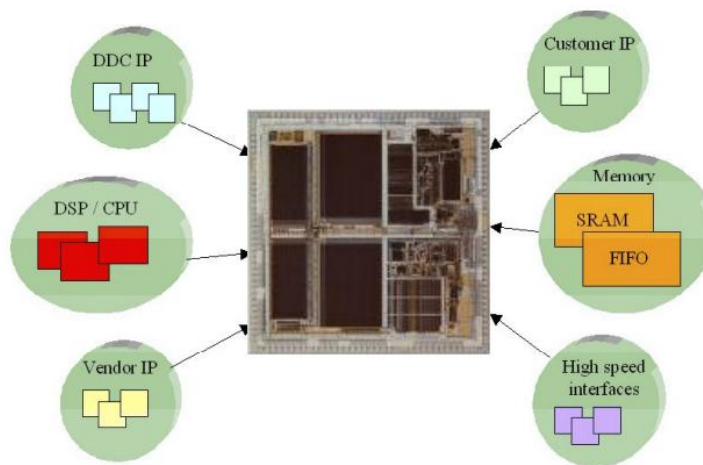


## Mini-Projet Radar 2D



**MOUSSY Alexandre, RUYNAT Eddy**

**5A FISE**

**Cours : Architecture des systèmes embarqués**

**Enseignant : Yann DOUZE**

### Table des matières

1. Introduction	4
2. Matériel	5
I. FPGA	5
II. Principe de Fonctionnement (HC-SR04)	6
III. Servo Moteur	6
3. Le Télémètre	7
I. Simulation de IP	7
II. Test télémètres sur la carte	12
III. Intégration de l'IP Avalon Télémètre dans Platform Designer	14
IV. Programmation logicielle et test de l'IP Télémètre	17
4. Conception de l'IP Servomoteur	19
I. Simulation IP	19
Simulation de notre IP servomoteur	19
Simulation de IP Avalon du servomoteur	23
II. Programmation logicielle et test de l'IP Servomoteur	26
5. Affichage des obstacles	28
6. Conclusion	31

## Table des illustrations

Figure 1: Illustration entre sortie VHDL télémètre	7
Figure 2 : Code télémètre	9
Figure 3 : Code tb télémètre	11
Figure 4 : Test 1 mesure de 0 cm	11
Figure 5 : Test 2 mesure 20cm	12
Figure 6 : Test 3 mesure 4M	12
Figure 8 : Code TOP level télémète	13
Figure 9 : Resultat Télémètre	14
Figure 10 : Code TOP level télémète Avalon	15
Figure 11 : Test 1 mesure 0 cm	16
Figure 12 : Test 2 mesure 20 cm	16
Figure 13 : Test 3 mesure 4 m	17
Figure 14 : Code C Télémètre	17
Figure 15 : Résultat affichage valeur télémètre sur la terminale	18
Figure 16 : Affichage 7seg	18
Figure 17 : Code de servomoteur	20
Figure 18 : Code de tb servomoteur	21
Figure 19 : Simulation commande pwm	21
Figure 20 : Top level servomotor	23
Figure 21 : TestBench servomoteur avalon	24
Figure 22 : Simulation PWM Avalon	24
Figure 23 : Top level Télémètre et servomoteur	26
Figure 24 : Code C servomoteur	27
Figure 25 : Résultat code C servomoteur	27
Figure 26 : Code C radar 2D	29
Figure 27 : Résultat radar 2D	29

### 1. Introduction

Dans le cadre du mini-projet *Radar 2D*, nous avons développé un système complet combinant électronique numérique, VHDL, communication Avalon et programmation embarquée. L'objectif principal était de mesurer la distance d'objets grâce à un télémètre ultrason HC-SR04 et de cartographier un secteur de 180° en faisant pivoter le capteur à l'aide d'un servomoteur commandé par PWM. Pour atteindre ce résultat, nous avons conçu deux IP matérielles distinctes :

- Une IP Télémètre capable de générer un signal Trigger précis et d'analyser la largeur du signal Echo
- Une IP Servomoteur assurant une modulation d'impulsion fidèle aux spécifications mécaniques du moteur.

Ces deux modules ont été entièrement décrits en VHDL, simulés sous ModelSim puis intégrés matériellement sur la carte DE10-Lite.

La seconde étape a consisté à connecter ces IP au processeur Nios II via le bus Avalon, permettant une commande logicielle et centralisée du système. Cette intégration a nécessité la création de wrappers Avalon-MM, l'utilisation de Platform Designer, la gestion du mapping mémoire, ainsi que l'écriture de programmes en langage C pour orchestrer le balayage angulaire, lire la distance mesurée et afficher les résultats sur les différents périphériques de la carte (LED, afficheurs 7-segments, console). Grâce à cette architecture hybride matériel-logiciel, nous avons pu réaliser un véritable mini-radar embarqué, capable de détecter et d'afficher des obstacles en temps réel sur un angle complet.

.

## 2. Matériel

### I. FPGA

Le cœur de la carte est un processeur programmable Intel MAX 10 le 10M50DAF484C7G. C'est un FPGA "non-volatile", ce qui signifie qu'il conserve sa configuration même après mise hors tension.

Caractéristiques Techniques du Chip :

- Éléments Logiques LEs : 50 000. Ce sont les blocs de base pour créer des circuits logiques.
- Mémoire Interne : 1 638 Kbits. Utilisée pour le stockage de données rapide ou les tampons (buffers).
- Mémoire Flash Utilisateur : 5 888 Kbits. Permet de stocker du code ou des données non-volatiles.
- Multiplicateurs 18x18 : 144. Blocs dédiés pour le calcul mathématique rapide (DSP).
- Boucles à verrouillage de phase : 4. Utilisées pour générer différentes fréquences d'horloge à partir d'une seule source.
- Convertisseur Analogique-Numérique : 1 bloc avec 1 entrée dédiée et 8 entrées partagées.

La carte Terasic DE10-Lite intègre un ensemble complet de ressources matérielles destinées à faciliter l'expérimentation numérique et le prototypage FPGA. Elle dispose notamment d'une mémoire SDRAM externe de 64 Mo, utile pour héberger des systèmes plus complexes tels qu'un processeur NIOS II ou pour stocker des données volumineuses. Pour l'interaction utilisateur, la carte propose 10 interrupteurs glissants, 2 boutons-poussoirs, 10 LEDs rouges, ainsi que 6 afficheurs 7-segments, permettant un large éventail d'entrées et d'affichages. La DE10-Lite offre également de nombreuses possibilités de connectivité, avec un accéléromètre 3 axes ADXL345, une sortie VGA (4 bits par couleur), et un connecteur GPIO à 40 broches permettant la connexion de périphériques externes tels qu'un télémètre ultrason ou un servomoteur. Enfin, la carte intègre un USB-Blaster directement embarqué, simplifiant la programmation et le débogage du FPGA via un simple câble USB.

### II. Principe de Fonctionnement (HC-SR04)

Le capteur HC-SR04 fonctionne selon le protocole suivant :

- Déclenchement (Trigger) : On envoie une impulsion haute de 10  $\mu\text{s}$  sur l'entrée Trig.
- Emission : Le capteur émet 8 salves d'ultrasons à 40 kHz.
- Réception (Echo) : Le signal Echo passe au niveau haut. Il reste haut jusqu'à ce que le récepteur détecte l'onde réfléchie.

La durée de l'impulsion Echo est proportionnelle à la distance parcourue.

- Vitesse du son :  $v \approx 340 \text{ m/s}$  (soit 1 cm  $\approx 29 \mu\text{s}$  de trajet simple).
- Aller-retour : 1 cm (aller-retour)  $\approx 58 \mu\text{s}$ .
- Formule : Distance (cm) =  $\frac{\text{Temps Echo } (\mu\text{s})}{58}$

### III. Servo Moteur

Le servomoteur est piloté par un signal MLI (Modulation de Largeur d'Impulsion) ou PWM. La position de l'axe est déterminée par la durée de l'impulsion haute, répétée à un intervalle régulier.

Spécifications temporelles (Standard) :

- Fréquence de rafraîchissement : L'impulsion doit être répétée toutes les 20 ms (50 Hz).
- Durée de l'impulsion ( $T_{high}$ ) :
  - ms  $\rightarrow$  Position 0°
  - 1.5 ms  $\rightarrow$  Position 90° (Centre)
  - 2.0 ms  $\rightarrow$  Position 180°

*Note : Certains modèles peuvent varier de 0.5 ms à 2.5 ms.*

### 3. Le Télémètre

#### I. Simulation de IP

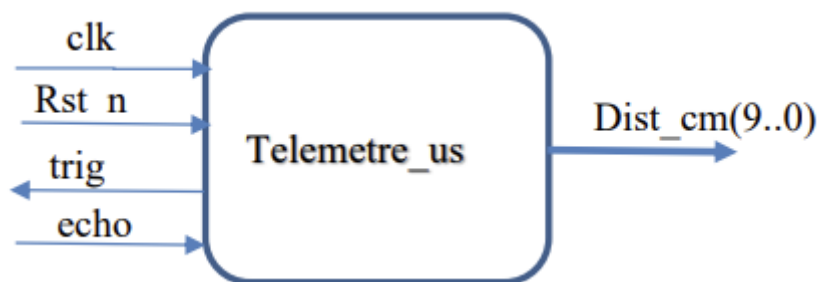


Figure 1: Illustration entre sortie VHDL télémètre

Dans un premier temps nous allons coder le télémètre en VHDL. L'IP a été réalisée selon une approche comportementale pour assurer une gestion précise des timings imposés par le capteur ultrason.

L'ensemble de la logique est piloté par une horloge de 50 MHz et s'articule autour de trois fonctions principales :

Gestion du Cycle Global (60 ms) :

- Le capteur HC-SR04 nécessite un intervalle d'au moins 60 ms entre deux mesures pour éviter que les échos résiduels ne perturbent la mesure suivante.
- Un compteur cnt\_global gère ce cycle complet (3 000 000 tops d'horloge).

Génération du Trigger :

- Au début de chaque cycle Global le module génère une impulsion trig de 10  $\mu$ s envoyée au capteur pour lancer une mesure.
- Quand le capteur reçoit le trig il envoie 8 impulsions ultrasoniques de 40 kHz. Ces impulsions vont rebondir sur une surface avant de revenir au capteur. Le temps de propagation sera matérialisé par le signal Echo.

Calcul de la Distance (Echo) :

- Synchronisation : Comme le signal Echo mesure une valeur physique il est asynchrone, pour éviter tout blocage nous utilisons un système de double bascule (echo\_sync, echo\_prev) pour le resynchroniser avec notre horloge.
- Nous avons décidé que notre capteur aura une précision au CM près. La physique du son indique qu'un aller-retour de 1 cm prend environ 58  $\mu$ s.
- L'IP que nous avons créée incrémente un compteur appelé cnt\_echo jusqu'à la valeur de 2 900 cycles ce qui représente 58  $\mu$ s. C'est à chaque fois que ce

compteur et remis à 0 que la distance en centimètres (distance\_buffer) est incrémentée de 1.

Sortie :

- Le système surveille le signal echo pour détecter le moment où il retombe à '0' (front descendant, fin de la mesure).
- À cet instant précis la valeur finale du compteur distance\_buffer est sauvegardée dans le registre dist\_cm\_reg
- dist\_cm\_reg et un registre qui permet si on le souhaite d'utiliser la valeur mesurée avant de l'envoyer sur une broche output ou elle ne sera plus récupérable
- Cette broche output est appelée à au front descendant du signal Echo ce qui envoie la valeur mesurée en dehors de l'IP

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

entity telemetre_us is
    port (
        clk      : in  std_logic;           -- Horloge 50 MHz
        rst_n    : in  std_logic;           -- Reset actif bas
        echo     : in  std_logic;           -- Entrée Echo du capteur
        trig     : out std_logic;           -- Sortie Trigger vers le capteur
        dist_cm  : out std_logic_vector(9 downto 0) -- Distance mesurée en cm
    );
end entity;

architecture bhv of telemetre_us is
    -- Constantes pour 50 MHz (période 20ns)
    -- 60 ms = 60,000,000 ns / 20 ns = 3,000,000 cycles
    constant CNT_60MS_MAX : integer := 3000000;
    -- 10 µs = 10,000 ns / 20 ns = 500 cycles
    constant CNT_10US_MAX : integer := 500;
    -- 58 µs = 58,000 ns / 20 ns = 2900 cycles (Temps pour parcourir 1 cm aller-retour)
    constant CNT_58US_MAX : integer := 2900;

    signal cnt_global      : integer range 0 to CNT_60MS_MAX := 0;
    signal cnt_echo        : integer range 0 to CNT_58US_MAX := 0;
    signal distance_buffer : unsigned(9 downto 0) := (others => '0');
    signal echo_sync       : std_logic;
    signal echo_prev       : std_logic;
    signal dist_cm_reg     : std_logic_vector(9 downto 0) := (others => '0');

begin
    -- Processus principal
    process(clk, rst_n)
    begin
        if rst_n = '0' then
            cnt_global      <= 0;
            cnt_echo        <= 0;
            trig            <= '0';
            distance_buffer <= (others => '0');
            echo_sync       <= '0';
            echo_prev       <= '0';
            dist_cm_reg     <= (others => '0');
        elsif rising_edge(clk) then
            -- Synchronisation du signal echo
            echo_sync <= echo;
            echo_prev <= echo_sync;

            -- Gestion du Trigger et du cycle de 60ms
            if cnt_global < CNT_60MS_MAX then
                cnt_global <= cnt_global + 1;
            end if;
        end if;
    end process;
end architecture;
```

```
else
    cnt_global <= 0;
end if;

-- Génération du pulse Trigger
if cnt_global < CNT_10US_MAX then
    trig <= '1';
else
    trig <= '0';
end if;

-- Mesure de la largeur d'impulsion de l'Echo
if echo_sync = '1' then
    if cnt_echo < CNT_58US_MAX then
        cnt_echo <= cnt_echo + 1;
    else
        -- Chaque fois qu'on atteint 58µs, on incrémente la distance de 1cm
        cnt_echo <= 0;
        if distance_buffer < 400 then -- Limite max 400 cm
            distance_buffer <= distance_buffer + 1;
        end if;
    end if;
else
    -- Si echo est bas, on remet le compteur à 0
    cnt_echo <= 0;

    -- Detect Falling Edge of Echo to update output
    if echo_prev = '1' and echo_sync = '0' then
        dist_cm_reg <= std_logic_vector(distance_buffer);
    end if;

    -- On reset le buffer au début du prochain cycle de trigger pour une nouvelle mesure
    if cnt_global = 0 then
        distance_buffer <= (others => '0');
    end if;
end if;
end if;
end process;

dist_cm <= dist_cm_reg;

end architecture;
```

Figure 2 : Code télémètre

Ensuite pour tester le bon fonctionnement de notre code télémètre on réalise un testbench. La simulation des Waves a été réalisée sous ModelSim le bon fonctionnement de la logique crée avant de l'envoyée sur carte.

Le banc de test simule le comportement physique du capteur HC-SR04 :

- Il attend la détection du front montant sur le signal trig ce qui déclenche l'envoi des ondes hypersoniques.
- Après cela nous avons forcé le signal il force le signal Echo à '1' pendant une durée précise. Cela simule le temp de réceptions des ondes envoyer. Et pour notre test nous avons choisie 3 valeurs simulent 0 cm, 20 cm et 400cm.

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

entity tb_telemetre_us is
end entity;

architecture test of tb_telemetre_us is

    component telemetre_us
        port (
            clk      : in  std_logic;
```

```
        rst_n      : in  std_logic;
        echo       : in  std_logic;
        trig       : out std_logic;
        dist_cm    : out std_logic_vector(9 downto 0)
    );
end component;

signal clk      : std_logic := '0';
signal rst_n    : std_logic := '0';
signal echo     : std_logic := '0';
signal trig     : std_logic;
signal dist_cm  : std_logic_vector(9 downto 0);

-- Clock period definitions
constant clk_period : time := 20 ns; -- 50 MHz

begin

    -- Instantiate the Unit Under Test (UUT)
    uut: telemetre_us
        port map (
            clk      => clk,
            rst_n    => rst_n,
            echo     => echo,
            trig     => trig,
            dist_cm  => dist_cm
        );

    -- Clock process definitions
    clk_process :process
    begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;

    -- Stimulus process
    stim_proc: process
    begin
        -- Hold reset state for 100 ns.
        rst_n <= '0';
        wait for 100 ns;
        rst_n <= '1';

        wait for 100 ns;

        -- =====
        -- MESURE 1 : 0 cm
        -- =====
        wait until rising_edge(trig);
        report "Trigger 1 détecté !";

        wait for 200 us;

        -- Simulation Echo pour 0 cm (durée < 58us)
        report "Simulation de l'Echo pour 0 cm (10 us)";
        echo <= '1';
        wait for 10 us;
        echo <= '0';

        -- =====
        -- MESURE 2 : 20 cm
        -- =====
        wait until rising_edge(trig);
        report "Trigger 2 détecté !";

        wait for 200 us;

        -- Simulation Echo pour 20 cm
        report "Simulation de l'Echo pour 20 cm (1160 us)";
        echo <= '1';
        wait for 1160 us;
        echo <= '0';
```

```
-- =====
-- MESURE 3 : 400 cm
-- =====
wait until rising_edge(trig);
report "Trigger 3 détecté !";

wait for 200 us;

-- Simulation Echo pour 400 cm
report "Simulation de l'Echo pour 400 cm (23.2 ms)";
echo <= '1';
wait for 23200 us;
echo <= '0';

-- EXTENSION DE LA DUREE
report "Attente résultat final...";
wait for 100 ms; -- On attend largement assez pour voir le reset et la fin du cycle

assert false report "Fin de la Simulation" severity failure;
end process;

end architecture;
```

Figure 3 : Code tb télémètre

Le banc de test simule le comportement du capteur physique pour vérifier la précision de la mesure.

Scénarios de test :

1. Mesure de 0 cm : mesure vide sans aucun echo.

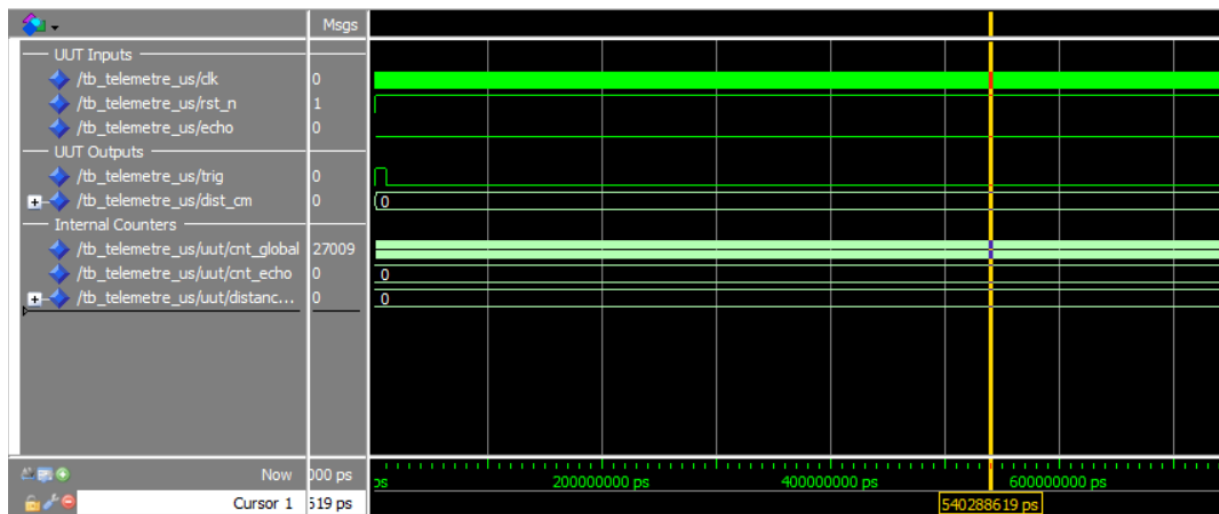


Figure 4 : Test 1 mesure de 0 cm

2. Mesure de 20 cm : On simule un signal Echo d'une durée de 580  $\mu$ s ( $10 \times 58 \mu$ s). On vérifie que dist\_cm indique bien 19.

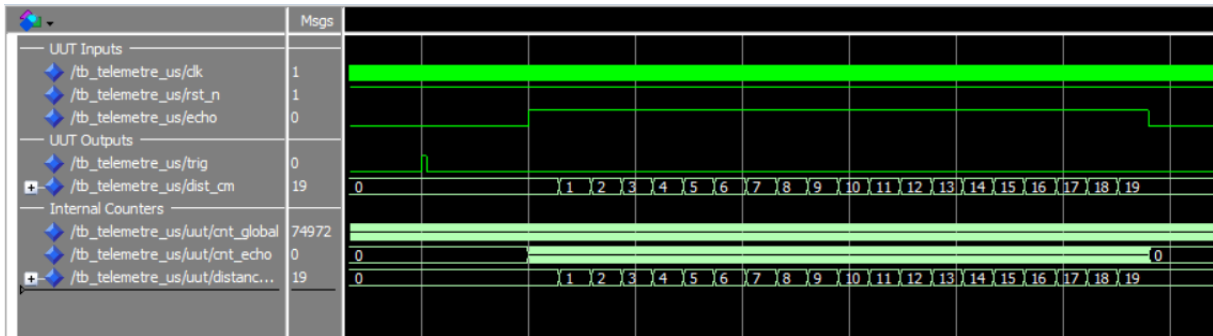


Figure 5 : Test 2 mesure 20cm

- Mesure de 400 cm : On simule un signal Echo d'une durée de 2.9 ms ( $20 \times 58 \mu s$ ). On vérifie la mise à jour de la sortie.

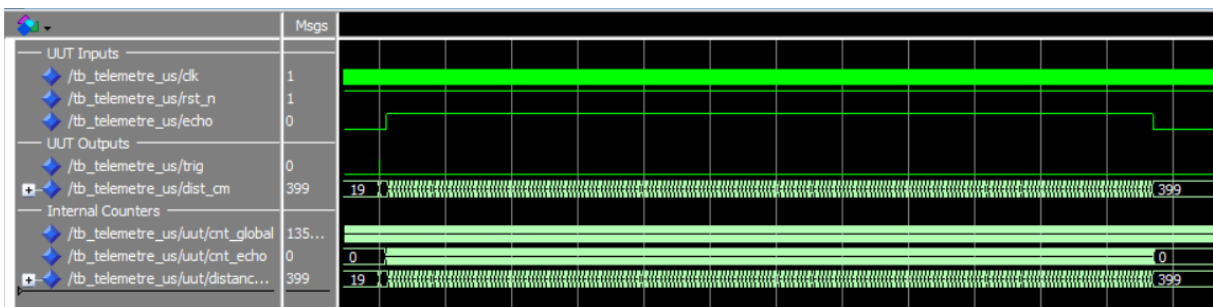


Figure 6 : Test 3 mesure 4M

Ces mesures ont été réalisées avant la création de la variable `dist_cm_reg`. C'est pour cela que `dist_cm` et incrémenter au fur et à mesure sur ces visuelles. Cette imprécision a été corrigée par la suite.

## II. Test télémètres sur la carte

Intégration du module VHDL utilisant l'IP télémètre. Une fois la bibliothèque télémètre ajoutée au projet, nous intégrons notre module VHDL qui configure les broches (pins) et exploite les fonctions de la bibliothèque. Le module a été testé avec succès sur la carte DE10-Lite, avec la configuration matérielle suivante :

Clock	50 MHz (Internal)
Reset	KEY0
Trigger	GPIO [1] (PIN_W10)
Echo	GPIO [3] (PIN_W9)
Sortie	LEDR [9..0] affichage de la distance en binaire

Les LEDs s'allument proportionnellement à la distance de l'obstacle devant le capteur, validant ainsi la partie opérative standalone.

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

entity DE10_Lite_Test_UL is
    port (
        MAX10_CLK1_50 : in  std_logic;           -- PIN_P11
        KEY             : in  std_logic_vector(1 downto 0); -- PIN_B8 (KEY0)
        LEDR            : out std_logic_vector(9 downto 0); -- LEDR[9..0]
        GPIO            : inout std_logic_vector(35 downto 0) -- Headers GPIO (JP1)
    );
end entity;

architecture rtl of DE10_Lite_Test_UL is

    component telemetre_us is
        port (
            clk      : in  std_logic;
            rst_n    : in  std_logic;
            echo     : in  std_logic;
            trig     : out std_logic;
            dist_cm  : out std_logic_vector(9 downto 0)
        );
    end component;

begin

    -- Instanciation de l'IP Telemetre
    -- Connexions selon le tableau du sujet :
    -- Rst_n    -> KEY0
    -- CLK      -> MAX10_CLK1_50
    -- Trig     -> GPIO[1] (PIN_W10)
    -- Echo     -> GPIO[3] (PIN_W9)
    -- Dist_cm  -> LEDR[9..0]

    u0 : telemetre_us
        port map (
            clk      => MAX10_CLK1_50,
            rst_n    => KEY(0),
            echo     => GPIO(3),
            trig     => GPIO(1),
            dist_cm  => LEDR
        );

end architecture;
```

Figure 7 : Code TOP level télémètre

On réalise une démonstration, pour cela on utilise le détecteur pauser à 20 cm d'un obstacle.

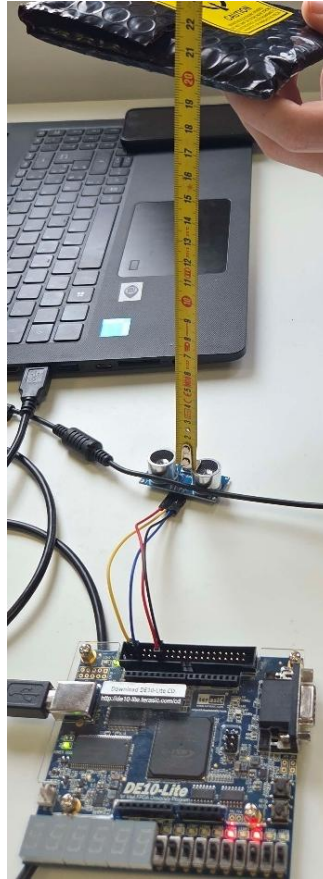


Figure 8 : Resultat Télémètre

On obtient un binaire 0001 0100 si on le convertie en décimale on a  $16 + 4 = 20$  cm comme on peut le voir sur notre mètre.

### III. Intégration de l'IP Avalon Télémètre dans Platform Designer

Ce fichier joue wrap l'IP du télémètre pour ajouter ça connexion au bus Avalon. Il sert d'interface entre le processeur Nios II et le module matériel `telemetre_us`.

Le composant interne effectue toute la mesure (génération de l'impulsion trig, chronométrage du retour echo, conversion en centimètres), tandis que le wrapper transforme cette distance en une donnée lisible par Avalon. Ainsi, dès que le processeur effectue une lecture (`chipselect = 1` et `read_n = 0`), le module renvoie la valeur mesurée dans `readdata`, en plaçant les 10 bits de distance dans les bits de poids faible.

En parallèle, la distance est également envoyée vers `dist_export` pour un affichage la valeur avec les LEDs. Ce composant encapsule donc la logique du télémètre et l'expose proprement au système Nios II, permettant au logiciel embarqué de lire la distance comme un simple registre mémoire.

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

entity telemetre_us_avalon is
    port (
        clk      : in  std_logic;
        rst_n    : in  std_logic;

        -- Interface Avalon-MM Slave
        chipselect : in  std_logic;
        read_n     : in  std_logic;
        readdata   : out std_logic_vector(31 downto 0);

        -- Interface Conduit (Vers l'extérieur)
        trig      : out std_logic;
        echo      : in  std_logic;
        dist_export : out std_logic_vector(9 downto 0) -- Pour afficher sur les LEDs
    );
end entity;

architecture rtl of telemetre_us_avalon is

    component telemetre_us is
        port (
            clk      : in  std_logic;
            rst_n    : in  std_logic;
            echo     : in  std_logic;
            trig     : out std_logic;
            dist_cm  : out std_logic_vector(9 downto 0)
        );
    end component;

    signal dist_cm_sig : std_logic_vector(9 downto 0);

begin

    -- Instanciation du coeur du télémètre
    u0 : telemetre_us
        port map (
            clk      => clk,
            rst_n    => rst_n,
            echo     => echo,
            trig     => trig,
            dist_cm  => dist_cm_sig
        );

    -- Connexion de la sortie Conduit (pour les LEDs)
    dist_export <= dist_cm_sig;

    -- Gestion de la lecture Avalon
    -- L'adresse n'est pas nécessaire car il n'y a qu'un seul registre à lire
    process(clk)
    begin
        if rising_edge(clk) then
            if chipselect = '1' and read_n = '0' then
                readdata <= (others => '0'); -- Mise à zéro des bits de poids fort
                readdata(9 downto 0) <= dist_cm_sig; -- Assignation de la distance
            end if;
        end if;
    end process;

end architecture;
```

Figure 9 : Code TOP level télémètre Avalon

Pareille que pour la simulation précédente nous avons tester 3 valeur différente

1. La mesure de 0 cm. Ce qui et inversent de constater pour cette mesure, vue que c'est la premier le bus readdata et a x vue que rien à été écris dedans il et dans

## Mini-Projet Radar 2D

un état neutre en attendant une valeur, nous avons belle et bien push la valeur 0 sur le bus mais que plus tard.

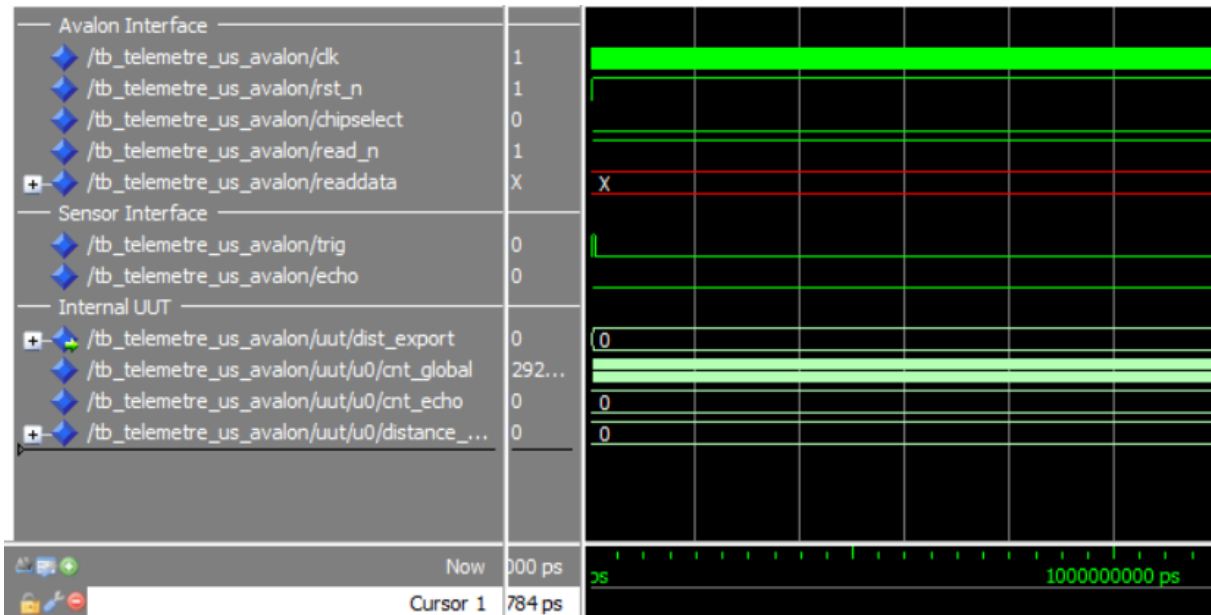


Figure 10 : Test 1 mesure 0 cm

- Mesure de 20 centimètres : on voit la mesure de 0 cm passer sur le bus readdata au début de la nouvelle mesure. Après 30 ns la valeur mesurée et push, on constat que la valeur et 19 et non 20. Cela et expliquer par une tout petite imprécision sur notre code le compteur comptait de 0 à 2900 inclus, ce qui fait 2901 cycles par centimètre. Et vue qu'en simulation nous avons rentre exactement 20 cm il nous manque 20 cycles pour avoir la valeur 20.

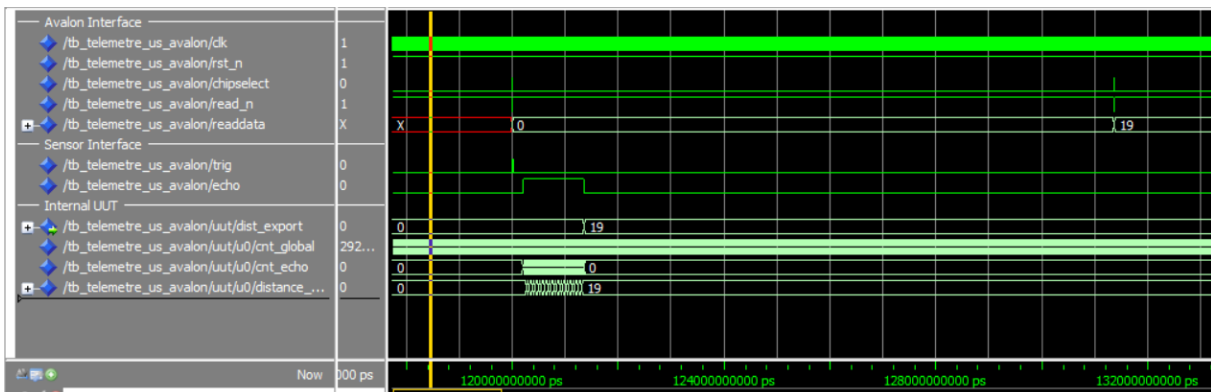


Figure 11 : Test 2 mesure 20 cm

- Pareille que pur la mesure 2 nous avons la petite imprécision ce qui nous donne 399cm aux lieux de 400. Et la valeur et push sur le bus Avalon (readdata) après un deley.

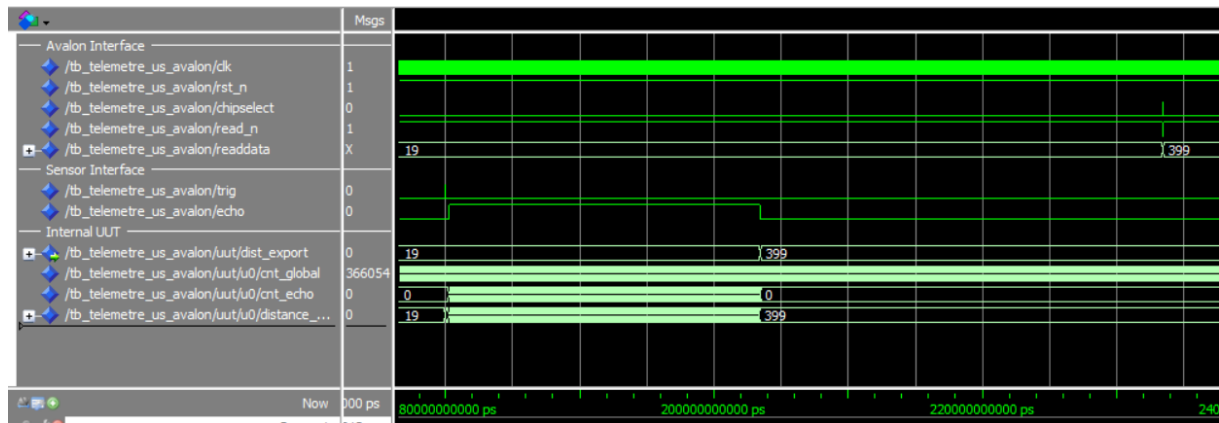


Figure 12 : Test 3 mesure 4 m

## IV. Programmation logicielle et test de l'IP Télémètre

Ce programme C réalise un test simple du télémètre ultrason connecté au système Nios II via le bus Avalon. Après un message d'initialisation, la boucle principale lit en continu la distance mesurée grâce à la macro IORD (TELEMETRE\_US\_AVALON\_0\_BASE, 0), qui permet de récupérer la valeur stockée dans le registre du module matériel telemetre\_us\_avalon. Comme alt\_printf ne supporte pas l'affichage décimal avec %d, la distance est affichée en hexadécimal, ce qui suffit pour vérifier que la valeur varie correctement en fonction de la position des objets devant le capteur. Une pause de 100 ms est ajoutée entre chaque lecture pour éviter de saturer la sortie série. Ce code constitue donc la première étape de validation fonctionnelle du télémètre avant l'ajout ultérieur de l'affichage 7 segments.

```
#include "sys/alt_stdio.h"
#include "system.h"
#include "io.h"
#include "unistd.h"

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

    while(1) {
        // Lecture de la distance avec la bonne macro
        int dist = IORD(TELEMETRE_US_AVALON_0_BASE, 0);

        alt_printf("Distance Hex: %x \n", dist);

        // Pause 100ms
        usleep(100000);
    }

    return 0;
}
```

Figure 13 : Code C Télémètre

Les tests réalisés montrent que le télémètre répond correctement aux variations de distance : lorsque la main ou un objet s'approche du capteur, la valeur affichée change instantanément, ce qui confirme que la liaison Avalon et le module matériel

fonctionnent comme prévu. Sur les photos, on voit clairement les mesures effectuées avec une règle ou avec une main placée à différentes hauteurs, et les valeurs renvoyées par le programme évoluent en conséquence. Le système détecte aussi bien les courtes distances (quelques centimètres) que les distances plus grandes, montrant que le trig/echo fonctionne correctement et que la conversion en centimètres effectuée dans l'IP est fiable. Ces résultats valident donc pleinement la communication entre le Nios II et le télémètre, ainsi que la justesse des mesures avant l'intégration de l'affichage HEX.

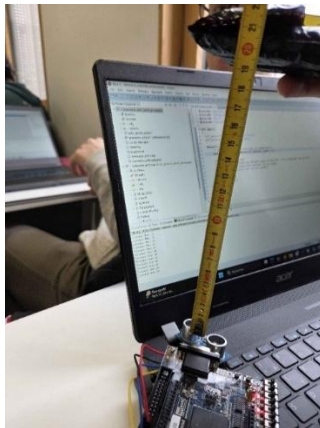


Figure 14 : Résultat affichage valeur télémètre sur la terminale

A la suite nous avons ajouter la valeur ce coup-ci en décimale sur l'afficher 7 segments. Nous avons malheureusement oublié cette étape nous l'avons fait dans la partie 3 donc le code sera présenté plus tard.

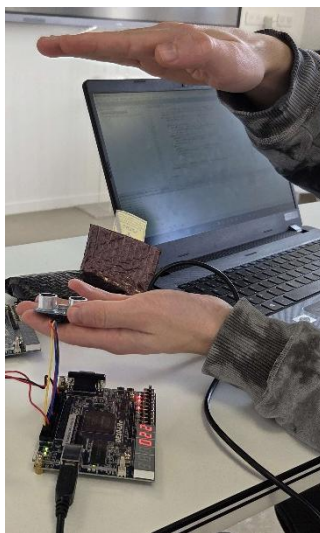


Figure 15 : Affichage 7seg

## 4. Conception de l'IP Servomoteur

### I. Simulation IP

#### Simulation de notre IP servomoteur

L'IP a été conçue de manière comportementale. Ce choix a été privilégié car il permet de décrire précisément les relations temporelles (période de 20ms et largeur d'impulsion de 1 à 2ms) de façon lisible et efficace sans multiplier les composants structurels.

L'architecture repose sur trois piliers logiques au sein d'un processus synchrone cadencé à 50 MHz :

- Le Compteur de Période, un simple compteur d'entiers allant de 0 à 1 000 000. Puisque l'horloge système est de 50 MHz (période de 20 ns), ce compteur définit la fréquence de rafraîchissement du servomoteur :  $1\,000\,000 \times 20\text{ ns} = 20\text{ ms}$  (50 Hz).
- Calcul du Rapport Cyclique, la position reçue sur 8 bits (0 à 255) est convertie en une durée d'impulsion. Formule appliquée :  $Cycles = 50\,000 + (Position \times 196)$  avec un cycle de 50 000 correspondant à 1 ms (angle 0°). L'ajout de  $Position \times 196$  permet d'atteindre progressivement 2 ms (100 000 cycles) lorsque la position est au maximum (255).
- Génération du Signal PWM, une simple comparaison, tant que le compteur est inférieur à la valeur duty\_cycle, le signal commande est à '1', sinon il passe à '0'

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

entity servomoteur is
    port (
        clk       : in  std_logic;
        reset_n   : in  std_logic;
        position   : in  std_logic_vector(7 downto 0);
        commande  : out std_logic
    );
end entity;

architecture bhv of servomoteur is
    -- 20 ms @ 50 MHz = 1 000 000 cycles
    constant CNT_PERIOD_MAX : integer := 1000000;

    -- Limites PWM pour atteindre 180° (0.6 ms à 2.4 ms)
    -- Base 0.6 ms = 30 000 cycles
    -- Max 2.4 ms = 120 000 cycles
    -- Pas = (120000 - 30000) / 255 = 353
    constant DUTY_MIN  : integer := 30000;
    constant DUTY_MAX  : integer := 120000;
    constant DUTY_STEP : integer := 353;

    signal counter      : integer range 0 to CNT_PERIOD_MAX := 0;
    signal duty_cycle   : integer range 0 to CNT_PERIOD_MAX := 75000; -- 1.5 ms par défaut (90°)
begin
    -----
    -- Calcul du duty cycle à partir de la position
```

```
-----  
process(clk, reset_n)  
    variable temp_duty : integer;  
begin  
    if reset_n = '0' then  
        duty_cycle <= 75000;  
    elsif rising_edge(clk) then  
        temp_duty := DUTY_MIN + (to_integer(unsigned(position)) * DUTY_STEP);  
  
        -- Saturation de sécurité  
        if temp_duty < DUTY_MIN then  
            duty_cycle <= DUTY_MIN;  
        elsif temp_duty > DUTY_MAX then  
            duty_cycle <= DUTY_MAX;  
        else  
            duty_cycle <= temp_duty;  
        end if;  
    end if;  
end process;  
  
-----  
-- Génération du PWM  
-----  
process(clk, reset_n)  
begin  
    if reset_n = '0' then  
        counter <= 0;  
        commande <= '0';  
    elsif rising_edge(clk) then  
        -- Compteur de période (20 ms)  
        if counter < CNT_PERIOD_MAX then  
            counter <= counter + 1;  
        else  
            counter <= 0;  
        end if;  
  
        -- Signal PWM  
        if counter < duty_cycle then  
            commande <= '1';  
        else  
            commande <= '0';  
        end if;  
    end if;  
end process;  
  
end architectu
```

Figure 16 : Code de servomoteur

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all;  
  
entity tb_servomoteur is  
end entity;  
  
architecture test of tb_servomoteur is  
    signal clk      : std_logic := '0';  
    signal reset_n  : std_logic := '0';  
    signal position : std_logic_vector(7 downto 0) := (others => '0');  
    signal commande : std_logic;  
  
    constant CLK_PERIOD : time := 20 ns; -- 50 MHz  
begin  
    -- Instanciation de l'UUT (Unit Under Test)  
    uut: entity work.servomoteur  
        port map (  
            clk      => clk,  
            reset_n  => reset_n,  
            position => position,  
            commande => commande  
        );  
  
    -- Génération de l'horloge  
    clk_process : process
```

```
begin
    clk <= '0';
    wait for CLK_PERIOD/2;
    clk <= '1';
    wait for CLK_PERIOD/2;
end process;

-- Stimuli
stim_proc: process
begin
    -- Initialisation
    reset_n <= '0';
    position <= "00000000";
    wait for 100 ns;
    reset_n <= '1';

    -- Test Position 0 (doit donner une impulsion de 1ms)
    position <= "00000000";
    report "Test Position 0 (0 degrees)";
    wait for 21 ms; -- Attendre une période PWM complète (20ms + marge)

    -- Test Position 127 (approx 90 degrees, doit donner une impulsion de ~1.5ms)
    position <= "01111111";
    report "Test Position 127 (approx 90 degrees)";
    wait for 20 ms;

    -- Test Position 255 (180 degrees, doit donner une impulsion de 2ms)
    position <= "11111111";
    report "Test Position 255 (180 degrees)";
    wait for 20 ms;

    report "Fin de la simulation";
    wait;
end process;
end architecture;
```

Figure 17 : Code de tb servomoteur

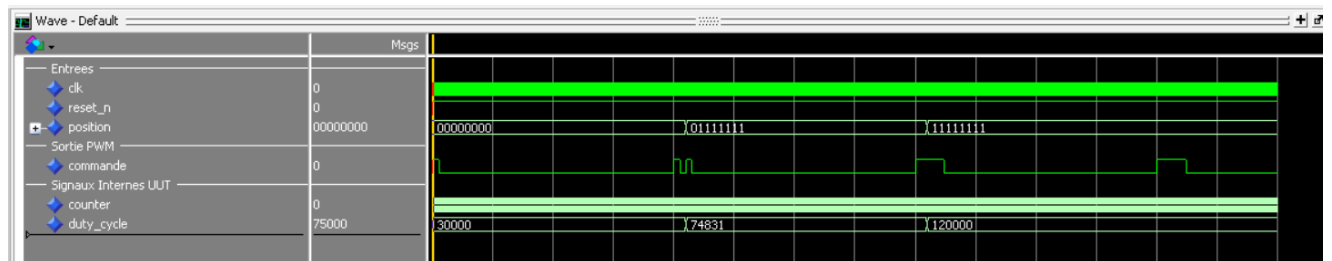


Figure 18 : Simulation commande pwm

L'IP s'articule autour d'un processus synchrone à 50 MHz (20 ns) et de trois briques logiques proposer précédemment. La figure 19 de simulation confirme le comportement attendu pour position=0 on observe une impulsion d'environ 0,6 ms (compteur  $\approx 30\,000$ ), pour position=127 une impulsion proche de 1,5 ms (compteur  $\approx 74\,831$ ), et pour position=255 une impulsion saturée à 2,4 ms (compteur 120 000). Ces impulsions se répètent toutes 20 ms, validant la linéarité du mappage position largeur d'impulsion et la stabilité de la période PWM.

La figure 20 définit un top-level minimal permettant de tester le servomoteur sur la carte DE10-Lite en reliant directement les interrupteurs, les LED et les broches GPIO au module servomoteur. Les afficheurs 7-segments sont entièrement désactivés pour simplifier le test, tandis qu'un diviseur d'horloge interne est utilisé pour faire clignoter la LED 9 et indiquer que le système fonctionne bien. Les interrupteurs SW (7..0) sont connectés aux LED correspondantes, servant de retour visuel et permettant de sélectionner la position du servomoteur en temps réel. Le module servomoteur est instancié avec l'horloge 50 MHz de la carte, un reset maintenu à '1' et la valeur de position fournie par les interrupteurs. Le signal PWM généré est reçu dans `pwm_signal`, visualisé faiblement sur la LED 8, puis envoyé vers la broche GPIO 0 pour piloter physiquement le servomoteur. Ce top-level agit donc comme une interface simple et directe entre les contrôles utilisateur de la carte et le module PWM, idéal pour vérifier que le servomoteur réagit correctement aux valeurs choisies via les switches.

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

entity DE10_Lite_Servo_Test is
    port (
        CLOCK_50 : in  std_logic;
        KEY      : in  std_logic_vector(1 downto 0);
        SW       : in  std_logic_vector(9 downto 0);
        LED      : out std_logic_vector(9 downto 0);
        GPIO     : out std_logic_vector(35 downto 0);
        HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : out std_logic_vector(7 downto 0)
    );
end entity;

architecture rtl of DE10_Lite_Servo_Test is
    signal clk_divider : unsigned(24 downto 0) := (others => '0');
    signal pwm_signal  : std_logic;
begin
    -- 1. Éteindre tout ce qui est inutile
    HEX0 <= (others => '1'); HEX1 <= (others => '1');
    HEX2 <= (others => '1'); HEX3 <= (others => '1');
    HEX4 <= (others => '1'); HEX5 <= (others => '1');

    -- 2. Témoin de fonctionnement : LED(9) clignote
    process(CLOCK_50)
    begin
        if rising_edge(CLOCK_50) then
            clk_divider <= clk_divider + 1;
        end if;
    end process;
    LED(9) <= clk_divider(24);

    -- 3. Feedback visuel : LED(7..0) montrent SW(7..0)
    LED(7 downto 0) <= SW(7 downto 0);

    -- 4. Témoin du signal Moteur : La LED(8) doit être allumée (faiblement)
    LED(8) <= pwm_signal;

    -- 5. Contrôleur de servomoteur
    i_servo : entity work.servomoteur
    port map (
        clk       => CLOCK_50,
        reset_n   => '1',           -- On force le reset à '1' (désactivé) pour le test
        position  => SW(7 downto 0),
        commande  => pwm_signal     -- On passe par un signal interne
    );
```

```
-- Envoi du signal vers la sortie physique PIN_V10
GPIO(0) <= pwm_signal;

end architecture;
```

Figure 19 : Top level servomotor

Par la suite, nous avons pu effectuer des tests, et tout a fonctionné correctement, même si certaines difficultés de calibrage ont pu être rencontrées. Nous avons ensuite étendu l'IP Servomoteur vers une version compatible avec le bus Avalon.

### Simulation de IP Avalon du servomoteur

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

entity tb_IP_Servo_Avalon is
end entity;

architecture test of tb_IP_Servo_Avalon is
    -- Signaux
    signal clk          : std_logic := '0';
    signal reset_n      : std_logic := '0';
    signal chipselect   : std_logic := '0';
    signal write_n      : std_logic := '1';
    signal writedata    : std_logic_vector(31 downto 0) := (others => '0');
    signal commande     : std_logic;

    constant CLK_PERIOD : time := 20 ns; -- 50 MHz
begin

    -- Instanciation de l'IP Avalon
    uut: entity work.IP_Servo_Avalon
        port map (
            clk          => clk,
            reset_n      => reset_n,
            chipselect   => chipselect,
            write_n      => write_n,
            writedata    => writedata,
            commande     => commande
        );

    -- Horloge 50 MHz
    clk_process : process
    begin
        clk <= '0'; wait for CLK_PERIOD/2;
        clk <= '1'; wait for CLK_PERIOD/2;
    end process;

    -- Stimuli Avalon
    stim_proc: process
    begin
        -- 1. Reset
        reset_n <= '0';
        wait for 100 ns;
        reset_n <= '1';
        wait for 100 ns;

        -- 2. Écriture Position 0°
        report "Ecriture Avalon : Position 0°";
        chipselect <= '1';
        write_n <= '0';
        writedata <= "00000000000000000000000000000000";
        wait for CLK_PERIOD;
        chipselect <= '0';
    end process;
end;
```

```

write_n    <= '1';

wait for 21 ms;

-- 3. Écriture Position 90° (valeur 127)
report "Ecriture Avalon : Position 90";
chipselect <= '1';
write_n    <= '0';
writedata  <= "0000000000000000000000000000000011111111";
wait for CLK_PERIOD;
chipselect <= '0';
write_n    <= '1';

wait for 21 ms;

-- 4. Écriture Position 180° (valeur 255)
report "Ecriture Avalon : Position 180";
chipselect <= '1';
write_n    <= '0';
writedata  <= "0000000000000000000000000000000011111111";
wait for CLK_PERIOD;
chipselect <= '0';
write_n    <= '1';
wait for 21 ms;

report "Fin de simulation Avalon";
wait;
end process;

end architecture;
```

Figure 20 : TestBench servomoteur avalon

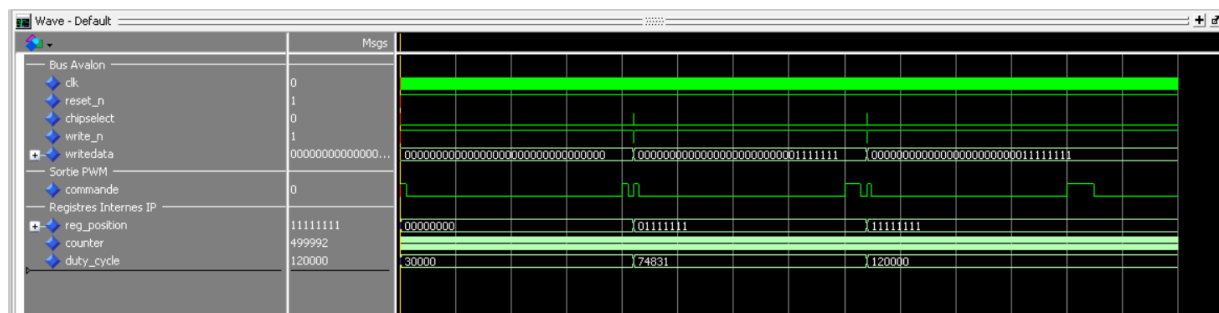


Figure 21 : Simulation PWM Avalon

L'extension de l'IP Servomoteur vers une version compatible Avalon-MM a nécessité la mise en place d'un testbench dédié afin de valider l'interface bus et son interaction avec la logique PWM interne. Le testbench reproduit un environnement minimal Avalon, génération d'une horloge 50 MHz, gestion du signal chipselect, commande d'écriture `write_n` ainsi que du bus `writedata`. Trois écritures successives sont réalisées pour simuler l'envoi d'angles de 0°, 90° (127) et 180° (255) via le bus. À chaque transaction, l'IP lit la valeur écrite dans son registre interne `reg_position`, met à jour le `duty_cycle` en fonction de cette nouvelle position, puis génère le signal PWM conformément aux valeurs de timing spécifiées (entre 30 000 cycles pour 0,6 ms et 120 000 cycles pour 2,4 ms).

La simulation confirme le bon fonctionnement de l'ensemble, les écritures Avalon sont correctement prises en compte, le registre interne reflète fidèlement les valeurs reçues, et la largeur d'impulsion du PWM évolue exactement comme attendu. On observe ainsi une impulsion minimale ( $\approx 30\,000$  cycles) lors de l'écriture de  $0^\circ$ , une impulsion intermédiaire ( $\approx 74\,831$  cycles) pour  $90^\circ$ , puis une impulsion maximale

(120 000 cycles) pour 180°. La cohérence entre le bus Avalon, la mise à jour du registre, et la génération PWM démontre que l'IP est pleinement fonctionnelle.

Le code VHDL de la figure 22 constitue le *top-level* du projet et sert d'interface entre la carte DE10-Lite. Il instancie le composant nios2\_system et connecte chaque signal physique de la carte à ses équivalents internes. Le télémètre ultrason est raccordé via les broches GPIO pour le signal trig et echo, tandis que la distance mesurée est affichée directement sur les LED rouges grâce au port telemetre\_us\_avalon\_dist. Le servomoteur est commandé par l'IP PWM intégrée dans le système Nios II, reliée ici à GPIO(0). Les afficheurs HEX0 à HEX5 reçoivent chacun les 8 bits correspondants depuis les bus Avalon hex3\_0 et hex5\_4. L'ensemble forme une architecture cohérente permettant au programme C, exécuté par le processeur Nios, de lire le télémètre, calculer la position du servo et piloter ses périphériques en temps réel.

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

entity DE10_Lite_Test_UL is
    port (
        MAX10_CLK1_50 : in  std_logic;           -- PIN_P11
        KEY             : in  std_logic_vector(1 downto 0); -- PIN_B8 (KEY0)
        LEDR            : out std_logic_vector(9 downto 0); -- LEDR[9..0]
        GPIO            : inout std_logic_vector(35 downto 0); -- Headers GPIO (JP1)
        SW              : in  std_logic_vector(9 downto 0); -- SW[9..0]
        HEX0, HEX1, HEX2, HEX3, HEX4, HEX5 : out std_logic_vector(7 downto 0) -- HEX[5..0]
    );
end entity;

architecture rtl of DE10_Lite_Test_UL is

    component nios2_system is
        port (
            clk_clk           : in  std_logic           := 'X';
            hex3_0_external_connection_export : out std_logic_vector(31 downto 0);
            hex5_4_external_connection_export : out std_logic_vector(15 downto 0);
            key_external_connection_export    : in  std_logic_vector(1 downto 0) := (others => 'X');
            led_external_connection_export    : out std_logic_vector(9 downto 0);
            pwmled_writeresponsevalid_n      : out std_logic;
            writeresponsevalid_n             : in  std_logic           := 'X';
            reset_reset_n                    : in  std_logic           := 'X';
            sw_external_connection_export     : in  std_logic_vector(9 downto 0) := (others => 'X');
            telemetre_us_avalon_trig         : out std_logic;
            telemetre_us_avalon_echo         : in  std_logic           := 'X';
            telemetre_us_avalon_dist         : out std_logic_vector(9 downto 0)
        );
    end component nios2_system;

begin

    u0 : component nios2_system
        port map (
            clk_clk           => MAX10_CLK1_50,
            reset_reset_n     => KEY(0),
            -- Télémètre
```

```
telemetre_us_avalon_trig      => GPIO(1),
telemetre_us_avalon_echo     => GPIO(3),
telemetre_us_avalon_dist     => LEDR, -- Affichage direct de la distance hardware

-- Servomoteur (via PWM0)
pwmled_writeresponsevalid_n  => GPIO(0),

-- Périphériques standards
sw_external_connection_export => SW,
key_external_connection_export => KEY,
led_external_connection_export => open, -- Connecté mais non utilisé (conflit avec LEDR
distance)

-- Afficheurs 7 segments (mappage partiel vers les vecteur 32/16 bits)
hex3_0_external_connection_export(7 downto 0) => HEX0,
hex3_0_external_connection_export(15 downto 8) => HEX1,
hex3_0_external_connection_export(23 downto 16) => HEX2,
hex3_0_external_connection_export(31 downto 24) => HEX3,
hex5_4_external_connection_export(7 downto 0) => HEX4,
hex5_4_external_connection_export(15 downto 8) => HEX5
);

end architecture;
```

Figure 22 : Top level Télémètre et servomoteur

## II. Programmation logicielle et test de l'IP Servomoteur

Le programme de la figure 24 effectue un test simple du servomoteur en envoyant trois valeurs fixes sur le bus Avalon 0, 127 et 255, correspondant respectivement aux positions 0°, 90° et 180°. À chaque écriture, le code utilise la macro IOWR\_32DIRECT pour transmettre directement la valeur dans le registre de commande du module PWM situé à l'adresse PWM0\_BASE. Entre chaque changement de position, le programme réalise une pause d'une seconde afin de laisser le servomoteur atteindre sa nouvelle orientation. Ainsi, le code permet de valider le bon fonctionnement de la communication Avalon et de vérifier que le servomoteur réagit correctement aux trois positions principales.

```
#include <stdio.h>
#include <unistd.h>
#include "system.h"
#include "io.h"

/*
 * L'adresse IP_SERVO_AVALON_BASE est définie dans votre system.h
 * après avoir généré le BSP.
 */

int main() {
    printf("Test du Servomoteur via Avalon\n");
    printf("-----\n");

    while (1) {
        // Position 0 (0°)
        printf("Position: 0°\n");

        // Nouvelle ligne corrigée
        IOWR_32DIRECT(PWM0_BASE, 0, 0);
        usleep(1000000); // 1 seconde

        // Position 127 (90°)
        printf("Position: 90°\n");
        IOWR_32DIRECT(PWM0_BASE, 0, 127);
        usleep(1000000);

        // Position 255 (180°)
        printf("Position: 180°\n");
```

```
IOWR_32DIRECT(PWM0_BASE, 0, 255);  
usleep(1000000);  
}  
  
return 0;  
}
```

Figure 23 : Code C servomoteur

```
Servo 180 deg | Dist: 191 cm  
Servo 0 deg | Dist: 18 cm  
Servo 90 deg | Dist: 464 cm  
Servo 0 deg | Dist: 500 cm  
Servo 90 deg | Dist: 329 cm  
Servo 180 deg | Dist: 500 cm  
Servo 0 deg | Dist: 195 cm  
Servo 90 deg | Dist: 500 cm  
Servo 180 deg | Dist: 61 cm  
Servo 0 deg | Dist: 500 cm  
Servo 90 deg | Dist: 500 cm  
Servo 180 deg | Dist: 375 cm
```

Figure 24 : Résultat code C servomoteur

Les résultats affichés montrent que le servomoteur se positionne successivement à 0°, 90° puis 180°, et que pour chaque angle, la distance mesurée par le télémètre varie en fonction de l'environnement. Les valeurs relevées vont de quelques dizaines à plusieurs centaines de centimètres, ce qui confirme que le servomoteur pivote correctement et que le capteur ultrason mesure une distance à chaque arrêt. Les distances élevées à 500 cm indiquent que le capteur ne détecte aucun obstacle dans cette direction (retour maximal), tandis que les valeurs intermédiaires traduisent la présence d'objets à des positions spécifiques. L'ensemble valide que la communication avec le PWM et la lecture du télémètre fonctionnent comme prévu.

### 5. Affichage des obstacles

Cette partie demande de fusionner le télémètre avec le serveur moteur pour réaliser un système de « radar 2D ». Le servomoteur balaie automatiquement l'espace de 0° à 180°, puis repart en sens inverse, avec des pas de 5° espacés de 100 ms. Pour chaque position angulaire, le code calcule la largeur d'impulsion PWM nécessaire afin d'orienter le servo, puis lit la distance mesurée par le capteur ultrason. Cette distance est affichée simultanément sur la console sous forme de barre horizontale et sur l'afficheur 7-segments. Le code gère également la saturation des angles, l'interpolation de la durée d'impulsion, et contrôle la fluidité du balayage.

```
/*
 * Radar 2D - Mini Projet ESIEA
 * Balayage Servomoteur 0-180 + Mesure Telemetre
 */

#include <stdio.h>
#include <unistd.h>
#include "system.h"
#include "io.h"
#include "altera_avalon_pio_regs.h"

// --- CONSTANTES ---
#define PWM_PERIOD      1000000 // 20ms @ 50MHz
#define PULSE_MIN       25000   // 0.5ms (Min - 0 deg)
#define PULSE_MAX       125000  // 2.5ms (Max - 180 deg)
#define DELAY_STEP_US   100000  // 100ms par pas
#define PAS_ANGLE        5       // pas d'angle

const unsigned char table_7seg[] = {
    0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82, 0xF8, 0x80, 0x90
};

int angle_to_cycles(int angle) {
    if (angle < 0) angle = 0;
    if (angle > 180) angle = 180;

    return 25000 + (angle * 555);
}

void update_display(int val) {
    if (val > 9999) val = 9999;
    int u = val % 10;
    int d = (val / 10) % 10;
    int c = (val / 100) % 10;

    unsigned int hex_val = (0xFF << 24) | (table_7seg[c] << 16) | (table_7seg[d] << 8) |
        table_7seg[u];

    IOWR_ALTERA_AVALON_PIO_DATA(HEX3_0_BASE, hex_val);
}

int main() {
    printf("--- DEMARRAGE RADAR ---\n");
    printf("Si le servo force en butee : Reduire PULSE_MAX ou Augmenter PULSE_MIN\n");

    IOWR_32DIRECT(PWM0_BASE, 4, PWM_PERIOD);

    int angle = 0;
    int sens = 1;

    while(1) {
        int cycles = angle_to_cycles(angle);
        IOWR_32DIRECT(PWM0_BASE, 0, cycles);
```

## Mini-Projet Radar 2D

```
usleep(DELAY_STEP_US);

int dist_cm = IORD_32DIRECT(TELEMETRE_US_AVALON_0_BASE, 0);
// printf("%d deg -> %d cm \n", angle, dist_cm);

// Affichage "Graphique"
printf("%3d deg | %3d cm : ", angle, dist_cm);

// Limite d'affichage pour ne pas saturer la console (max 80 chars approx)
int affichage_dist = (dist_cm > 80) ? 80 : dist_cm;

for(int i=0; i<affichage_dist; i++) {
    printf("|");
}
if (dist_cm > 80) printf("..."); // Indiquer si ça continue
printf("\n");
update_display(dist_cm);

angle += (PAS_ANGLE * sens);
if (angle >= 180) {
    angle = 180;
    sens = -1;
    printf("--- Retour ---\n");
} else if (angle <= 0) {
    angle = 0;
    sens = 1;
    printf("--- Aller ---\n");
    sens = 1;
    printf("--- Aller (0 -> 180) ---\n");
    sens = 1;
    printf("--- Aller (0 -> 180) ---\n");
}
}
return 0;
}
```

Figure 25 : Code C radar 2D

```
10 deg | 50 cm : |||||||||||||||||||||||||||||||||||||||
15 deg | 27 cm : ||||||||||||||||||||||||||||
20 deg | 27 cm : ||||||||||||||||||||||||||||
25 deg | 23 cm : ||||||||||||||||||||||||
30 deg | 21 cm : ||||||||||||||||||||
35 deg | 22 cm : |||||||||||||||||||
40 deg | 22 cm : |||||||||||||||||||
45 deg | 21 cm : |||||||||||||||||||
50 deg | 21 cm : |||||||||||||||||||
55 deg | 21 cm : |||||||||||||||||||
60 deg | 21 cm : |||||||||||||||||||
65 deg | 23 cm : |||||||||||||||||||
70 deg | 22 cm : |||||||||||||||||||
75 deg | 23 cm : |||||||||||||||||||
80 deg | 23 cm : |||||||||||||||||||
85 deg | 22 cm : |||||||||||||||||||
90 deg | 22 cm : |||||||||||||||||||
95 deg | 22 cm : |||||||||||||||||||
100 deg | 21 cm : |||||||||||||||||||
```

Figure 26 : Résultat radar 2D

Les résultats montrent l'évolution de la distance mesurée en fonction de l'angle du servomoteur pour chaque position, la console affiche une ligne du type « XX deg | YY cm : ||||| » où la longueur de la barre reflète directement la distance détectée. Dans la figure 22, le système détecte des distances proches (entre 21 cm et 27 cm) ce qui produit des barres de taille similaire, confirmant le bon fonctionnement de la lecture du télémètre. Le balayage angulaire fonctionne, et l'affichage change en fonction des valeurs renvoyées par le capteur, prouvant que la communication PWM, la mesure ultrason et la visualisation fonctionnent parfaitement ensemble.

### 6. Conclusion

Ce projet nous a permis d'aborder de manière concrète et complète le processus de conception d'un système embarqué complexe, depuis la description RTL jusqu'à l'intégration logicielle. La simulation des deux IP a validé la précision temporelle de nos conceptions, tandis que leur implémentation sur la carte DE10-Lite a démontré le bon fonctionnement du système en conditions réelles. La communication Avalon et le processeur Nios II ont joué un rôle central en permettant une interaction fluide entre les modules matériels et la couche logicielle, rendant possible un contrôle précis du servomoteur et une lecture fiable du télémètre.

En aboutissant à un radar fonctionnel capable de balayer un environnement et d'afficher en continu les distances mesurées, nous avons validé l'ensemble de la chaîne conception VHDL, simulation, intégration matérielle, routage des signaux, génération du bitstream, développement en C et tests en situation réelle. Ce travail nous a ainsi offert une expérience pratique précieuse, couvrant les aspects fondamentaux de l'ingénierie FPGA et des systèmes embarqués. Il constitue une base solide pour d'éventuelles évolutions, comme l'affichage graphique sur écran VGA, la reconstruction d'une carte 2D complète, ou l'intégration d'algorithmes de filtrage et de détection avancés.