=>clk_pwm , e_div8=>"11111111" , S_div8=>clk_pwm2 );

    dv_pwm2 : div8
    port map ( clk_div8=>clk_pwm2 , e_div8=>"00001101" , S_div8=>sig1 );

    cmp_pwm : cmpt8
    port map ( E_cmpt8=>sig1 , S_cmpt8=>sig2 );

    compa_pwm : compara
    port map ( e_compara=>sig2 , duty_compara=>"00011111" , S_compara=>sig3 );
    process(clk_pwm ,reset_pwm)
    begin
        IF clk_pwm 'event and clk_pwm = '1' THEN
```



```
if reset_pwm = '1' then
sig4 <= '0';
else sig4 <= sig3;
end if;
end if;
end process;
s_pwm<=sig4;
end architecture;
```

7.1.4 deg_compa

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity deg_compa is port(
    deg_in,in_pwm : in std_logic;
    deg_s, deg_s2: out std_logic_vector(8 downto 0));
end entity;

architecture dv of deg_compa is
    signal sig2,sig3 : std_logic_vector(8 downto 0);
begin
    process(deg_in)
    begin
        if deg_in 'event and deg_in = '1' then

            if in_pwm='1' then
                sig2 <=sig2+1;

            else
                sig2<="000000000";

            end if;

            if sig2>0 then
                sig3<=sig2-10;

            end if;

        end if;
    end process;
    deg_s<=sig3;
    deg_s2<=sig2;

end architecture dv ;
```

7.1.5 continu_component

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity continu_component is port(
    clk_continu : in std_logic;
    sig_entree,sig_entree2: in std_logic_vector(8 downto 0);
    --sorite_valid: out std_logic;
    sortie: out std_logic_vector(8 downto 0));
end entity;

architecture cc of continu_component is
    signal sig: std_logic_vector(8 downto 0);
    --signal data: std_logic;
begin
    process(clk_continu)
    begin
        if clk_continu 'event and clk_continu = '1' then
            if sig_entree2="00000000" then
                sig<=sig_entree;
                --data<='1';
            end if;
        end if;
    end process;
    sortie<=sig;
    --sorite_valid<=data;

end architecture cc ;
```

7.1.6 ss_component

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity ss_component is port(
    start,raz : in std_logic;
    sig_entree,sig_entree2: in std_logic_vector(8 downto 0);
    --sorite_valid: out std_logic;
    sortie: out std_logic_vector(8 downto 0));
end entity;

architecture ss of ss_component is
    signal sig: std_logic_vector(8 downto 0);
    --signal data: std_logic;
begin
    process(start,raz)
    begin
        if raz = '1' then
            if start 'event and start = '1' then

                if sig_entree2="000000000" then
                    sig<=sig_entree;
                    --data<='1';
                end if;
            end if;
        elsif raz = '0' then
            sig<="000000000";

        end if;
    end process;
    sortie<=sig;
    --sorite_valid<=data;

end architecture ss ;
```

7.2 Compas avec NiosII

7.2.1 nios_compas_vhdl

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity ni_compa is
    port(
        clk, razn, in_pwm :in std_logic;
        switch: in std_logic_vector(3 downto 0);
        provisoire : in std_logic;
        data_valid: out std_logic:='0';
        out_1s : out std_logic;
        data_compass : out std_logic_vector(8 downto 0);
        out_pwm : out std_logic
    );
end entity;

architecture c_n of ni_compa is
    signal sig_ss, sig_dv,sig_out, sig_pwm: std_logic :='0';
    signal sig_cd,sig_deg: std_logic_vector(8 downto 0);
    signal sig_led: std_logic_vector(7 downto 0);

    component compass_2 is
        port(
            clk, razn, in_pwm, continu, start_stop :in std_logic;
            data_valid: out std_logic:='0';
            out_1s : out std_logic;
            data_compas : out std_logic_vector(8 downto 0);
            out_pwm : out std_logic
        );
    end component;

    component ni_compa_arch is
        port (
            clk_clk           : in  std_logic           := '0';
            data_valid_in_export : in  std_logic           := '0';
            data_valid_out_export : out std_logic;
            degre_export       : in  std_logic_vector(8 downto 0) := (others => '0');
            leds_degre_export  : out std_logic_vector(8 downto 0);
            pwm_out_export     : out std_logic_vector(7 downto 0);
            reset_reset_n     : in  std_logic           := '0';
            ss_export          : out std_logic;
            switch_export      : in  std_logic_vector(3 downto 0) := (others => '0')
        );
    end component;
begin
```

```

c2:compass_2
port map(
    clk=>clk,
    razn=>razn,
    in_pwm=>in_pwm,
    continu=>'0',
    start_stop=>sig_ss,
    data_valid=>sig_dv,
    out_1s=>out_1s,
    data_compas=>sig_deg,
    out_pwm=>out_pwm);

nc:ni_compa_arch
port map(
    clk_clk=>clk,
    data_valid_in_export=>sig_dv,
    data_valid_out_export=>data_valid ,
    pwm_out_export=>sig_led,
    reset_reset_n=>razn,
    degre_export=>sig_deg,
    ss_export=>sig_ss,
    switch_export=>switch,
    leds_degre_export=>data_compass);

end architecture;

```

7.2.2 compass_2

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity compass_2 is port(
clk, razn, in_pwm, continu, start_stop :in std_logic;
data_valid: out std_logic:='0';
  out_1s : out std_logic;
data_compas : out std_logic_vector(8 downto 0);
out_pwm : out std_logic

);
end entity;

architecture C of compass_2 is

signal sig1,sig5,pwm_sig,sig_data_valid : std_logic;
signal sig2,sig3,sig6,sig7,sig8,sig_continu,sig_ss : std_logic_vector(8 downto 0);
signal sig4 : std_logic_vector(1 downto 0);

component div_compa is
port (
  clk_div : in std_logic;
  S_div: out std_logic);
end component;

component div_compa2 is
port (
  clk_div : in std_logic;
  S_div: out std_logic);
end component;

component deg_compa is
port(
  deg_in,in_pwm : in std_logic;
  deg_s, deg_s2: out std_logic_vector(8 downto 0));
end component;

component pwm_q18 is
port(
  clk_pwm, reset_pwm: in std_logic;
  --freq_pwm, duty_pwm: in std_logic_vector(7 downto 0);
  s_pwm : out std_logic
);
end component;
```

```

component continu_component is
port(
    clk_continu : in std_logic;
    sig_entree,sig_entree2: in std_logic_vector(8 downto 0);
    --sorite_valid: out std_logic;
    sortie: out std_logic_vector(8 downto 0));
end component;

component ss_component is
port(
    start,raz : in std_logic;
    sig_entree,sig_entree2: in std_logic_vector(8 downto 0);
    --sorite_valid: out std_logic;
    sortie: out std_logic_vector(8 downto 0));
end component;

begin
dv_compas : div_compa
port map ( clk_div=>clk , S_div=>sig1 );
dv_compas_2 : div_compa2
port map ( clk_div=>sig1 , S_div=>sig5 );
dv_pwm : pwm_q18
port map(clk_pwm=>clk, reset_pwm=>'0',s_pwm=>pwm_sig);
dv_deg : deg_compa
port map(deg_in=>sig1, in_pwm=>pwm_sig, deg_s=>sig6, deg_s2=>sig8);
dv_continu: continu_component
port map(clk_continu=>sig5, sig_entree=>sig6 , sig_entree2=>sig8, sortie=>sig_continu);
dv_ss: ss_component
port map(start=>start_stop, sig_entree=>sig6, raz=>razn , sig_entree2=>sig8, sortie=>sig_ss)

process(clk)
begin
if clk 'event and clk = '1' then
if razn ='1' then
if start_stop='1' then
sig7<=sig_ss;
sig_data_valid<='1';
elsif start_stop='0' then
sig_data_valid<='0';
end if;
elsif razn ='0' then
sig7<="000000000";
end if;
end if;

end process;

out_1s<=sig5;

```



```
data_compas<=sig7;  
out_pwm<=pwm_sig;  
data_valid<=sig_data_valid;  
  
end architecture;
```

7.2.3 nios_C.c

```
/*
 *Written by: Hichen Salah
 *Written by: Hamdan Feras
 */
#include <system.h>
#include <alt_types.h>
#include <altera_avalon_pio_regs.h>
#include <stdio.h>

unsigned char flag,switches,compass,continu,start_stop,sortie,data_valid;
int degre;
int main()
{
    printf("ZIDANE !!!!!\n");

    while(1)
    {
        switches = IORD_ALTERA_AVALON_PIO_DATA(SWITCH_BASE);
        //compass = IORD_ALTERA_AVALON_PIO_DATA(S_COMPASS_BASE);

        continu = switches & 0x1;
        start_stop = switches & 0x2;

        if(start_stop==0x2)
            start_stop=1;

        if(continu&&01==1)
        {
            IOWR_ALTERA_AVALON_PIO_DATA(SS_BASE,1);
            degre = IORD_ALTERA_AVALON_PIO_DATA(DEGRE_BASE);
            printf("data compass = %d\n",degre);
            IOWR_ALTERA_AVALON_PIO_DATA(SS_BASE,0);
            IOWR_ALTERA_AVALON_PIO_DATA(LED0_DATA,1);

            usleep(1000000);

        }
        else if(start_stop==1 && flag==1)
        {
            flag=0;
            IOWR_ALTERA_AVALON_PIO_DATA(SS_BASE,0);
            usleep(20000);
            IOWR_ALTERA_AVALON_PIO_DATA(SS_BASE,1);
            degre = IORD_ALTERA_AVALON_PIO_DATA(DEGRE_BASE);
            printf("data compass = %d\n",degre);
            IOWR_ALTERA_AVALON_PIO_DATA(LED0_DATA,!flag);

        }
        else if(start_stop==0)
```

```

{ flag=1;
  IOWR_ALTERA_AVALON_PIO_DATA(LED_S_DEGRE_BASE,!flag);
}

data_valid = IORD_ALTERA_AVALON_PIO_DATA(DATA_VALID_IN_BASE);

/*  degree = IORD_ALTERA_AVALON_PIO_DATA(DEGRE_BASE);
    printf("degree=%d\n",degree);
    usleep(5000000);*/

}

return 0;
}

```

7.3 Gestion du convertisseur

7.3.1 Gestion du convertisseur

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity SPI_ADC is port(
    clk          : in  std_logic;
    reset        : in  std_logic;
    cmd          : in  std_logic;
    data_in      : in  std_logic;
    cs           : out std_logic:= '1';
    clk_1meg     : out std_logic;
    ack          : out std_logic;
    --daa        : out std_logic;
    --meg1       : out std_logic;
    data_out     : out std_logic_vector(11 downto 0);
    LED_out      : out std_logic_vector(11 downto 0)
);
end entity;

architecture ADC of SPI_ADC is
    type state is (
        E_initial,
        E_begin,
        E_attente,
        E_lecture,
        E_end
    );

    signal EP, ES : state;

    signal clk_1m          : std_logic;
    signal timer_100      : std_logic;
    signal timer_1        : std_logic;
    signal start_timer_100 : std_logic;
    signal start_timer_1   : std_logic;
    signal start_lecture   : std_logic;
    signal start_cpt_clk   : std_logic;
    signal start_cpt_data  : std_logic;
    signal cpt_clk_1m      : std_logic_vector(4 downto 0);
    signal cpt_clk         : std_logic_vector(1 downto 0);
    signal cpt_data        : std_logic_vector(3 downto 0);
    signal compteur_100    : std_logic_vector(2 downto 0);
    signal compteur_1      : std_logic_vector(5 downto 0);
    signal data            : std_logic_vector(11 downto 0);
```

```

signal data_memoire          : std_logic_vector(11 downto 0);
signal sig_ack               : std_logic;
begin
    process(clk,reset)
    begin
        if (reset = '0') then
            EP <= E_end;
        elsif clk'event and clk='1' then
            EP <= ES;
        end if;
    end process;

    process(EP, cmd, timer_100, cpt_clk, cpt_data, timer_1)
    begin
        case EP is
            when E_initial =>
                if cmd = '0' then
                    ES <= E_begin;
                else
                    ES <= EP;
                end if;

            when E_begin =>
                if timer_100 = '1' then
                    ES <= E_attente;
                else
                    ES <= EP;
                end if;

            when E_attente =>
                if cpt_clk = "10" then
                    ES <= E_lecture;
                else
                    ES <= EP;
                end if;

            when E_lecture =>
                if cpt_data = "1111" then
                    ES <= E_end;
                else
                    ES <= EP;
                end if;

            when E_end =>
                if timer_1 = '1' then
                    ES <= E_initial;
                else
                    ES <= EP;
                end if;
        end case;
    end process;
end;

```

```

end process;

process(EP)
begin
case EP is
when E_initial =>
    --Cs <= '1';
    start_timer_1 <= '0';

when E_begin =>
    Cs <= '0';
    sig_ack <= '0';
    start_timer_100 <= '1';

when E_attente =>
    Start_cpt_clk <= '1';
    start_timer_100 <= '0';

when E_lecture =>
    Start_lecture <= '1';
    Start_cpt_clk <= '0';
    start_cpt_data <= '1';

when E_end =>
    Cs <= '1';
    sig_ack <= '1';
    start_timer_1 <= '1';
    Start_lecture <= '0';
    start_cpt_data <= '0';

end case;
end process;

-- end machine d'état--

process(clk) --clk 1 Mhz
begin
if clk'event and clk='1' then
    cpt_clk_1m <= cpt_clk_1m + 1;
if cpt_clk_1m = "11000" then
    clk_1m <= not clk_1m;
    cpt_clk_1m <= "00000";
end if;
end if;

end process;

process(clk) --compteur 100ns
begin

```

```

        if clk'event and clk='1' then
            if start_timer_100 = '0' then
                timer_100 <= '0';
            elsif start_timer_100 = '1' then
                compteur_100 <= compteur_100 + 1;
            if compteur_100 = "101" then
                timer_100 <='1';
                --start_timer_100 <= '0';
                compteur_100 <= "000";
            end if;
        end if;
    end if;

end process;

process(clk) --compteur 1us
begin
    if clk'event and clk='1' then
        if start_timer_1 = '0' then
            timer_1 <= '0';
        elsif start_timer_1 = '1' then
            compteur_1 <= compteur_1 + 1;
        if compteur_1 = "110010" then
            timer_1 <='1';
            --start_timer_1 <= '0';
            compteur_1 <= "000000";
        end if;
    end if;
end if;

end process;

process(clk_1m) --compteur 2
begin
    if clk_1m'event and clk_1m='1' then
        if Start_cpt_clk = '0' then
            cpt_clk <= "00";
        else
            cpt_clk <= cpt_clk + 1;
        --if cpt_clk = "10" then
        --Start_cpt_clk <= '0';
        --end if;
    end if;
end if;

end process;

process(clk_1m) --compteur data
begin

```

```

        if clk_1m'event and clk_1m='1' then
        if Start_cpt_data = '0' then
            cpt_data <= "0000";
        else
            cpt_data <= cpt_data + 1;
            --if cpt_data = "1101" then
                --Start_cpt_data <= '0';
            --end if;
        end if;
    end if;

end process;

process(clk_1m) -- lecture data
begin
    if clk_1m'event and clk_1m='1' then
        if start_lecture = '0' then
            data_memoire <= data;
        else
            data(11 downto 0) <= data(10 downto 0) & data_in;
        end if;
    end if;
end process;

ack <= sig_ack;
data_out <= data_memoire;
LED_out <= data_memoire;
clk_1meg <= clk_1m;
--daa<=data_in;
--meg1<=clk_1m;
end architecture;

```


7.3.2 Gestion du convertisseur avec NiosII

7.3.3 Top entity VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity SPI_main is port(
    clk_main          : in  std_logic;
    reset_main        : in  std_logic;
    data_in_main      : in  std_logic;
    cs_main           : out std_logic;
    csl               : out std_logic;
    led               : out std_logic_vector(11 downto 0);
    clk_1meg_main     : out std_logic
);
end entity SPI_main;

architecture SPI of SPI_main is

    signal    sig_cmd          : std_logic;
    signal    sig_ack          : std_logic;
    signal    sig_data_out     : std_logic_vector(11 downto 0);
    signal    sig_led          : std_logic_vector(11 downto 0);

    signal    cs_sig          : std_logic;

    component nios_spi is
        port (
            acknowledge_external_connection_export : in  std_logic
            clk_clk                               : in  std_logic
            command_external_connection_export     : out std_logic;
            data_external_connection_export        : in  std_logic_vector(11 downto 0)
            reset_reset_n                         : in  std_logic
        );
    end component;

    component SPI_ADC is
        port (
            clk          : in  std_logic;
            reset        : in  std_logic;
            cmd           : in  std_logic;
            data_in       : in  std_logic;
            cs            : out std_logic;
            clk_1meg      : out std_logic;
            ack           : out std_logic;
            data_out      : out std_logic_vector(11 downto 0);
            LED_out       : out std_logic_vector(11 downto 0)
        );
    end component;

end architecture SPI;
```

```

    );
end component;

begin

nios : nios_spi
port map
(
    acknowledge_external_connection_export => sig_ack,
    clk_clk                                => clk_main,
    command_external_connection_export     => sig_cmd,
    data_external_connection_export        => sig_data_out,
    reset_reset_n                          => reset_main
);

Convertisseur      : SPI_ADC
port map
(
    clk              => clk_main,
    reset            => reset_main,
    cmd              => sig_cmd,
    data_in          => data_in_main,
    cs               => cs_main,
    clk_1meg => clk_1meg_main,
    ack             => sig_ack,
    data_out => sig_data_out,
    LED_out  => led

);

csl<=sig_cmd;
--cs_main<=cs_sig;

end architecture;

```

7.3.4 Nios C

```
/*
 * "Hello World" example.
 *
 * This example prints 'Hello from Nios II' to the STDOUT stream. It runs on
 * the Nios II 'standard', 'full_featured', 'fast', and 'low_cost' example
 * designs. It runs with or without the MicroC/OS-II RTOS and requires a STDOUT
 * device in your system's hardware.
 * The memory footprint of this hosted application is ~69 kbytes by default
 * using the standard reference design.
 *
 * For a reduced footprint version of this template, and an explanation of how
 * to reduce the memory footprint for a given application, see the
 * "small_hello_world" template.
 */

#include <stdio.h>
#include <alt_types.h>
#include <altera_avalon_pio_regs.h>
#include <system.h>
#include <unistd.h>

int data_adc;
int tension;

int main()
{
    printf("Hello from Nios II!\n");
    while(1)
    {
        IOWR_ALTERA_AVALON_PIO_DATA(COMMAND_BASE,0);
        usleep(2);
        IOWR_ALTERA_AVALON_PIO_DATA(COMMAND_BASE,1);
        while(IORD_ALTERA_AVALON_PIO_DATA(ACKNOWLEDGE_BASE) == 0);
        data_adc = IORD_ALTERA_AVALON_PIO_DATA(DATA_BASE);
        tension = data_adc*5000/4095;
        printf("La tension lu: %d (en mV) \n",tension);
        usleep(1000000);

    }

    return 0;
}
```

7.3.5 Compas et Gestion du convertisseur

7.3.6 Top entity VHDL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity adc_comp is port(
    clk                      : in  std_logic;
    reset                    : in  std_logic;
    data_in                  : in  std_logic;
    degre_in                 : in  std_logic;
    clk_1meg                 : out std_logic;
    cs                       : out std_logic;
    continue                 : in  std_logic;
    ss                       : in  std_logic;
    data_valid               : out std_logic;
    led                      : out std_logic_vector(11 downto 0);

    out_pwm                 : out std_logic
);
end entity adc_comp;

architecture main of adc_comp is

    signal sig_cmd_compas      : std_logic;
    signal sig_cmd_adc         : std_logic;
    signal sig_ack             : std_logic;
    signal sig_data_out ,sig_data_outs : std_logic_vector(11 downto 0);
    --signal sig_led           : std_logic_vector(11 downto 0);
    signal cs_sig              : std_logic;
    signal position_sig        : std_logic_vector(11 downto 0);
    signal data_compas_sig     : std_logic_vector(8 downto 0);
    signal data_valid_sig      : std_logic;
    signal out_ls_sig          : std_logic;

    component adc_cmp is
        port (
            acknowledge_external_connection_export : in  std_logic
            clk_clk                                : in  std_logic
            cmd_adc_external_connection_export     : out std_logic;
            cmd_compas_external_connection_export  : out std_logic;
            continue_external_connection_export    : in  std_logic
            data_valid_external_connection_export  : out std_logic;
            degre_in_external_connection_export    : in  std_logic_vector(8 downto 0)
            position_in_external_connection_export : in  std_logic_vector(11 downto 0)
            reset_reset_n                          : in  std_logic
            ss_external_connection_export           : in  std_logic
```

```

    );
end component;

component compass_2 is
port(
    clk, razn, in_pwm, continu, start_stop :in std_logic;
    data_valid: out std_logic:='0';
    out_1s : out std_logic;
    data_compas : out std_logic_vector(8 downto 0);
    out_pwm : out std_logic

);
end component;

component SPI_ADC is
    port (
        clk                : in  std_logic;
        reset               : in  std_logic;
        cmd                 : in  std_logic;
        data_in             : in  std_logic;
        cs                  : out std_logic;
        clk_1meg            : out std_logic;
        ack                 : out std_logic;
        data_out            : out std_logic_vector(11 downto 0);
        LED_out             : out std_logic_vector(11 downto 0)

    );
end component;

begin

nios : adc_cmp
port map
    (
        sig_ack,
        clk,
        sig_cmd_adc,
        sig_cmd_compas,
        continue,
        data_valid,
        data_compas_sig,
        sig_data_out,
        reset,
        ss

    );

Convertisseur          : SPI_ADC
port map
    (
        clk,
        reset,

```

```

        sig_cmd_adc,
        data_in,
        cs,
        clk_1meg,
        sig_ack,
        sig_data_out,
        led

    );

Compas : compass_2
port map
(
    clk,
    reset,
    degre_in,
    '0',
    sig_cmd_compas,
    data_valid_sig,
    out_1s_sig,
    data_compas_sig,
    out_pwm
);

--csl<=sig_cmd;
--cs_main<=cs_sig;

end architecture;
```

7.3.7 Nios C (version pour test)

```
/*
 * "Hello World" example.
 *
 * This example prints 'Hello from Nios II' to the STDOUT stream. It runs on
 * the Nios II 'standard', 'full_featured', 'fast', and 'low_cost' example
 * designs. It runs with or without the MicroC/OS-II RTOS and requires a STDOUT
 * device in your system's hardware.
 * The memory footprint of this hosted application is ~69 kbytes by default
 * using the standard reference design.
 *
 * For a reduced footprint version of this template, and an explanation of how
 * to reduce the memory footprint for a given application, see the
 * "small_hello_world" template.
 */

#include <stdio.h>
#include <system.h>
#include <alt_types.h>
#include <altera_avalon_pio_regs.h>
#include <altera_avalon_timer_regs.h>
#include <sys/alt_irq.h>

int degre, data_adc, tension, flag=0;

unsigned char switches, compass, continu, start_stop, sortie, data_valid;

static void timer_irq(void* context)
{
    flag++;
    if (flag==10){
        //compas
        flag=0;
        IOWR_ALTERA_AVALON_PIO_DATA(CMD_COMPAS_BASE,1);
        degre = IORD_ALTERA_AVALON_PIO_DATA(DEGRE_IN_BASE);
        printf("data compass = %d\n",degre);
        IOWR_ALTERA_AVALON_PIO_DATA(CMD_COMPAS_BASE,0);
        printf("La tension lu: %d (en mV) \n",tension);

        //IOWR_ALTERA_AVALON_PIO_DATA(LEDs_DEGRE_BASE,1)
    }
    //ADC
    IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,0);
    usleep(2);
    IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,1);
    while(IORD_ALTERA_AVALON_PIO_DATA(ACKNOWLEDGE_BASE) == 0);
    data_adc = IORD_ALTERA_AVALON_PIO_DATA(POSITION_IN_BASE);
    tension = data_adc*5000/4095;
    //printf("La tension lu: %d (en mV) \n",tension);
    //usleep(100000);
}
```

```

        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_ADC_BASE,0);
    }
    int main()
    {
        printf("Hello from Nios II!\n");

        alt_irq_cpu_enable_interrupts();
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_ADC_BASE,0x0003);
        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_ADC_BASE,0);
        alt_ic_isr_register(TIMER_ADC_IRQ_INTERRUPT_CONTROLLER_ID,
                           TIMER_ADC_IRQ,
                           timer_irq,
                           0,
                           0);
        IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_ADC_BASE,0x0007);

        while(1){
            /*
                IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,0);
                usleep(2);
                IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,1);
                while(IORD_ALTERA_AVALON_PIO_DATA(ACKNOWLEDGE_BASE) == 0);
                data_adc = IORD_ALTERA_AVALON_PIO_DATA(POSITION_IN_BASE);
                tension = data_adc*5000/4095;
                printf("La tension lu: %d (en mV) \n",data_adc);
                //
                usleep(1000000);
            */
        }

        return 0;
    }

```


7.3.8 Nios C

```
#include <stdio.h>
#include <system.h>
#include <alt_types.h>
#include <altera_avalon_pio_regs.h>
#include <altera_avalon_timer_regs.h>
#include <sys/alt_irq.h>

int degree, data_adc, tension, flag=0;

unsigned char switches, compass, continu, start_stop, sortie, data_valid;

static void timer_irq(void* context)
{
    flag++;
    if (flag==10){
        //compas
        flag=0;
        if((IORD_ALTERA_AVALON_PIO_DATA(CONTINUE_BASE)==1) {
            IOWR_ALTERA_AVALON_PIO_DATA(CMD_COMPAS_BASE,1);
            degree = IORD_ALTERA_AVALON_PIO_DATA(DEGRE_IN_BASE);
            printf("data compass = %d\n",degree);
            IOWR_ALTERA_AVALON_PIO_DATA(CMD_COMPAS_BASE,0);
            IOWR_ALTERA_AVALON_PIO_DATA(DATA_VALID_BASE,0)

        }
        //ADC
        IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,0);
        usleep(2);
        IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,1);
        while(IORD_ALTERA_AVALON_PIO_DATA(ACKNOWLEDGE_BASE) == 0);
        data_adc = IORD_ALTERA_AVALON_PIO_DATA(POSITION_IN_BASE);
        tension = data_adc*5000/4095;
        printf("La tension lu: %d (en mV) \n",tension);

        IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_ADC_BASE,0);
    }
}

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

    alt_irq_cpu_enable_interrupts();
    IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_ADC_BASE,0x0003);
    IOWR_ALTERA_AVALON_TIMER_STATUS(TIMER_ADC_BASE,0);
    alt_ic_isr_register(TIMER_ADC_IRQ_INTERRUPT_CONTROLLER_ID,
                        TIMER_ADC_IRQ,
                        timer_irq,
                        0,
                        0);
}
```

```

IOWR_ALTERA_AVALON_TIMER_CONTROL(TIMER_ADC_BASE,0x0007);

while(1){
    if((IORD_ALTERA_AVALON_PIO_DATA(SS_BASE)==1 && ((IORD_ALTERA_AVALON_PIO_DATA(CONTINU
    {
        IOWR_ALTERA_AVALON_PIO_DATA(DATA_VALID_BASE,0)
        IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,0);
        usleep(2);
        IOWR_ALTERA_AVALON_PIO_DATA(CMD_ADC_BASE,1);
        while(IORD_ALTERA_AVALON_PIO_DATA(ACKNOWLEDGE_BASE) == 0);
        data_adc = IORD_ALTERA_AVALON_PIO_DATA(POSITION_IN_BASE);
        tension = data_adc*5000/4095;
        printf("La tension lu: %d (en mV) \n",data_adc);
        IOWR_ALTERA_AVALON_PIO_DATA(DATA_VALID_BASE,1)
    }

}

return 0;
}

```