Mini-Projet
sur DE0 Nano SoC:
Radar 2D

YANG Liyun
JIN Qianhui

EISE4

Année: 2020-2021
Enseignement: M.Douze

Sommaire

I. Introduction

L'architecture des processeurs est un module d'enseignement de notre deuxième année du cycle ingénieurs en EISE (électronique, informatique et système embarqué). Pendant ce module, nous devons réaliser un mini-projet, Radar 2D. Le but de ce mini-projet est de cartographier une scène (Radar 2D) à l'aide d'un télémètre ultrason et un servomoteur avec un affichage sur des LEDs. Le télémètre ultrason sert à détecter des obstacles et calculer la distance en centimètre. Afin d'atteindre cet objectif, nous devons réaliser les codes en vhdl qui décrit la marche du IP télémètre ultrason et IP servomoteur, puis ajouter des bus avalon pour implémenter l'interface les IPs avec le bus avalon dans la carte NIOS2 à l'aide de Qsys. A la fin, il faut réaliser les codes en c qui permettent de lire la valeur de distance et l'angle tourné par le servomoteur puis l'afficher dans le terminal.

II. Implémentation de l'IP télémètre ultrason HC SR04

A. Principe

Avant de commencer à écrire les codes, nous devons comprendre comment le capteur HC SR04 marche. En effet, pour déclencher la mesure, il faut envoyer un signal *trig* de 10us au capteur, ensuite, ce capteur va envoyer un signal interne pour détecter l'obstacle. S'il y a un obstacle, il va envoyer un signal *echo* qui nous permet de mesurer la durée d'aller-retour entre l'obstacle et le capteur. Sinon, après 60 ms, le capteur va recommencer les mêmes étapes.

B. Réalisation de l'IP

1. Code d'IP sous VHDL

Selon le schéma au dessus, on a les ports d'IP telemetre comme ce figure en dessous:

```
entity Telemetre_us is
port(
  clk: in std_logic; --50MHz
  rst: in std_logic; --active en '0'
  echo: in std_logic;
  trig: out std_logic;
  LEDR: out std_logic_vector(7 downto 0)
);
end Telemetre_us;
```

IP Telemetre_us contient 2 processus : le processus de la génération du signal ***Trig*** et le processus du calcul de la distance entre l'obstacle et le capteur ultrason.

Pour déclencher une mesure, on a besoin d'un signal Trig avec une impulsion "high" (=1) d'au moins 10 us, puis il est resté à '0' pendant 60 ms. Dans ce cas là, on prend un compteur (***signal cpt_trig: integer:= 0;***) pour distinguer d'où le signal doit être "1" et "0". Or l'horloge de FPGA est 20 ns (50 MHz), par conséquent, pendant ***cpt_trig*** est entre 0 et 500 (exclu) (soit 10 us), le signal ***Trig*** est au niveau '1', ci ***cpt_trig*** est entre 500 et 3000000 (soit 60 ms), ***Trig*** est au niveau '0'. Le processus de la génération de Trig est comme la figure en-dessous:

```
process(rst,clk)
  begin
    if rst = '0' then
      s_trig <= '0';
      cpt_trig<= 0;
    elsif rising_edge(clk) then
      cpt_trig <= cpt_trig +1;
      if cpt_trig < 500 then  --quand le signal trig
                              --ne sont pas encore durer de 10us
        s_trig <= '1';
      elsif cpt_trig < 3000000 then --quand l'envoie du signal trig
                                    --n'a pas encore passer la limite:60ms
        s_trig <='0';
      else -- si 60ms
        cpt_trig <= 0;
      end if;
    end if;
end process;
trig <= s_trig;
```

Le capteur ultrason émet une série de 8 impulsions ultra-soniques à 40 kHz, puis il attend le signal réfléchi. Lorsque celui-ci est détecté, il envoie un signal "high" sur la sortie "***Echo***", donc la durée est proportionnelle à la distance mesurée.

Pour cela, on a pris aussi un compteur pour calculer la durée de ***Echo*** (***cpt_echo***). ce compteur s'incrémente si et seulement si le signal "***Echo***" est apparu, c'est-à- dire que ***Echo*** est au niveau '1'. Si ***cpt_echo*** égale à 2920 (2 cm pour aller-retour), alors le signal ***dis*** (la distance entre l'obstacle et le capteur) s'incrémente, ***cpt_echo*** est mis à zéro.

Lorsque le signal ***Echo*** est au niveau '0', ***cpt_echo*** et ***dis*** sont tous mis a zéro. ***LEDR*** maintient la valeur de distance jusqu'à un nouvel ***Echo*** apparaît.

Pour ne pas empêcher le signal ***dis***, on a pris un autre signal ***dis_tmp*** pour récupérer la valeur du signal ***dis*** après son chaque incrémentation, puis ***dis_tmp*** donne la valeur à ***LEDR*** à la fin du processus.

La figure en dessous est le processus du calcul:

```
calcul:process(rst,clk)
begin
  if rst = '0' then
    dis <= (others => '0');
    cpt_echo <= 0;
  elsif rising_edge(clk) then
    --si on a detecte un obstacle
    if echo = '1' then
      cpt_echo <= cpt_echo +1;
       --si cpt_echo est egale a 2920 = 2cm pour aller-retour
      if cpt_echo = 2920 then
         dis <= dis +1;
         cpt_echo <= 0;
      end if;
      dis_tmp <= dis;
    elsif echo='0' then --si echo est fini
      cpt_echo <= 0;
      dis <= (others => '0');
    end if;
  end if;
end process;
```

on a déclaré ces signaux interne dans l'architecture de IP_telemetre:

```
architecture Behav of Telemetre_us is
  signal cpt_trig: integer:= 0;
  signal cpt_echo: integer:= 0;
  signal s_trig: std_logic :='0';
  signal dis: unsigned(7 downto 0); --la distance
  signal dis_tmp: unsigned(7 downto 0);
```

2. Test-bench et simulation sous VHDL

Dans un premier temps, on a déclaré le telemetre_us dans l'architecture de test bench:

```
architecture test of tb_us is
--declaration des component pour uut
component telemetre_us
port(
  clk,rst: in std_logic;
  echo: in std_logic;
  trig: out std_logic := '0';
  LEDR: out std_logic_vector(7 downto 0):= (others =>'0')
);
end component;
```

Puis, on a déclaré les signaux internes de IP télémètre et l'horloge du FPGA (***clk_periode***):

```
--input
signal rst: std_logic:='0';
signal clk: std_logic:='0';
signal echo: std_logic:='0';
--output
signal trig: std_logic;
signal LEDR: std_logic_vector(7 downto 0);
--definition de la periode de CLOCK
constant clk_periode: time:=20 ns;
```

On a instancie les signaux et crée d'abord le processus de l'horloge:

```
begin
--Instancier uut
uut : telemetre_us port map(
  clk =>clk,
  rst =>rst,
  echo =>echo,
  trig =>trig,
  LEDR =>LEDR
);
```

```
--clock process
clk_process : process
  begin
    clk<= '0';
    wait for clk_periode/2;
    clk<= '1';
    wait for clk_periode/2;
  end process;
```

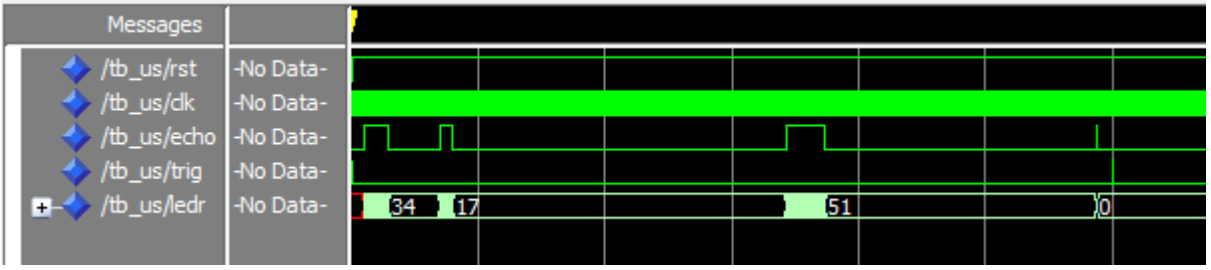
Pour le processus de simulation, on a testé 4 durées différentes de signaux ***Echo*** : 2 ms, 1 ms, 3 ms et 100 ns.

```
simulation:process
begin
  wait for 100 ns;
  rst <= '1';
  wait for 100 ns;
  rst <= '0';
  wait for 100 ns;
  rst <= '1';
  wait for 1 ms;
  echo <= '1';
  wait for 2 ms; -- distance=68 cm
  echo <= '0';
  wait for 4 ms;
  echo <= '1';
  wait for 1 ms; -- distance=34 cm
  echo <= '0';
  wait for 0.3 ms;
  echo <= '0';
  wait for 20 ms;
  echo <= '0';
  wait for 5 ms;
  echo <= '0';
  wait for 1 ms;
  echo <= '1';
  wait for 3 ms;  -- distance=51 cm
  echo <= '0';
  wait for 21.5 ms;
  echo <= '1';
  wait for 100 ns;
  echo <= '0';
  wait;
end process;
end test;
```

```
simulation:process
begin
  wait for 100 ns;
  rst <= '1';
  wait for 100 ns;
  rst <= '0';
  wait for 100 ns;
  rst <= '1';
  wait for 1 ms;
  echo <= '1';
  wait for 2 ms; -- distance=68 cm
  echo <= '0';
  wait for 4 ms;
  echo <= '1';
  wait for 1 ms; -- distance=34 cm
  echo <= '0';
  wait for 0.3 ms;
  echo <= '0';
  wait for 20 ms;
  echo <= '0';
  wait for 5 ms;
  echo <= '0';
  wait for 1 ms;
  echo <= '1';
  wait for 3 ms;  -- distance=51 cm
  echo <= '0';
  wait for 21.5 ms;
  echo <= '1';
  wait for 100 ns;
  echo <= '0';
  wait;
end process;
end test;
```

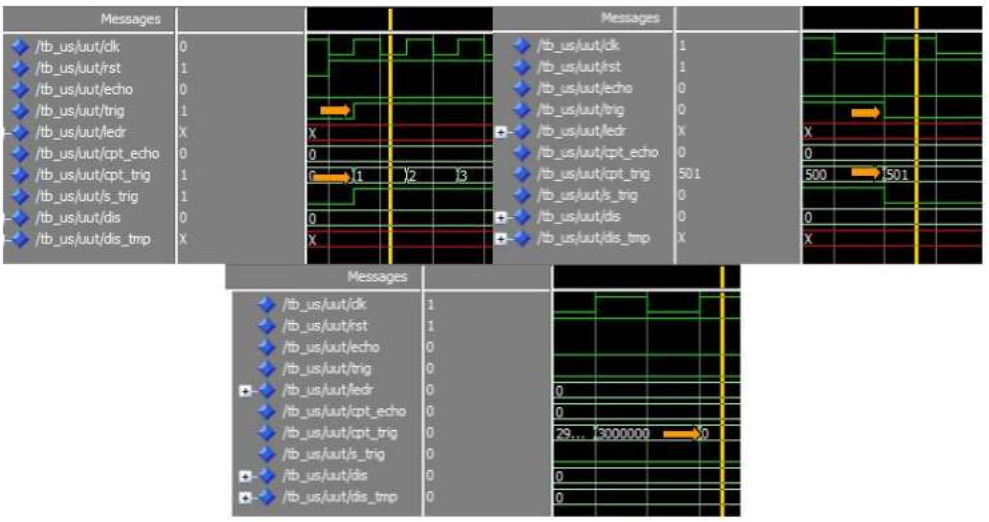
Ces signaux Echo sont dans la première période du signal ***Trig***, On a la simulation dans la figure en dessous, et vous explique en détail.

La simulation globale:

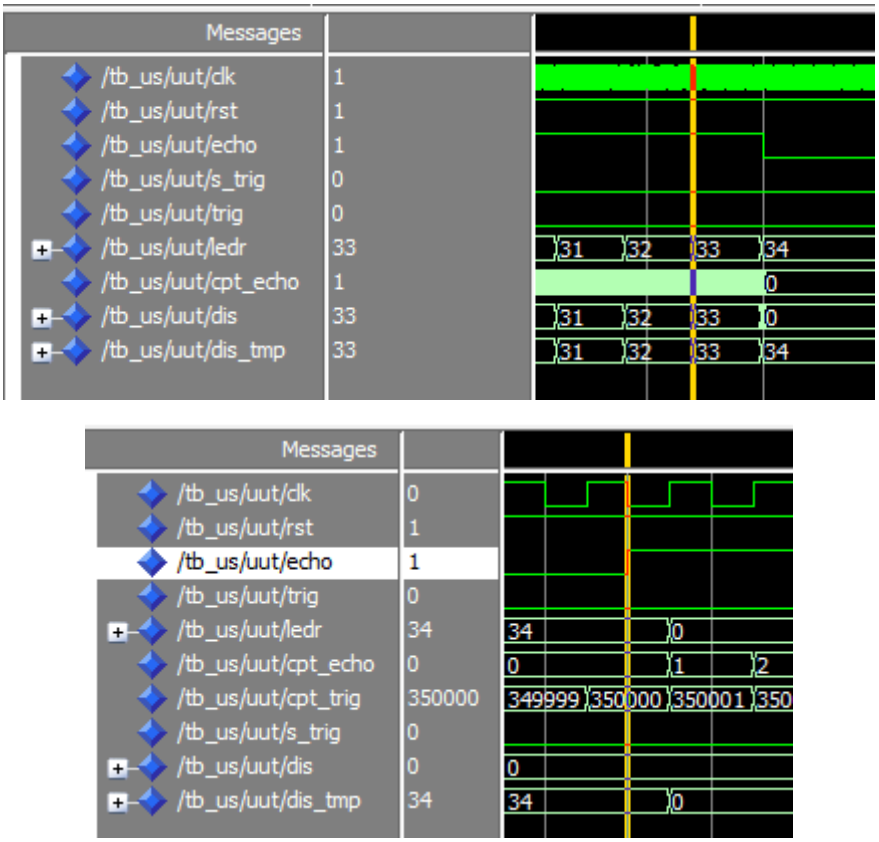


Le calcul déclenche bien dès que le signal ***Echo*** est au niveau “high”, ***LEDR*** a la valeur de la distance pendant la présence de ***Echo*** et retenir la valeur de distance jusqu'à un nouvel signal ***Echo***. Or le dernier ***Echo*** est très petit (100 ns) la distance est d'environ 0.

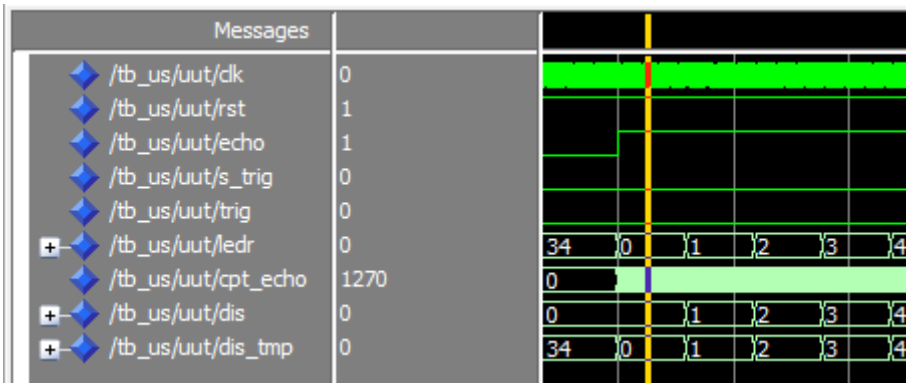
Les détails:



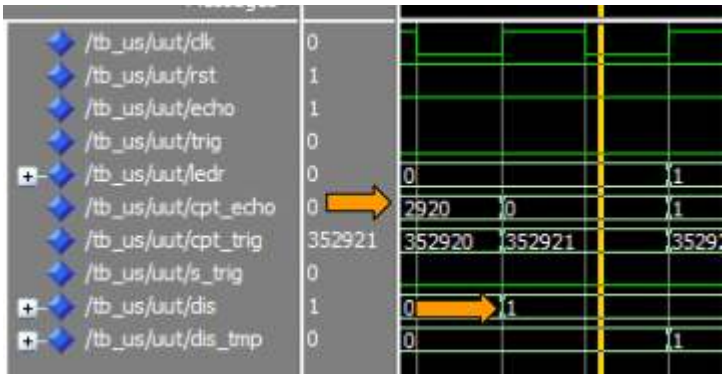
On voit bien que le signal ***trig*** finit bien lorsque ***cpt_trig*** est 500 (10 us), et commence bien si ***cpt_trig*** est 1.



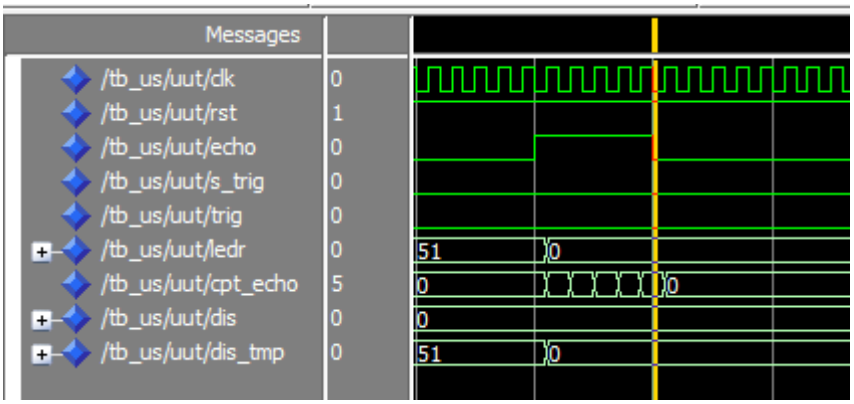
On voit bien que ***cpt_echo*** s'incrémente quand ***Echo*** est présent avec ***clk*** est à '1'. ***cpt_echo*** est aussi mise à zéro lorsque ***Echo*** est fini. Ainsi, ***LEDR*** maintient la valeur de distance s'il n'y a pas de signal ***Echo***.



La valeur de ***LEDR*** est mise à zéro lors de la présence d'un nouvel signal ***Echo***.



Lorsque **cpt_echo** est égale à 2920, il est mise à zéro, dans ce cas la distance s'incrémente.

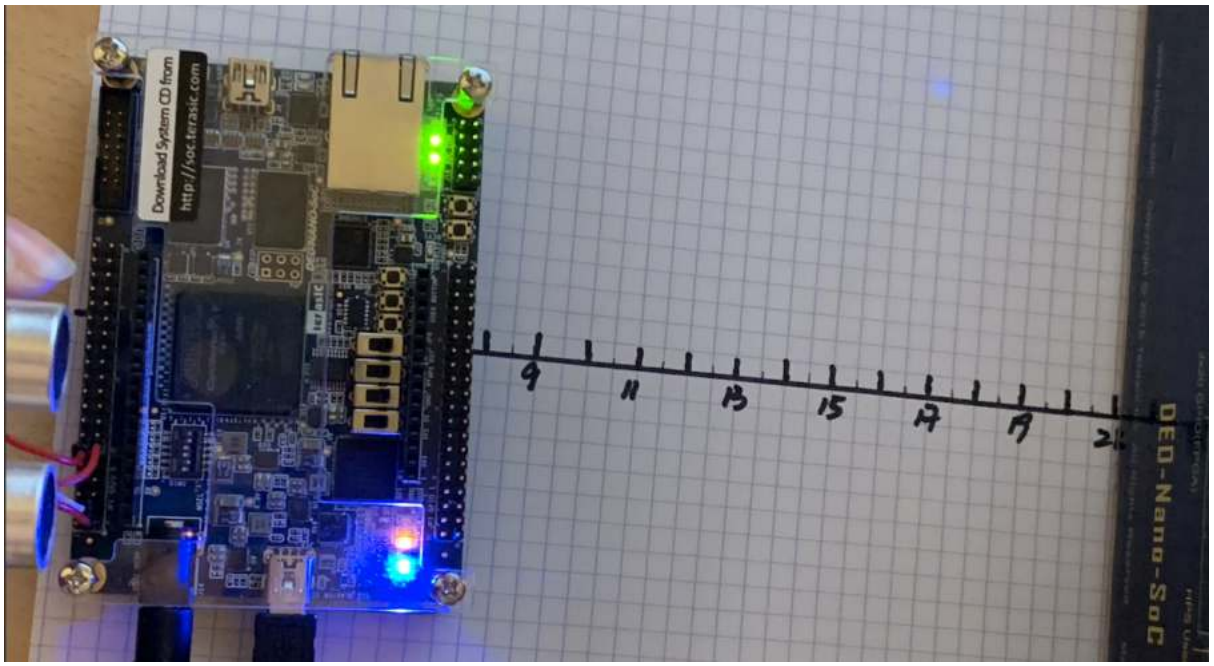


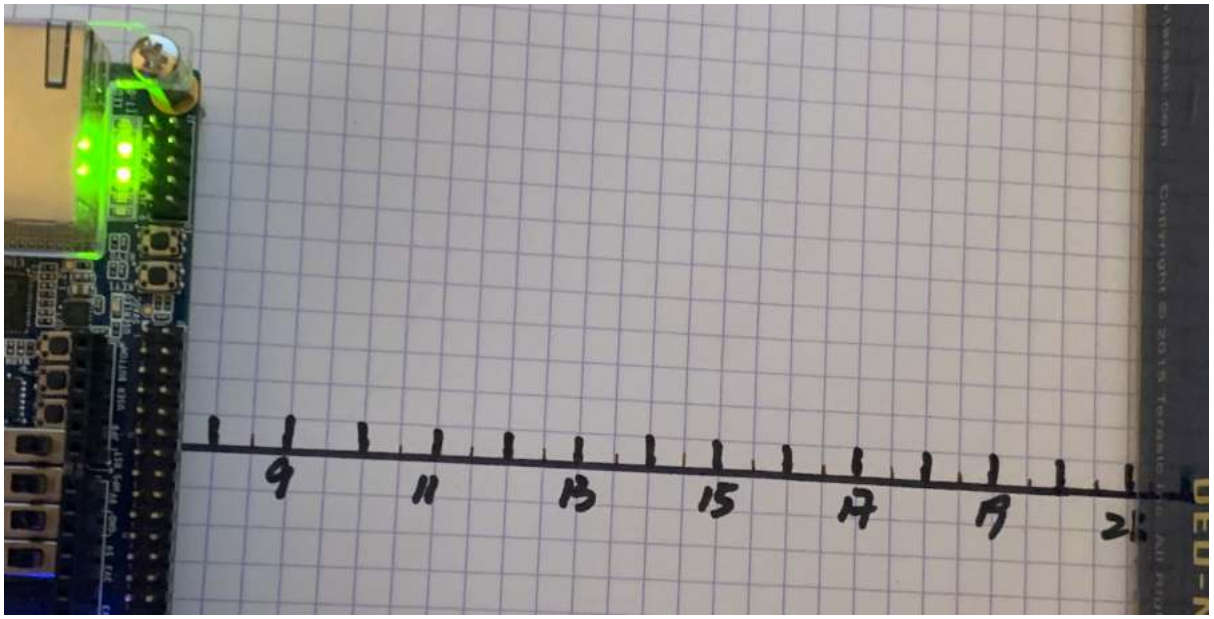
cpt_echo fonctionne même si ***Echo*** est petit (100 ans), mais il est mis à zéro aussi quand ***Echo*** est fini.

C.Test de l’IP sur carte

On test IP sur la carte DE0_Nano_Soc avec plusieurs tests, la valeur binaires des LEDs évolue bien en fonction de la distance entre la boîte et le télémètre.

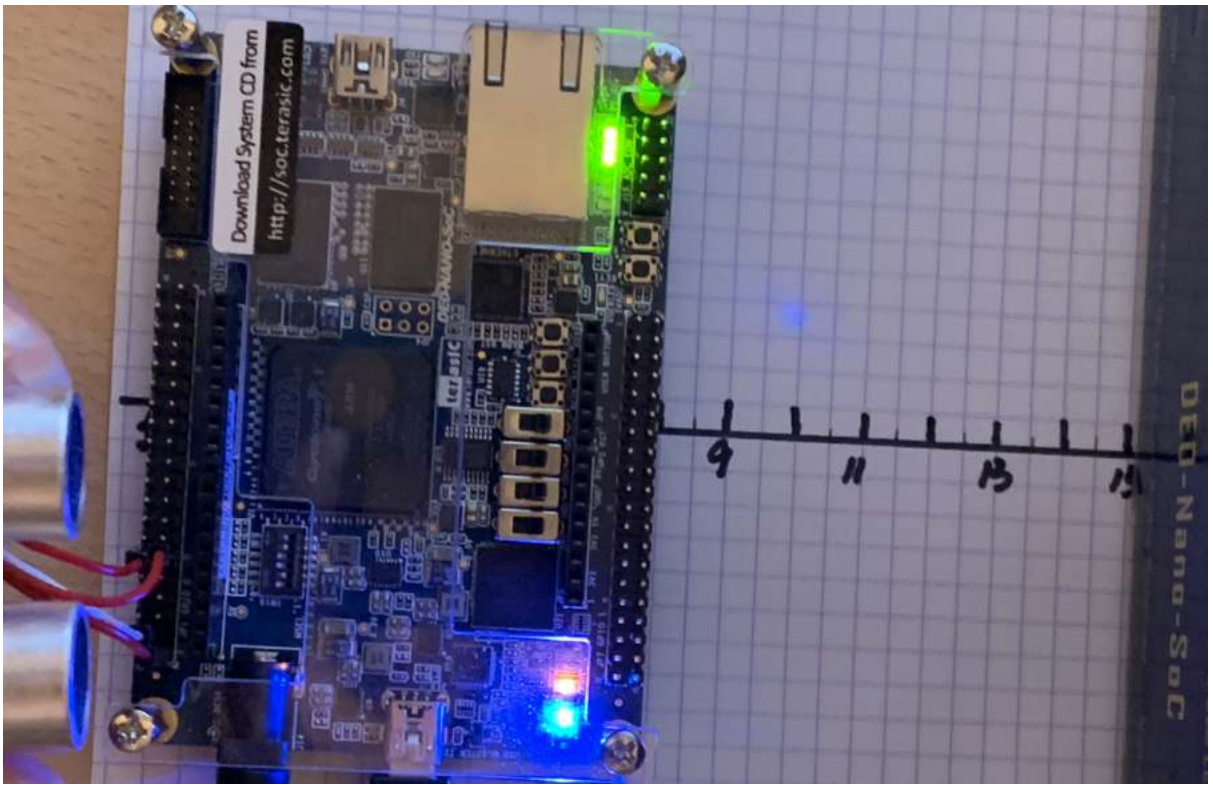
Voici les photos de nos tests :





On a fixé le capteur à 1 cm, la boîte est située à 21 cm, d’où la distance réelle est d'environ 20 cm. On voit que le 3e et le 5e LEDs sont allumés, cela signifie qu’une distance de 10100 en binaire et de 20 cm en décimal. La mesure est correcte.

On fixe aussi le capteur à 1 cm, puis on rapproche la boîte au capteur avec une distance de 14 cm.

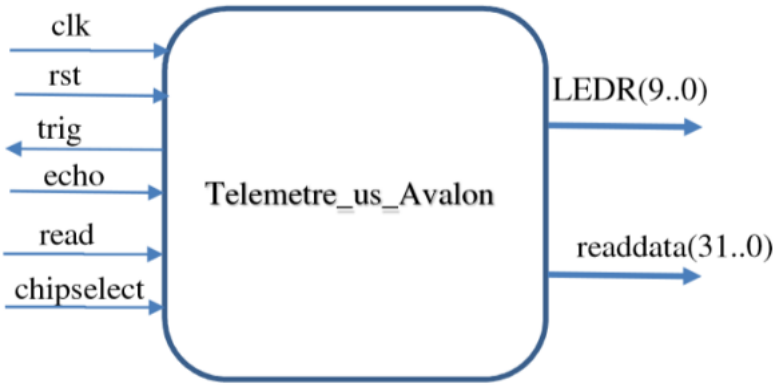


On voit que le 2e, le 3e et le 4e de LEDs sont allumés, d'où une distance de 14 cm (1110) entre la boîte et le télémètre. Cela assure que IP fonctionne bien.

D.Intégration de Bus avalon

1. Code

Afin d'ajouter l'interface avalon dans IP télémètre, on va d'abord modifier notre code en vhdl en ajoutant les entrées: ***read*** et ***chipselect***, et le sortie ***readdata***.



Et notre entité devient comme le figure en dessous:

```
entity Telemetre_us is
port(
  clk: in std_logic; --50MHz
  rst_n: in std_logic; --active en '0'
  echo: in std_logic;
  read: in std_logic;
  chipselect: in std_logic;
  readdata: out std_logic_vector(31 downto 0);
  trig: out std_logic;
  LEDR: out std_logic_vector(7 downto 0)
);
end Telemetre_us;
```

Le bus avalon sert à lire la valeur de distance et stocker la valeur dans le ***readdata*** quand le ***chipselect*** et ***read*** sont en valeur 1.

Voici notre code pour la marche du bus avalon:

```
registre:process(clk, rst_n)
begin
  if rst_n = '0' then
    readdata <= (others => '0');
  elsif rising_edge(clk) then
    if (chipselect = '1') and (read = '1') then
      readdata <= std_logic_vector(dis_tmp);
    end if;
  end if;
end process;
```

En plus, la taille de ***readdata*** est 32 bits, donc on change la taille de ***dis*** et ***dis_tmp*** en 32 bits afin de permet de stocker la valeur de distance dans le ***readdata***. Et on ne lit que les 8 derniers bits du ***dis_tmp*** dans le LEDR.

```
LEDR <= std_logic_vector(dis_tmp(7 downto 0));
```

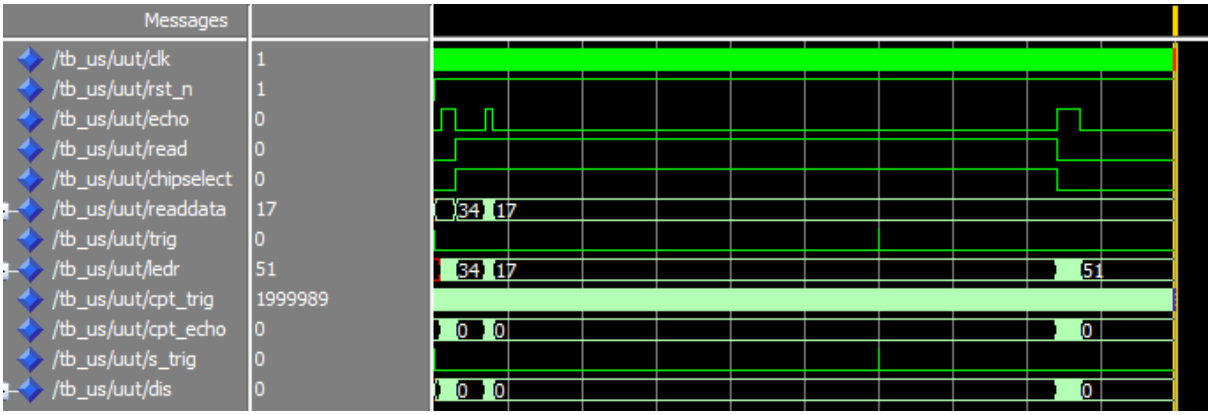
2. Test

Pour la simulation, on ajoute des ports manquant pour le bus avalon. Et on a fait deux différentes simulations, une est que la valeur du signaux ***read*** et ***chipselect*** égale à 1, c'est-à-dire, le signal ***readdata*** va lire la valeur ***dis***. Et l'autre est que la valeur du signaux ***read*** et ***chipselect*** égale à 0, c'est à dire, le signal ***readdata*** ne lit plus la valeur ***dis***.

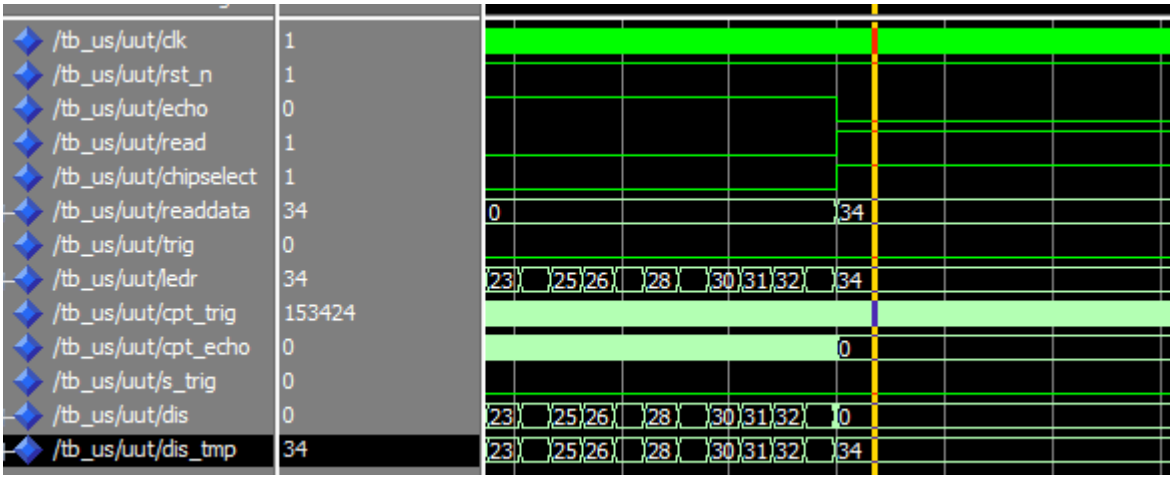
Voici le code pour la simulation:

```
wait for 1 ms;
echo <= '1';
wait for 2 ms; -- distance=34 cm
echo <= '0';
read <= '1';  -- lire la distance
chipselect <= '1';
wait for 4 ms;
echo <= '1';
wait for 1 ms; -- distance=17 cm
echo <= '0';
wait for 70 ms; --pas de obstacle
echo <= '0';
wait for 5 ms;
echo <= '0';
wait for 1 ms;
echo <= '1';
read <= '0';      --ne lit plus la distance
chipselect <= '0';
wait for 3 ms;  -- distance=51 cm
echo <= '0';
wait;
```

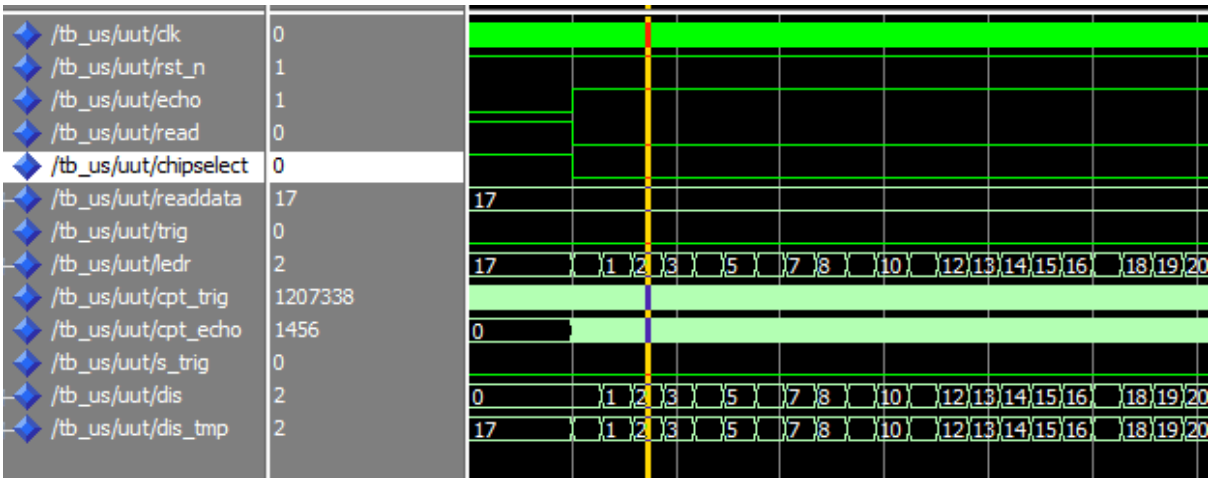
Voici la simulation global:



Le signal ***readdata*** a bien lu la valeur du ***dis_tmp*** quand ***read*** et ***chipselect*** sont en 1.



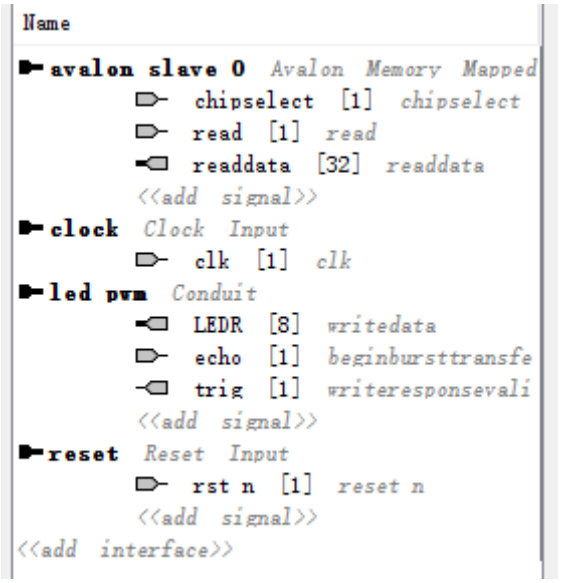
Quand ***read*** et ***chipselect*** sont en 0, le signal ***readdata*** garde l'ancienne valeur, il ne lit plus la nouvelle valeur du ***dis_tmp***.



3. Intégration dans Qsys

Afin d'intégrer le bus Avalon dans notre projet pour programmer sur la carte, on doit ensuite ajouter un nouveau composant nommé ***avalon_pwm*** qui correspond à notre IP télémètre. Dans ce composant, les signaux sont répartis dans 4 interfaces différentes qui sont le ***avalon_slave_0***, ***clock, led_pwm*** et ***reset***.

Voici le composant:



Ensuite on l'ajoute dans le système et relie le ***clock*** de ce composant avec le ***clock*** du système, relier le ***reset*** de ce composant avec le ***reset*** du système et relier le ***avalon_slave_0*** avec ***data_master*** de nios2. Et mettre le ***led_pwm*** en export.

Puis, on instancie le code en vhdl *DE0_nano_SoC_ADC.vhd* avec ce qu'on a obtenu dans le Qsys en utilisant le *generate.*

```
        );
    end component DE0_NANO_SOC_QSYS;
begin
    u0 : component DE0_NANO_SOC_QSYS
        port map (
            adc_ltc2308_conduit_end_CONVST            => ADC_CONVST,
            adc_ltc2308_conduit_end_SCK               => ADC_SCK,
            adc_ltc2308_conduit_end_SDI               => ADC_SDI,
            adc_ltc2308_conduit_end_SDO               => ADC_SDO,
            clk_clk                                   => FPGA_CLK1_50,
            --pll_sys_locked_export                     => CONNECTED_TO_pll_sys
            --pll_sys_outclk2_clk                       => CONNECTED_TO_pll_sys
            reset_reset_n                             => KEY(0),
            sw_external_connection_export             => SW,              --  SW
            avalon_pwm_0_led_pwm_writedata            => LED,             --
            avalon_pwm_0_led_pwm_writeresponsevalid_n => GPIO_0(0), --
            avalon_pwm_0_led_pwm_beginbursttransfer   => GPIO_0(2)    --
        );
```

E. Programmation logicielle et test de l'IP

1. Code en C

Dans cette partie, on doit afficher la distance dans la console en utilisant un code en c.

Alors, pour lire et écrire la valeur de distance, on a besoin deux fonctions, ce sont *IORD_AVALON_DISTANCE* et *IOWR_AVALON_DISTANCE* que nous avons définie dans le fichier *avalon_telemetre_mesure.h.* La fonction *IORD_AVALON_DISTANCE* permet de lire la valeur de *base*, et la fonction *IOWR_AVALON_DISTANCE* permet d' écrire la valeur du *data* dans la variable *base.*

avalon_telemetre_mesure.h | telemetre.c

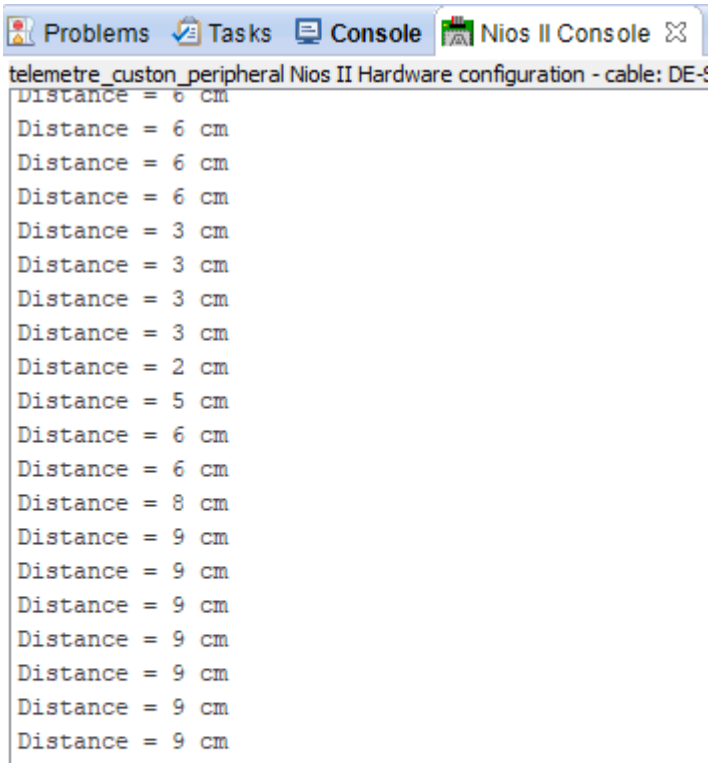
```
/* avalon_telemetre_mesure.h

#ifndef AVALON_TELEMETRE_MESURE_H_
#define AVALON_TELEMETRE_MESURE_H_

#include <io.h>

#define IORD_AVALON_DISTANCE(base)       IORD(base,0)
#define IOWR_AVALON_DISTANCE(base,data) IOWR(base,0,data)

#endif /* AVALON_TELEMETRE_MESURE_H_ */
```

Ensuite, dans le fichier *telemetre.c,* on utilise les deux fonctions. D'abord, on déclare un variable *distance.* Puis, on utilise la fonction *IOWR_AVALON_DISTANCE* pour écrire la valeur 0 dans *AVALON_PWM_0_BASE* afin de faire l'initialisation. Après, dans la boucle infinie, on utilise la fonction *IORD_AVALON_DISTANCE* pour lire la valeur du *AVALON_PWM_0_BASE* et stocker dans la variable *distance.* A la fin, on utilise le *printf* pour afficher la valeur du *distance.*

avalon_telemetre_mesure.h | telemetre.c

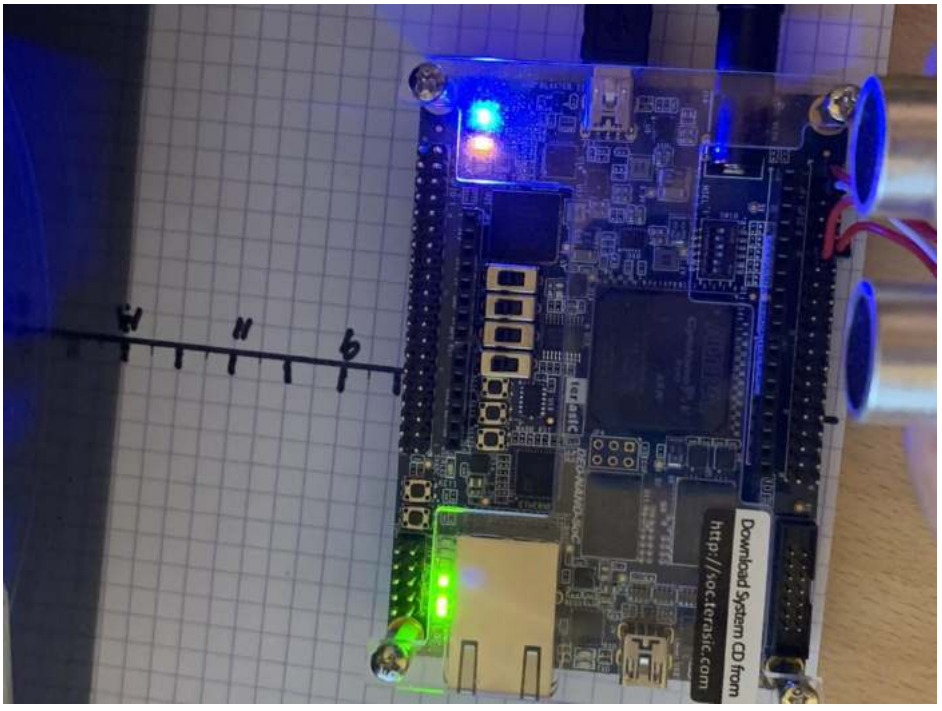
```
/* telemetre.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "system.h"
#include "avalon_telemetre_mesure.h"
int main(){
    int distance;
    printf("Telemetre mesure\n");
    IOWR_AVALON_DISTANCE(AVALON_PWM_0_BASE,0x00);
    while(1){
        usleep(100000);
        distance = IORD_AVALON_DISTANCE(AVALON_PWM_0_BASE);
        printf("Distance = %d cm\n",distance);
    }
    return 0;
}
```

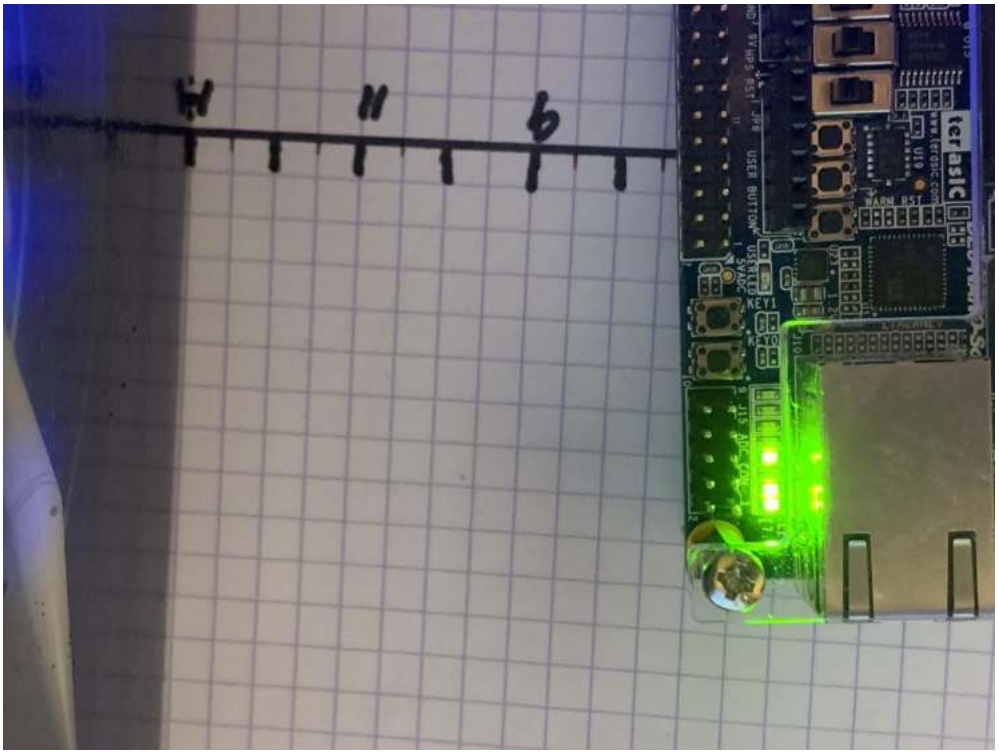
2. Simulation en comparant la valeur afficher et la valeur lire sur la carte FPGA

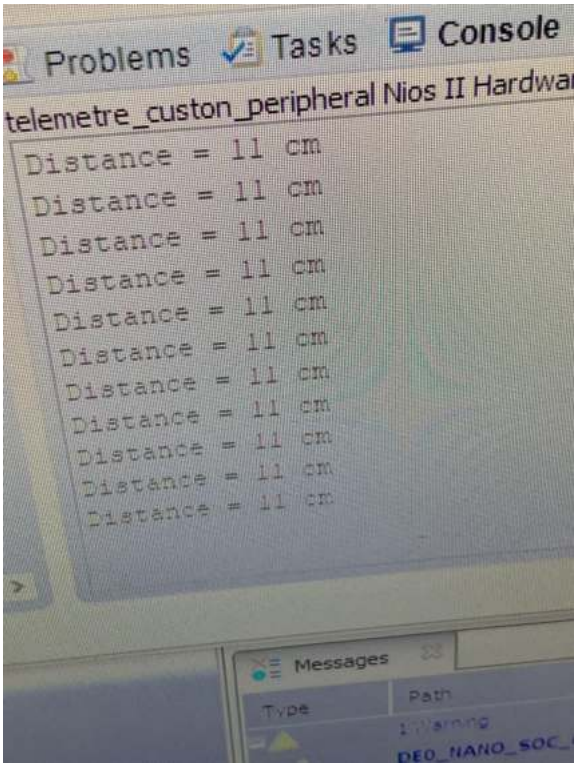


La figure en dessus nous montre bien que notre code fonctionne, or il s'affiche la variation de distance entre l'obstacle et le télémètre.

On a aussi vérifié la cohérence entre l'affichage sur terminal et l'indication par LEDs. Voici sont les photos:







On a fixé la position du télémètre à 2 cm, on a placé la boîte à 13 cm, on voit que le deuxième, le troisième et le quatrième LEDs sont allumés, d'où une distance de 11 cm (1011).

L'affichage sur terminal fonctionne bien, elle affiche la même valeur comme LEDs.

On vous fournit aussi une démonstration en vidéo:
https://drive.google.com/file/d/1Mj8LQgENV5PU0d4GaQvkifL-fFs6yTy6/view?usp=sharing

F. Manipulation sur le code exemplaire

1. Simulation de l'IP

Dans le code exemplaire, il utilise le même processus pour intégrer le bus avalon. Puis il utilise un autre processus pour compter le temps en us.

```
process (RST,CLK)
begin
  if RST='1' then
    Count1      <= 0;
    Ticklus  <= '0';
   echo_r <= '0';
   echo_rr <= '0';
  elsif rising_edge (CLK) then
    echo_r <= echo;       -- resynchronisation du signal echo
  echo_rr <= echo_r;    -- resynchronisation du signal echo

    Ticklus  <= '0';
    if Count1 < Divisor_us-1 then
      Count1 <= Count1 + 1;
    else
      Count1 <= 0;
      Ticklus <= '1';
    end if;
  end if;
end process;
```

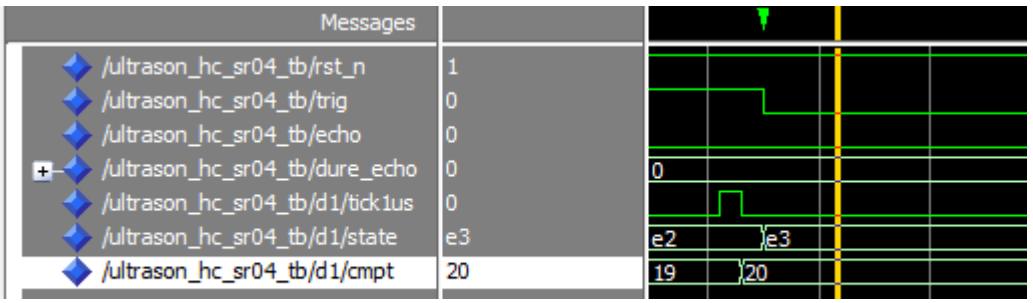
Pour réaliser la détection de l'obstacle par le télémètre ultrason, il utilise la machine à état. Quand *Rst* égale 1, il initialise l'état en état initial E0 et met les signaux en valeur 0. En E0, si le temps passe 1us, il envoie le signal *trig* et passe à l'état suivant: E2. Sinon, il réinitialise la valeur des signaux *cmpt* et *trig* en 0. Dans E2, il compte le temps après l'émission du *trig*, quand le temps atteint à 20us, il passe à l'état E3. Ensuite, il attend 1us pour passer E3 à E4. Dans E4, il attend le début de l'*echo* pour réinitialiser le signal *cmpt* et passer à l'état E5. En E5, il commence à compter le temps pour faire aller-retour entre le télémètre et l'obstacle, si l'*echo* est fini, il passe à l'état E6. Dans E6, il y a deux cas, si le temps d'aller-retour est inférieur 30ms, il passe à l'état E7, sinon E8. En E7, il calcule la distance à partir de

la durée pour faire aller-retour. Pourquoi il divise le temps par 58? Parce que la vitesse de son est de 340m/s => 34e-3 cm/us => durant 1us il peut parcourir 34e-3 cm, alors pour 34cm il faut 1000us. Or *Tick1us* signifie 1us, donc *cmpt* s'incrémente chaque 1us. Alors si *cmpt* est égale à 1000, c'est-à-dire la durée de *echo* est 1000us, cela signifie une distance d'aller-retour de 34cm, la distance entre l'obstacle et le télémètre est 34/2=17cm. Donc la proportion entre la durée et la distance est 1000/17 = 58. Par conséquent, il faut diviser *cmpt* par 58 pour obtenir la distance entre l'obstacle et le télémètre. Puis il passe à l'état E8. Dans E8, on passe à l'état initial quand le temps est supérieur de 60ms.

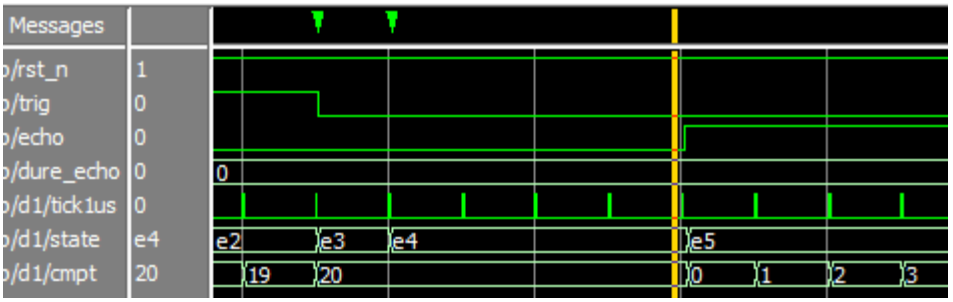
Voici la simulation:

Après chaque l'émission du *trig*, s'il y a un écho, IP mesure la durée de echo => mesurer la distance entre l'obstacle et le télémètre.

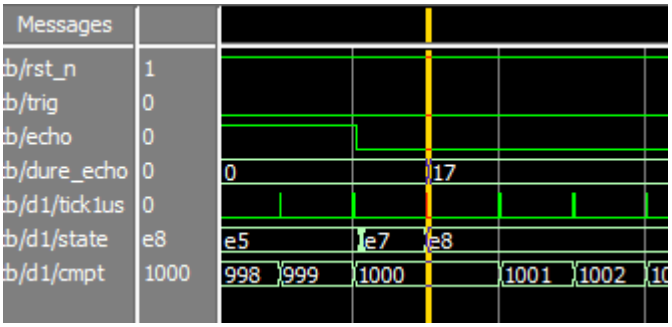
```
# ** Note: t2 - t1 = 20020000 ps
#    Time: 21050 ns  Iteration: 1  Instance: /ultrason_hc_sr04_tb
# ** Note: valeur trig ok
#    Time: 22050 ns  Iteration: 0  Instance: /ultrason_hc_sr04_tb
# ** Note: t2 - t1 = 20020000 ps
#    Time: 60049050 ns  Iteration: 1  Instance: /ultrason_hc_sr04_tb
# ** Note: valeur trig ok
#    Time: 60050050 ns  Iteration: 0  Instance: /ultrason_hc_sr04_tb
# ** Note: t2 - t1 = 20020000 ps
#    Time: 120074050 ns  Iteration: 1  Instance: /ultrason_hc_sr04_tb
# ** Note: valeur trig ok
#    Time: 120075050 ns  Iteration: 0  Instance: /ultrason_hc_sr04_tb
# ** Note: t2 - t1 = 20020000 ps
#    Time: 180098050 ns  Iteration: 1  Instance: /ultrason_hc_sr04_tb
# ** Note: valeur trig ok
#    Time: 180099050 ns  Iteration: 0  Instance: /ultrason_hc_sr04_tb
# ** Note: ****************** Termine !*********************
#    Time: 186100050 ns  Iteration: 0  Instance: /ultrason_hc_sr04_tb
```



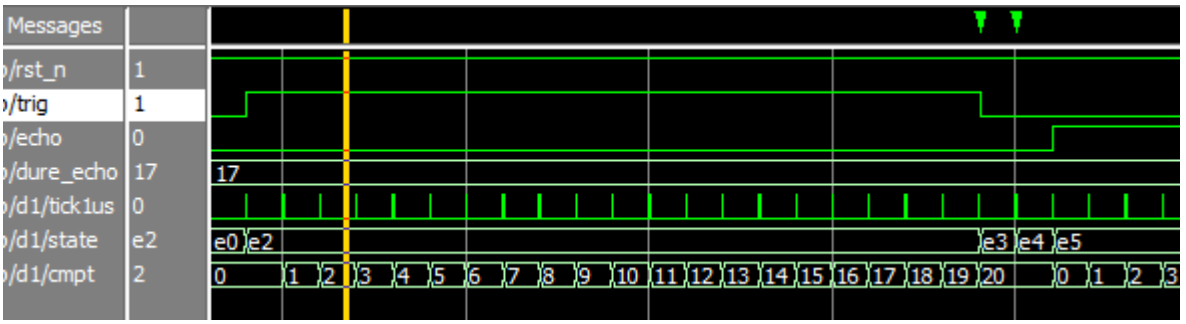
on a bien le temps de *trig* égale 20us (*cmpt*=20) qui est supérieur à 10us afin d'être sûr que l'émission du *trig* a bien réalisé.



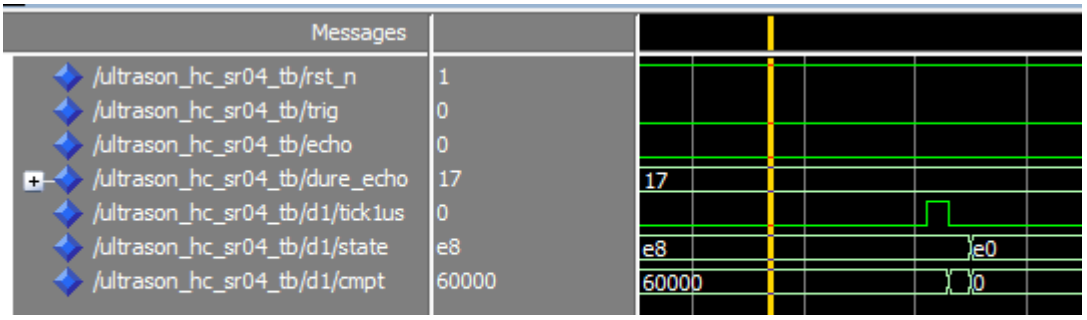
A état E5, *cmpt* a bien remise à zéro (la présence de *echo*, *cmpt* sert à mesurer la durée de *echo*).



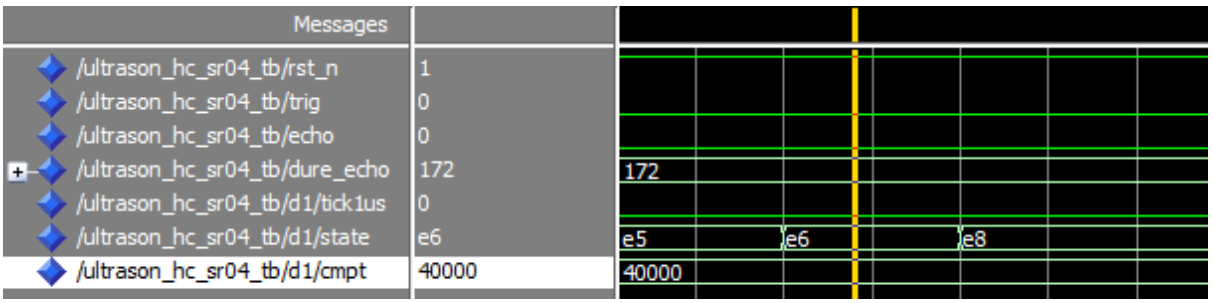
Quand *echo* finit, la durée est inférieure à 30ms, on passe de E6 à E7 et calcule la distance, puis passe à E8. A état E8, il affiche la distance de echo en cm: cmpt/58 environ égale à 17cm.



A E2, on génère un *trig* de 20us. A E3 et E4, on arrête de incrémenter *cmpt*, si echo est présente, on passe à E5.



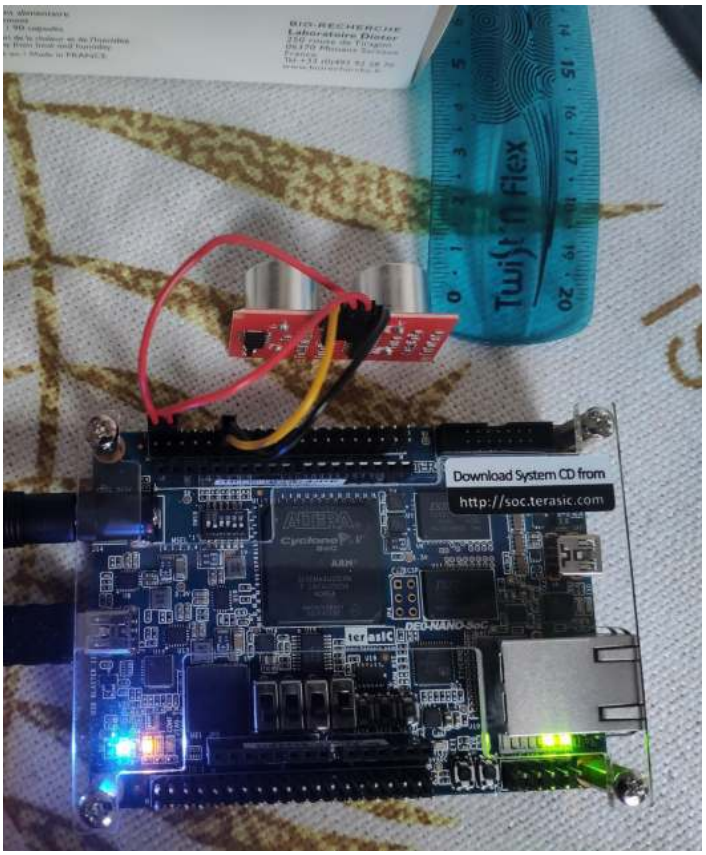
Quand la durée après l'émission du *trig* est supérieur de 60ms, on passe à E0, on re-déclenche la détection.



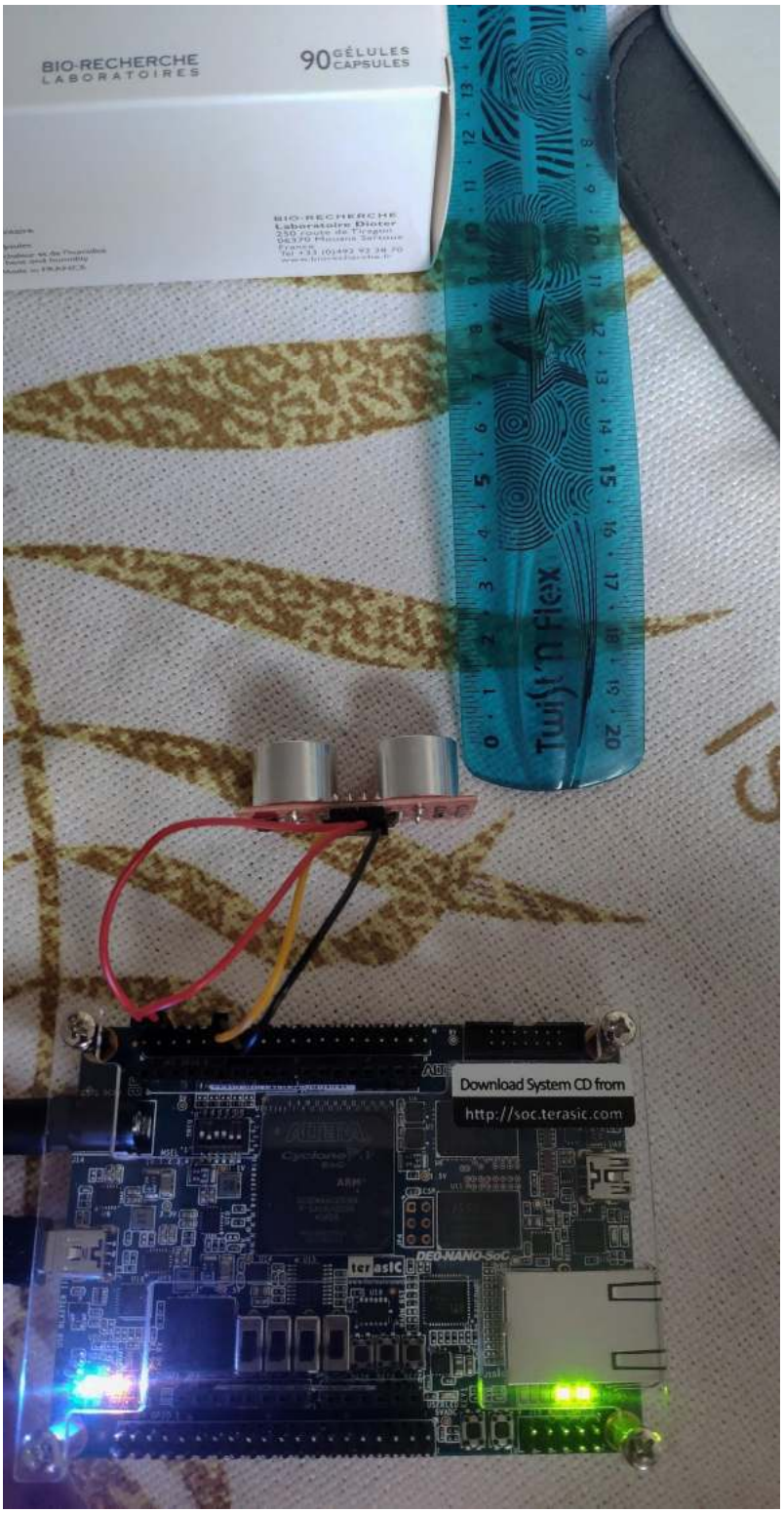
A E6, quand *cmpt* est supérieur à 30ms, on passe directement à E8, car la durée maximum pour aller-retour est 60ms, donc la durée maximum pour aller est 30ms.

2. Test de l'IP sur carte

On a fait le test sur la carte FPGA:

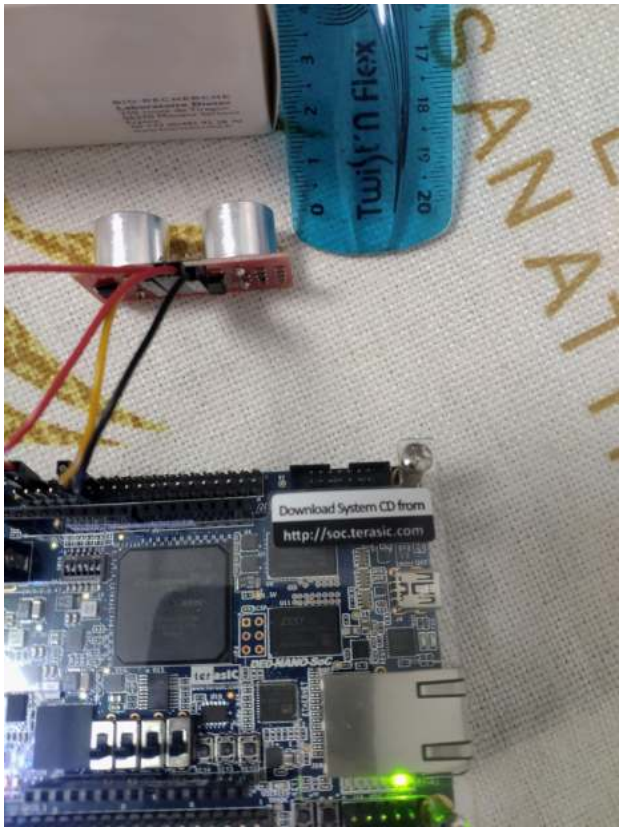


On a bien mesuré la distance entre l'obstacle et le télémètre qui égale 5cm (affichage de LED en 00000101)

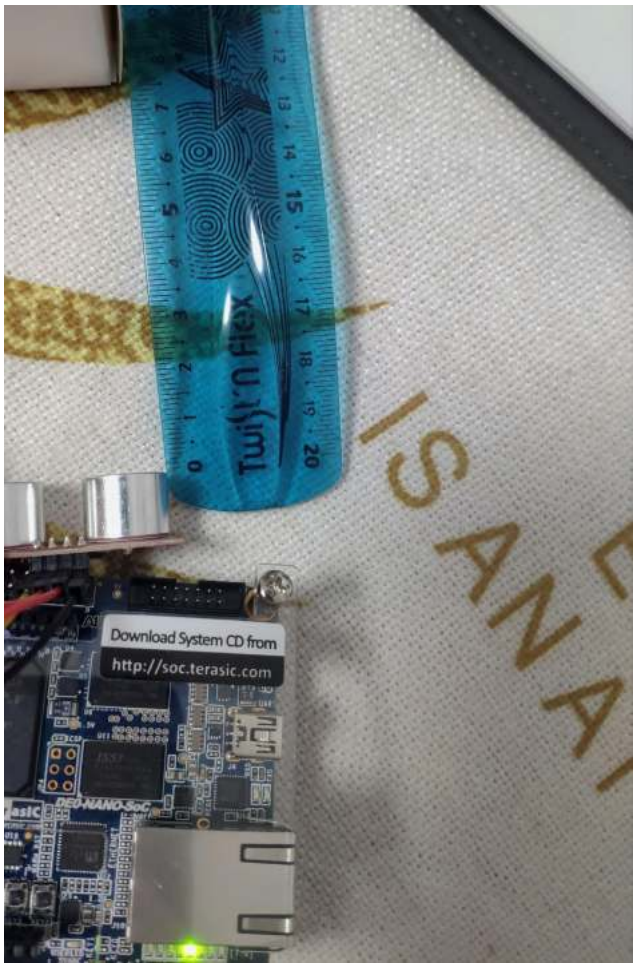


Pour la distance entre l'obstacle et le télémètre est 10cm, on a bien un affichage de LED en 00001010.

3. Intégration de l'IP Télémètre dans Qsys



L'affichage de LED 00000010 correspond bien la distance entre l'obstacle et le télémètre = 2cm.



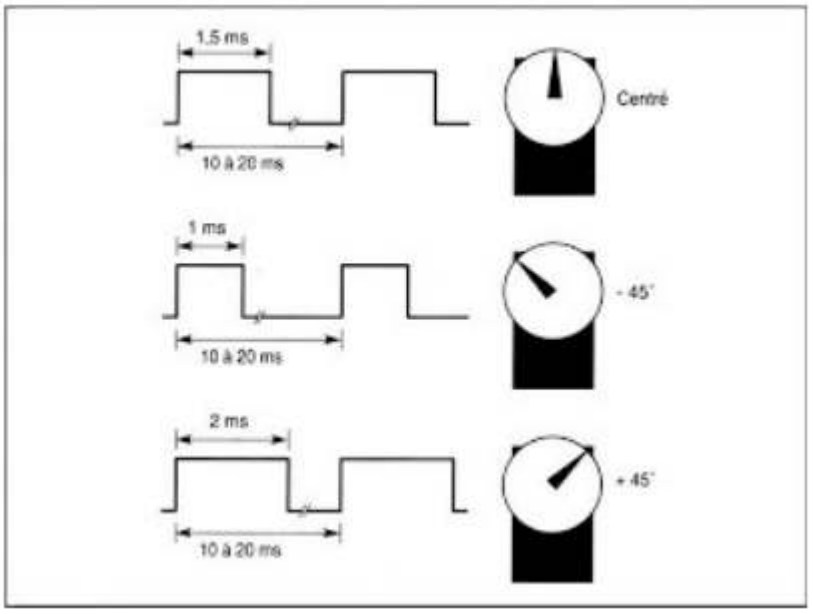
L'affichage de LED 00001000 correspond bien la distance entre l'obstacle et le télémètre = 8cm.

III. Conception de l'IP Servomoteur

A. Principe

Le servomoteur est un système qui permet de produire un mouvement de rotation en réponse à une commande externe. Le servomoteur est asservi en position angulaire à travers un signal codé en largeur d'impulsion qui est la durée d'impulsion.

Les durées d'impulsion sont standard, si l'angle du servo-moteur est 0 degré, cela correspond à 1 ms, si l'angle du servo-moteur est 90 degré, cela correspond à 1.5 ms, si c'est 180 degré, alors la durée d'impulsion est 2 ms. La correspondance entre la longueur d'impulsion et l'angle du servo-moteur est comme la graphique ci-dessous.

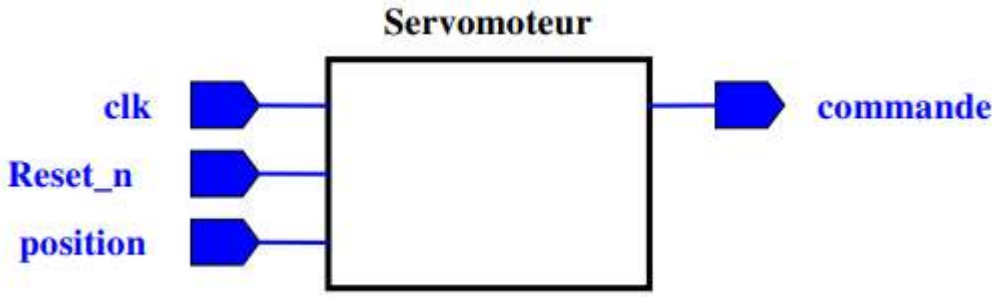


B. Conception de la partie opérationnelle du composant Servomoteur

1. Développement de l'IP Servomoteur

Dans un premier temps nous allons réaliser un composant VHDL qui doit permettre de commander le servomoteur depuis les interrupteurs (switch) afin de vérifier la partie opérationnelle de votre IP.

L'entité à développer en VHDL devra être conforme aux spécifications données ci-dessous :



L'IP devra comporter en entrée :

- un signal position sur 8 ou 10 bit qui permet de choisir la position du servomoteur (dans notre cas, on le réalise sur 4 bit)
- un signal de reset_n (actif à l'état bas).
- une horloge système à 50 MHz.

L'IP devra comporter en sortie :

- un signal commande d'un bit connecté sur l'entrée du servo-moteur.

Selon l'entité de servo-moteur, nous avons rédigé l'entité en VHDL comme la figure ci-dessous:

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity servomoteur is
6     port(
7       clk,Reset_n : in std_logic;  --Reset_n actif a l'etat bas
8       --position: in std_logic_vector(7 downto 0);
9       position: in std_logic_vector(3 downto 0);
10      commande: out std_logic
11    );
12  end servomoteur;
```

Puis nous avons mis les signaux internes:

- signal cpt_periode : permet de compter l'interval de répétition
- signal angle : représente l'angle du servo-moteur
- signal duree : est le temps d'impulsion
- signal impulse : représente la longueur d'impulsion

```
13
14  architecture behav of servomoteur is
15    signal cpt_periode: integer:=1;
16    --signal angle:unsigned(7 downto 0);
17    signal angle:unsigned(3 downto 0);
18    signal duree: integer:= 0;
19    signal impulse: integer:= 0;
```

On avons réalisé 3 processus comme les figures ci-dessous:

1. processus de l'angle qui permet de transformer la valeur de position en la durée de l'impulsion:

Dans un premier temps, on transforme la valeur de ***position*** en valeur de l'***angle*** (de type unsigned). Puis on transforme la valeur de l'***angle*** en valeur de la ***durée*** (de type integer) avec la formule ***"to_integer(angle) *3333+50000"***.

Parce que l'horloge système est 50 MHz soit 20 ns, ainsi la position initiale de servomoteur est 0 degré, soit 50000 de clock. Avec la formule a*angle + b, on a obtenu b = 50000. De plus, avec 7 interrupteurs, la valeur maximale de l'impulsion est 255, on obtient a = 196, ainsi nous avons la carte avec 4 interrupteurs, donc la valeur maximale de la position est 15, donc on a mis **a** = 50000/15 = 3333.

Si la durée est supérieure à 100000, la longueur de l'impulsion est 100000 soit 180 degré de l'angle.

```
21  begin
22    angleprocess : process(clk,Reset_n,position)
23    begin
24    if Reset_n= '0' then  --actif à l?état bas
25      duree <= 50000;--position initiale au 0 degre soit 1ms de l'impulsion
26    elsif clk'event and clk = '1' then
27        angle <= unsigned(position); -- convertir la position en angle
28                                     -- valeur maximale est 15 pour 4 sw, 255 pour 8 sw
29        --duree <= (to_integer(angle)*196+50000);
30        --180 degre soit 255 donc 100000 clk, alors la proportion est 196
31        duree <= (to_integer(angle)*3333+50000);--pour la carte avec 4 sw soit 15 à 180 degre
32        if duree > 100000 then
33         impulse <= 100000;
34       else
35          impulse <= duree;
36        end if;
37    end if;
38  end process;
```

2. processus de l'intervalle qui permet de générer le signal de l'intervalle de 20 ms soit 1000000.

```
Interval_process : process(clk,Reset_n)
    begin
      if Reset_n = '0' then
         cpt_periode <= 1;
       elsif clk'event and clk = '1' then
         if cpt_periode > 1000000 then--une periode de 20ms
            cpt_periode <= 1;
        else
          cpt_periode <= cpt_periode + 1;
        end if;
      end if;
    end process;
```

3. processus de la commande: générer le signal commande qui permet de contrôler la rotation de servomoteur : si la longueur de l'impulsion est supérieure à l'intervalle, la commande est à l'état haut, sinon, la commande est à l'état bas.

```
  Commandeprocess : process(clk,Reset_n)
  begin
    if Reset_n = '0' then
      commande <= '0';
    elsif clk'event and clk = '1' then
      if cpt_periode < impulse then
        commande <= '1';
        else
          commande <= '0';
          end if;
    end if;
  end process;
end behav;
```

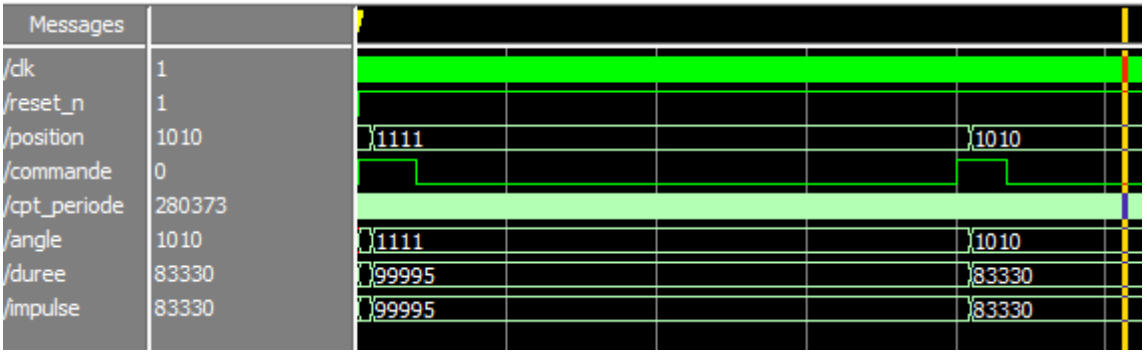
Le test bench de servomoteur est comme la figure ci-dessous:

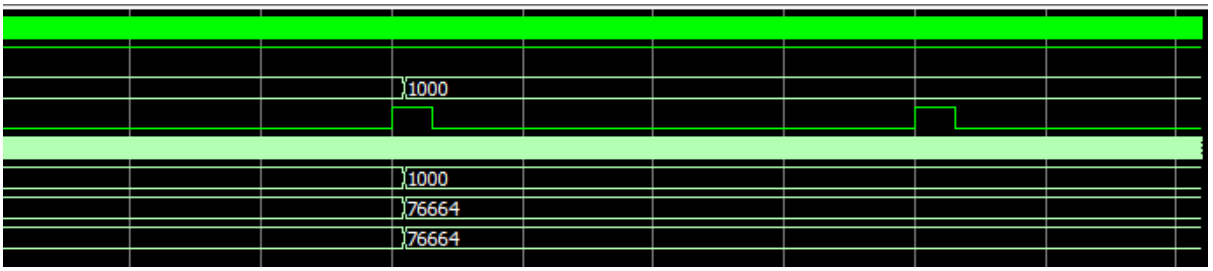
```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5   entity tb_servomoteur is
6   end tb_servomoteur;
7
8   architecture test of tb_servomoteur is
9   component servomoteur
10  port(
11    clk,Reset_n : in std_logic:= '0';  --Reset_n actif a l'etat bas
12      --position: in std_logic_vector(7 downto 0);
13      position: in std_logic_vector(3 downto 0);
14      commande: out std_logic :='0'
15  );
16  end component;
```

```
18  --input
19  signal clk: std_logic:='0';
20  signal Reset_n: std_logic:='0';
21  --signal position: std_logic_vector(7 downto 0):= (others =>'0');
22  signal position: std_logic_vector(3 downto 0):= (others =>'0');
23  --output
24  signal commande: std_logic;
25  --definition de la periode de CLOCK
26  constant clk_periode: time:= 20 ns;
27
28  begin
29  -- Instancier uut
30  uut: servomoteur port map(
31    clk =>clk,
32    Reset_n =>Reset_n,
33    position =>position,
34    commande => commande
35  );
36
37  --clock process
38  clock_process: process
39  begin
40    clk <= '0';
41    wait for clk_periode/2;
42    clk <= '1';
43    wait for clk_periode/2;
44  end process;
46  --simu process
47  simulation: process
48  begin
49    wait for 100 ns;
50    Reset_n <= '1';
51    wait for 100 ns;
52    Reset_n <= '0';
53    wait for 100 ns;
54    Reset_n <= '1';
55    wait for 0.5 ms;
56    --position <= "01110001"; --113 degree
57    position <= "1111"; --150 degree
58    wait for 20 ms;
59    --position <= "01010100"; --84 degree
60    position <= "1010"; -- 100 degree
61    wait for 20 ms;
62    --position <= "00011011"; --27 degree
63    position <= "1000"; -- 80 degree
64    wait;
65
66  end process;
67  end test;
```

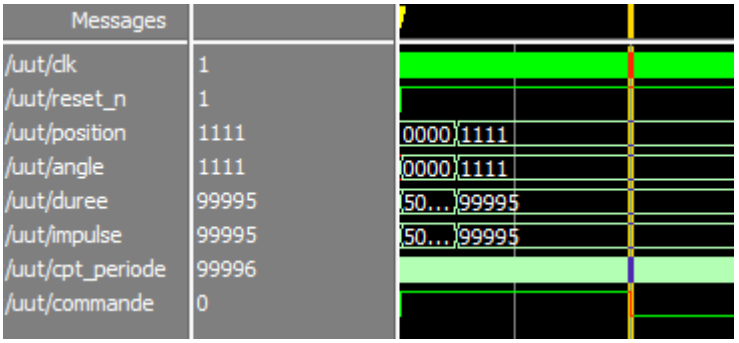
2. Simulation et validation

Lorsque l'on a fait la simulation, nous avons eu la simulation globale comme la figure ci-dessous:

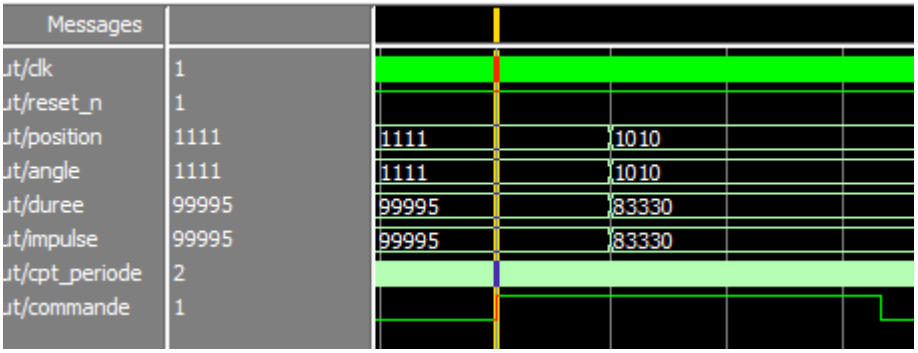


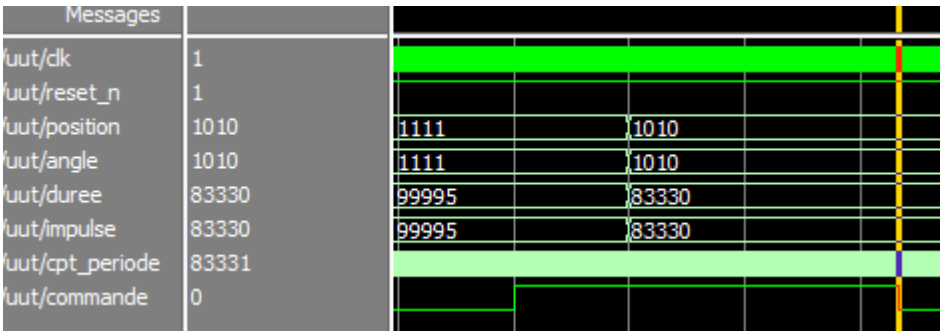


On voit bien que le signal ***commande*** fonctionne.

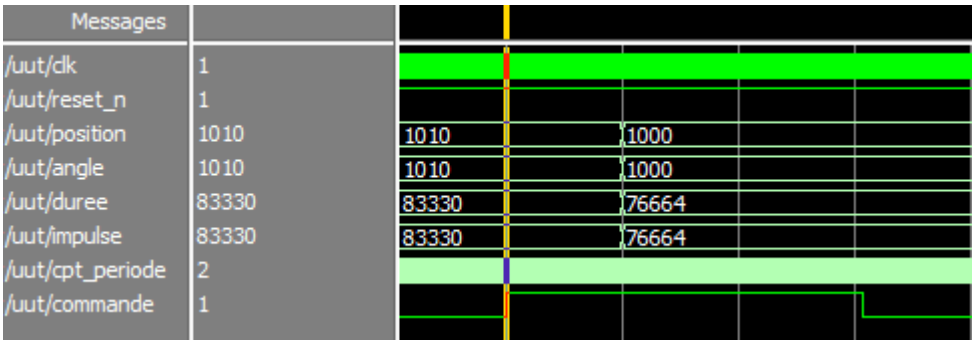


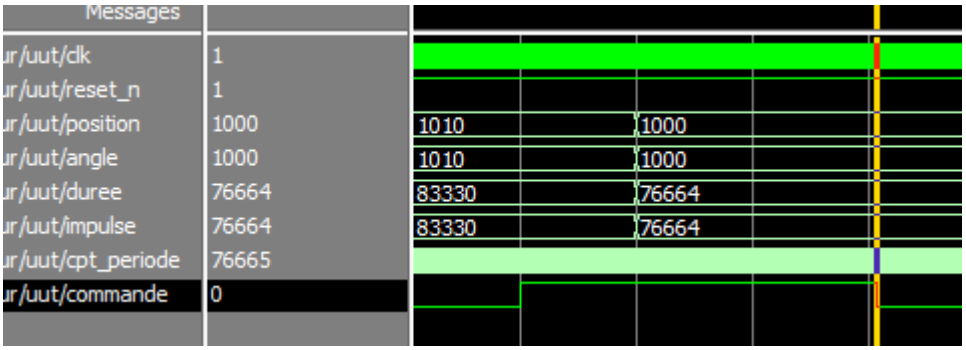
Pour le premier signal de commande, on voit bien que le signal ***commande*** est à l'état haut lorsque ***cpt_periode*** est inférieure au signal ***impulse***. Et la longueur du signal ***commande*** est de 99995 clk, soit 180 degré.



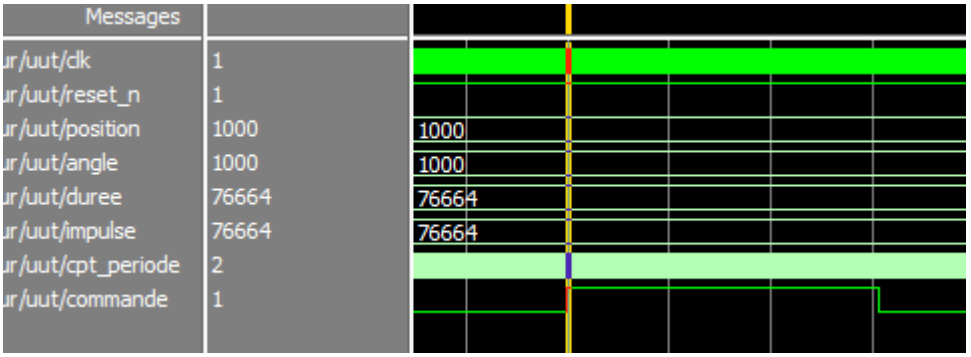


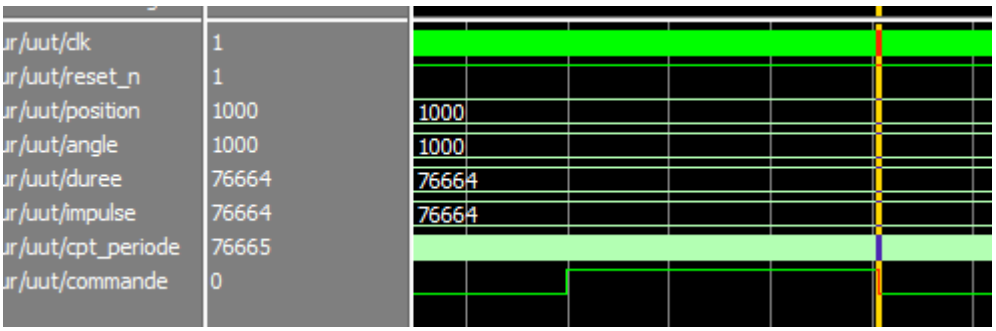
Dans la figure en dessus, on voit que le signal ***commande*** change l'état de 0 à 1 lorsque ***cpt_periode*** a été mise à jour. Puis le signal ***commande*** est allé à l'état 0 si la valeur de ***cpt_periode*** est plus grande que la valeur de ***impulse***. On trouve bien que la durée de la commande est la longueur de l'impulsion.





Les deux figures ci-dessus vérifient que le signal ***commande*** fonctionne bien.

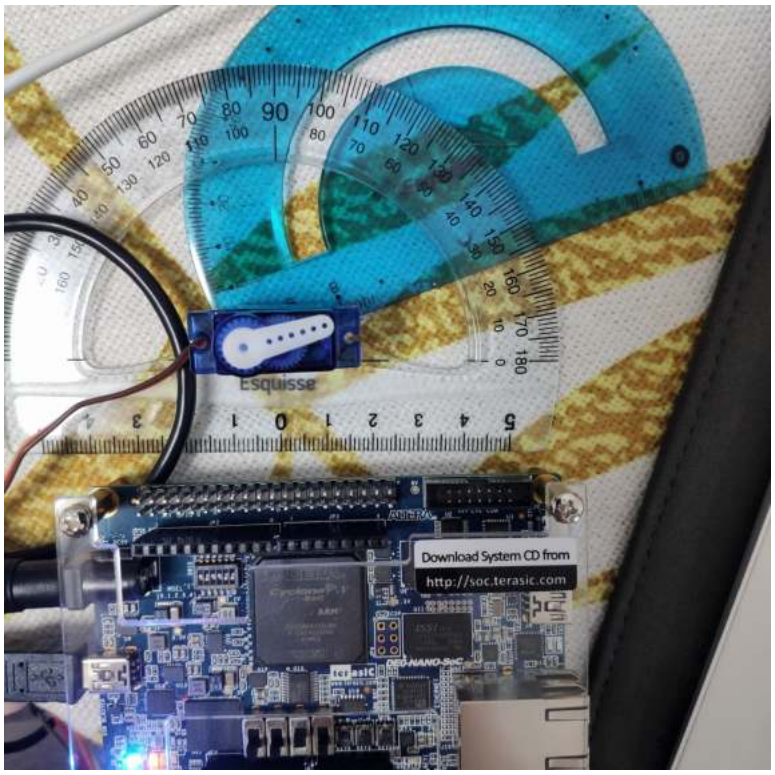




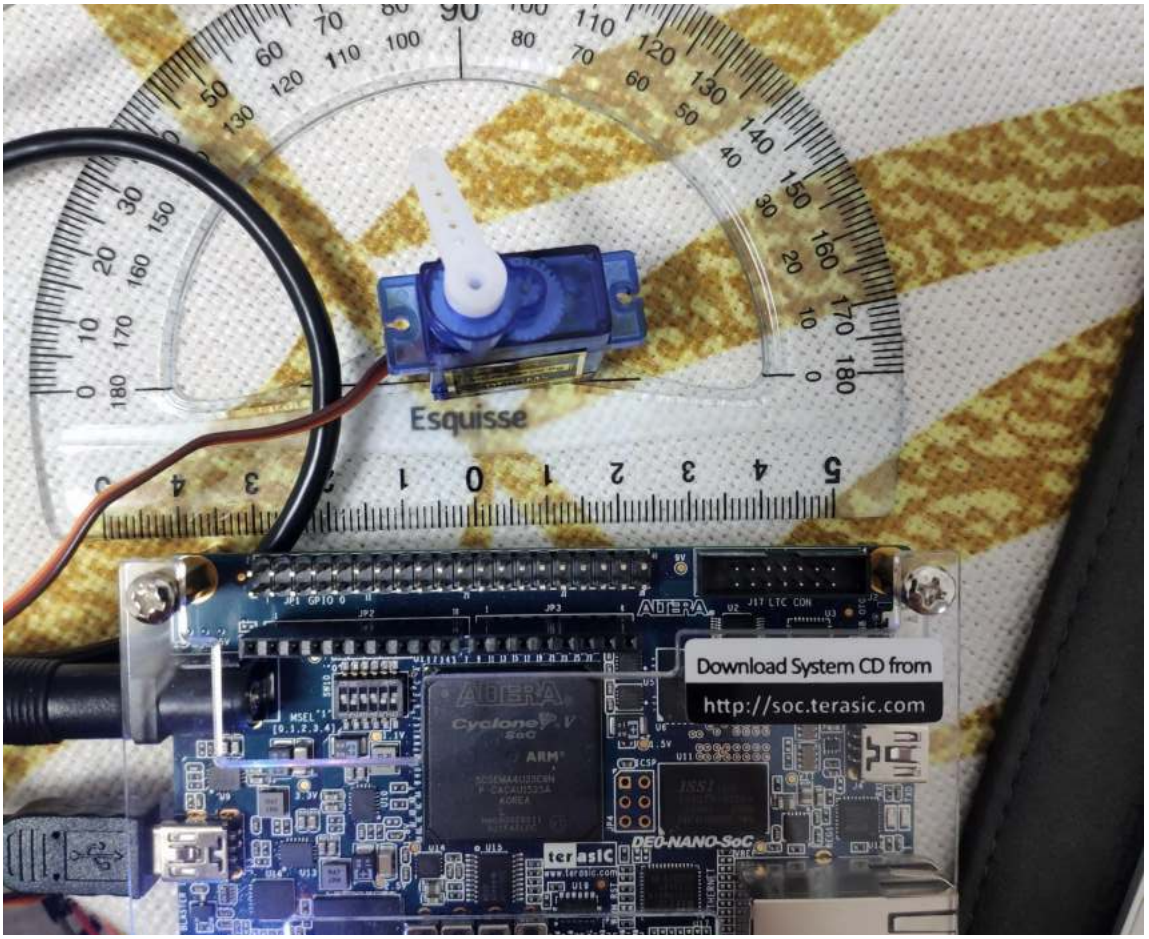
Le dernier signal commande indique que le signal commande sera répété après 20 ms lorsque la valeur de l'impulsion ne change pas.

3. Vérification sur la carte

La valeur de la position est "0000" qui est la position initiale de servomoteur.

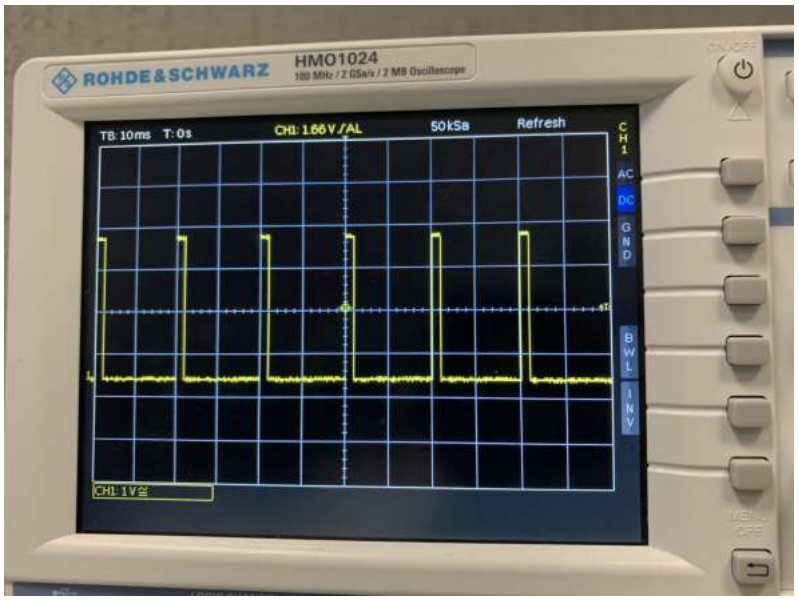


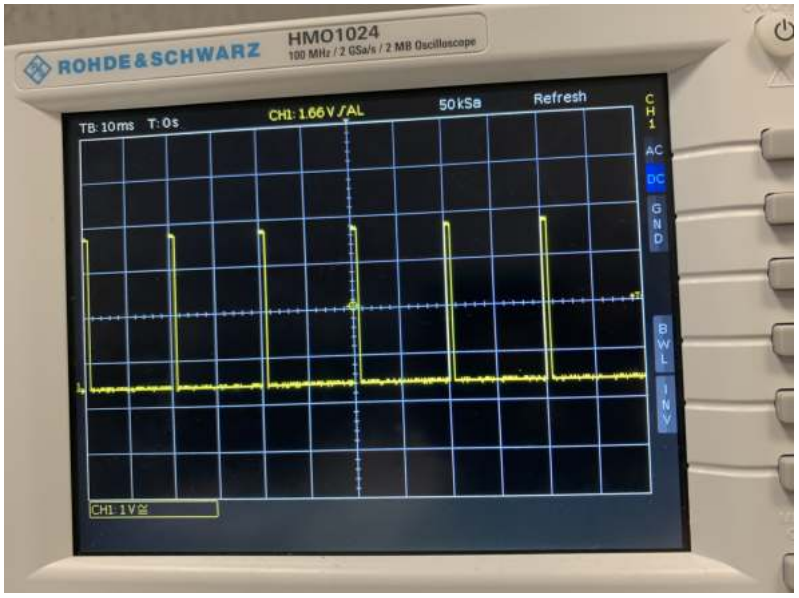
La valeur de la position est “0010”, on obtient bien une rotation de servomoteur de 24 degrés.



Quand la position est “1111”, la rotation doit être de 180 degrés, mais dans en réel, le servomoteur n’a pas tourné de 180 degrés, il a tourné de 70 degrés à peu près.

A l'aide de l'oscilloscope, on a vérifié que le timing de la sortie commande correspond bien aux valeurs attendues. Dans les figures suivantes, on voit bien que le timing de la sortie commande a changé, lorsque on a modifié les valeurs:



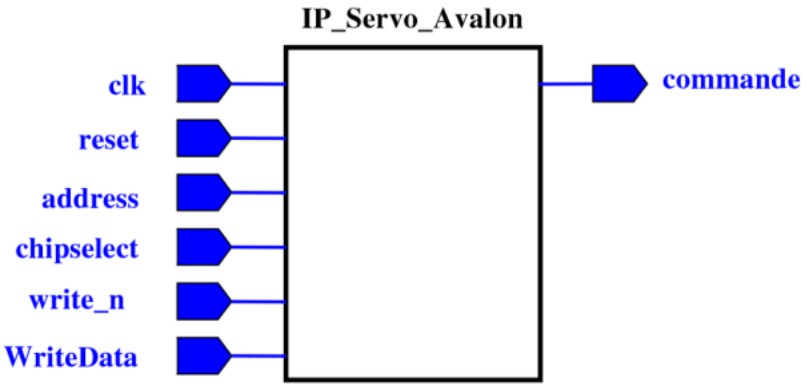


C. Extension de l'IP Servomoteur vers une version connectable au bus Avalon

1. Développement de l'IP

Dans cette partie, nous allons modifier l'IP en ajoutant trois nouveaux entrées dans l'entité de VHDL (*address, chipselect et write_n)* qui permet de prendre en compte les signaux sur le bus Avalon. Les trois nouveaux entrées ont des fonctions

différentes. L'***address*** permet d'accéder aux registres de l'IP. Le ***chipselect*** sélectionne le périphérique sur le bus Avalon. Et le ***write_n*** donne l'autorisation de lecture dans les registres de l'IP, il active par l'état bas.



Et notre entité devient comme le figure en dessous:

```
port(
  clk,Reset_n : in std_logic:= '0';  --Reset_n actif a l'etat bas
  address, chipselect, write_n: in std_logic;
  --position: in std_logic_vector(7 downto 0);
  position: in std_logic_vector(3 downto 0);
  commande: out std_logic :='0'
);
end component;
```

Le bus avalon sert à lire la valeur de ***position***(=WriteData) afin de modifier la valeur de ***commande*** quand le ***chipselect*** et ***address*** sont en valeur 1 et le write_n est en valeur 0.

Pour réaliser cette condition dans notre IP et améliorer l'IP, on a ajouté juste 4 lignes de code qui se situent sur la ligne 28, 39, 40 et 41.

Voici notre code:

```
28      if address = '1' and chipselect = '1' and write_n = '0' then
29        angle <= unsigned(position); -- convertir la position en angle
30                                     -- valeur maximale est 15 pour 4 sw, 255 pour 8 sw
31        --duree <= (to_integer(angle)*196+50000);
32        --or 180 degre corresponde a 100000 clk, alors la proportion est 196
33        duree <= (to_integer(angle)*3333+50000);--pour la carte avec 4 sw soit 15 à 180 degre
34        if duree > 100000 then
35         impulse <= 100000;
36       else
37          impulse <= duree;
38        end if;
39      else
40        impulse <= 0;
41      end if;
```

2. Simulation et validation

Pour vérifier le bon fonctionnement de l'IP, on a ajouté quelques lignes de code dans notre test-bench.

Au début, le *address, chipselect* et *write_n* sont tous en état inactifs.

```
22    signal address: std_logic:='0';
23    signal chipselect: std_logic:='0';
24    signal write_n: std_logic:='1';
```

Puis, on les active pour faire les trois premières rotation demandés.

```
63        address <= '1';
64        chipselect <= '1';
65        write_n <= '0';
66        wait for 1 ms;
67        --position <= "01110001"; --80 degree
68        position <= "1111"; --180 degree
69        wait for 20 ms;
70        --position <= "01010100"; --59 degree
71        position <= "1010"; -- 120 degree
72        wait for 20 ms;
73        --position <= "00011011"; --19 degree
74        position <= "1000"; -- 96 degree
75        wait for 20 ms;
```

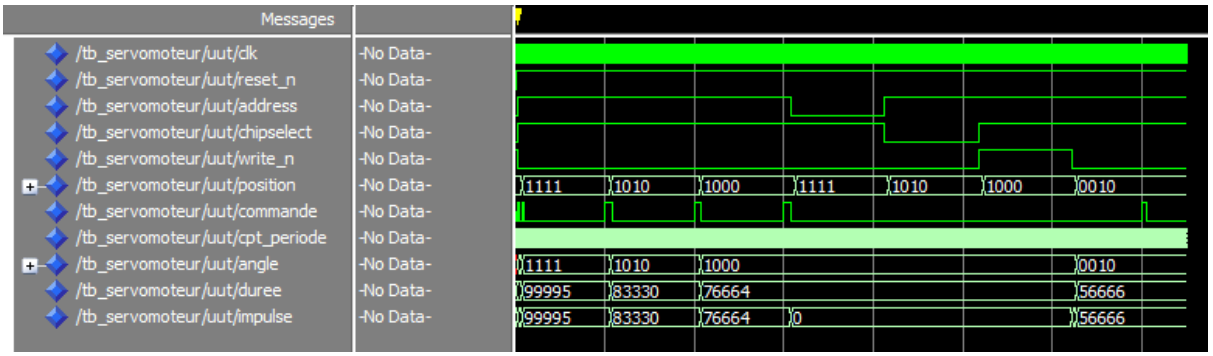
Ensuite, on désactive un des trois signaux afin de ne pas réaliser la rotation

```
76        address <= '0';
77        wait for 1 ms;
78        position <= "1111"; --180 degree (pas de résultat)
79        wait for 20 ms;
80        address <= '1';
81        chipselect <= '0';
82        wait for 1 ms;
83        position <= "1010"; -- 120 degree (pas de résultat)
84        wait for 20 ms;
85        chipselect <= '1';
86        write_n <= '1';
87        wait for 1 ms;
88        position <= "1000"; -- 96 degree (pas de résultat)
```

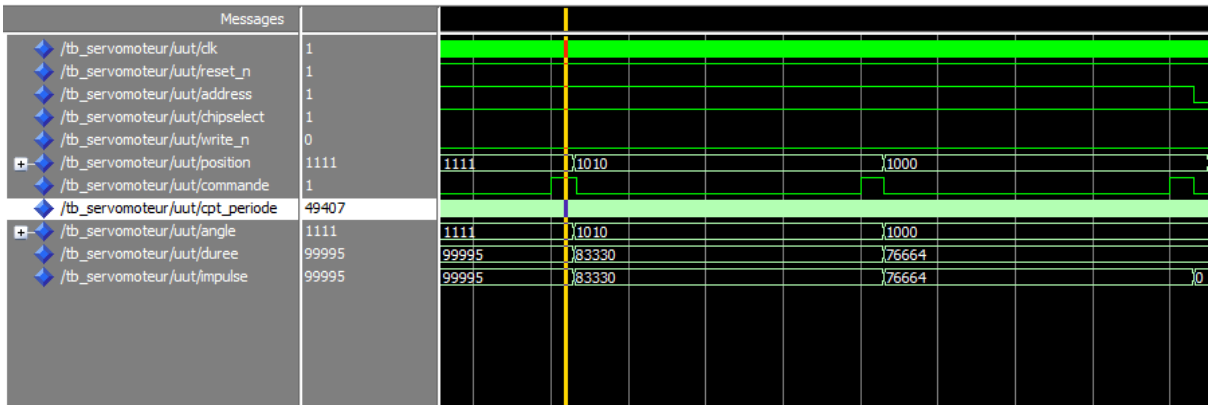
Et on réactif tous les trois signaux et réalise un rotation:

```
90        write_n <= '0';
91        wait for 1 ms;
92        position <= "0010"; -- 24 degree
93
```

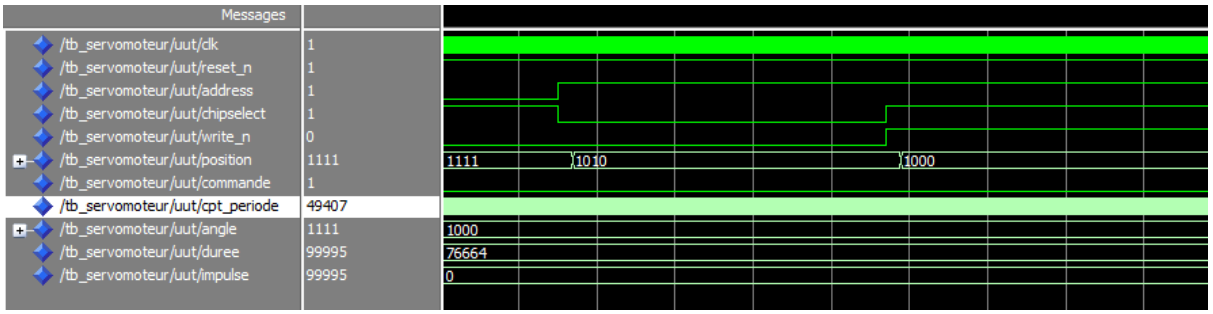
Voici la simulation global:



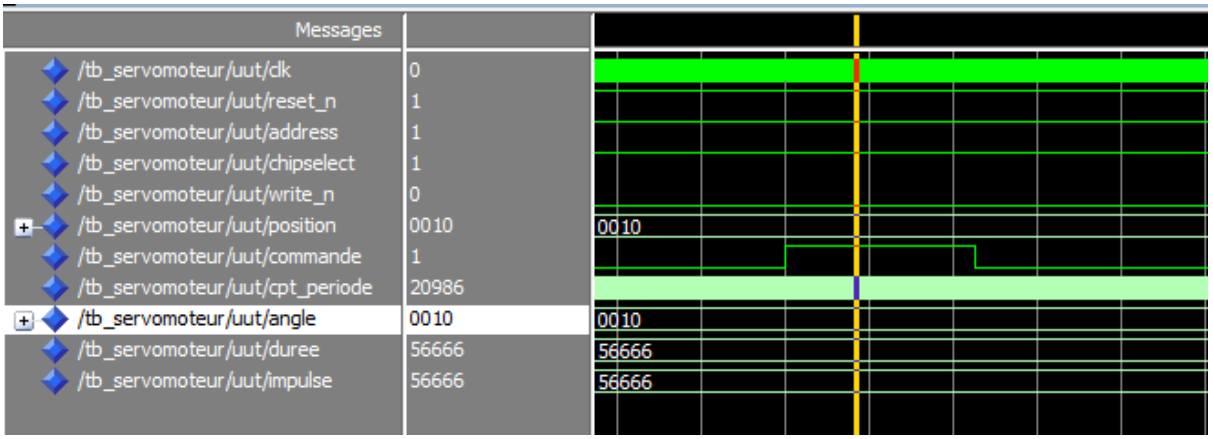
On a bien réussi à réaliser les trois premières rotations demandées, les signaux ***duree*** et ***impulse*** ont les résultats qui correspondent bien l'***angle*** demandé, et la ***commande*** passe de 0 à 1 et 1 à 0, comme la figure au-dessous.



Et pour les trois rotations suivantes, on n'a pas réalisé, car l'un des trois signaux (***address***, ***chipselect*** *et* ***write_n***) n'est pas actif. Et bien que la ***commande*** ne change pas, il reste en 0. En plus, la valeur des signaux ***duree*** et ***impulse*** ont les résultats qui ne correspondent plus l'***angle*** demandé, comme la figure ci-dessous.



Quand on réactive les trois signaux, on réalise bien la rotation demandée, comme la figure ci-dessous.

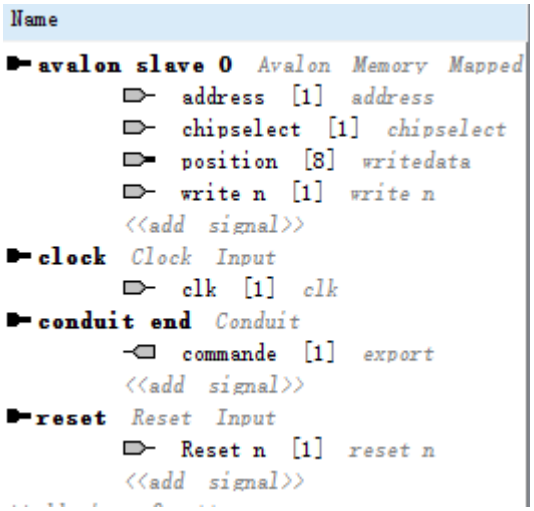


IV. Intégration de votre IP dans Qsys et programmation logicielle

A.Intégration matérielle de l'IP Servomoteur

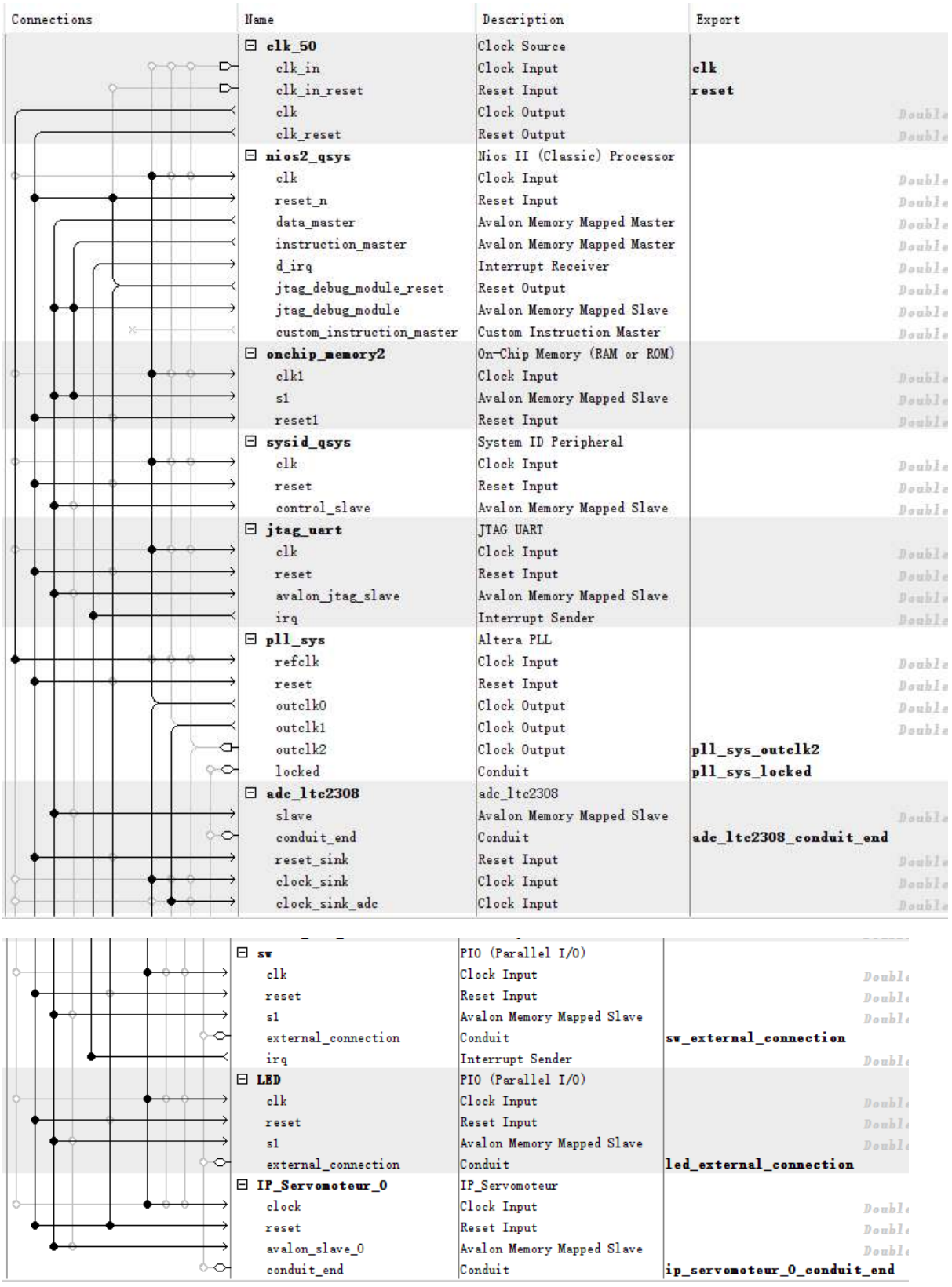
Afin d'intégrer le bus Avalon dans notre projet pour programmer sur la carte, on doit ensuite ajouter un nouveau composant nommé ***IP_Servomoteur***. Dans ce composant, les signaux sont répartis dans 4 interfaces différentes qui sont le ***avalon_slave_0***, ***clock, conduit end*** et ***reset***.

Voici le composant:



Ensuite on l'ajoute dans le système et relie le ***clock*** de ce composant avec le ***clk*** du composant ***nios2_qsys***, relie le ***reset*** de ce composant avec le ***clk_reset*** du

composant ***clk_50*** et relie le ***avalon_slave_0*** avec ***data_master*** du composant ***nios2_qsys***. Et mettre le ***conduit_end*** en export. Les connexions entre les composantes sont comme la figure ci-dessous.



Connections	Name	Description	Export
	clk_50	Clock Source	
	clk_in	Clock Input	**clk**
	clk_in_reset	Reset Input	**reset**
	clk	Clock Output	
	clk_reset	Reset Output	
	nios2_qsys	Nios II (Classic) Processor	
	clk	Clock Input	
	reset_n	Reset Input	
	data_master	Avalon Memory Mapped Master	
	instruction_master	Avalon Memory Mapped Master	
	d_irq	Interrupt Receiver	
	jtag_debug_module_reset	Reset Output	
	jtag_debug_module	Avalon Memory Mapped Slave	
	custom_instruction_master	Custom Instruction Master	
	onchip_memory2	On-Chip Memory (RAM or ROM)	
	clk1	Clock Input	
	s1	Avalon Memory Mapped Slave	
	reset1	Reset Input	
	sysid_qsys	System ID Peripheral	
	clk	Clock Input	
	reset	Reset Input	
	control_slave	Avalon Memory Mapped Slave	
	jtag_uart	JTAG UART	
	clk	Clock Input	
	reset	Reset Input	
	avalon_jtag_slave	Avalon Memory Mapped Slave	
	irq	Interrupt Sender	
	pll_sys	Altera PLL	
	refclk	Clock Input	
	reset	Reset Input	
	outclk0	Clock Output	
	outclk1	Clock Output	
	outclk2	Clock Output	**pll_sys_outclk2**
	locked	Conduit	**pll_sys_locked**
	adc_ltc2308	adc_ltc2308	
	slave	Avalon Memory Mapped Slave	
	conduit_end	Conduit	**adc_ltc2308_conduit_end**
	reset_sink	Reset Input	
	clock_sink	Clock Input	
	clock_sink_adc	Clock Input	
	sw	PIO (Parallel I/O)	
	clk	Clock Input	
	reset	Reset Input	
	s1	Avalon Memory Mapped Slave	
	external_connection	Conduit	**sw_external_connection**
	irq	Interrupt Sender	
	LED	PIO (Parallel I/O)	
	clk	Clock Input	
	reset	Reset Input	
	s1	Avalon Memory Mapped Slave	
	external_connection	Conduit	**led_external_connection**
	IP_Servomoteur_0	IP_Servomoteur	
	clock	Clock Input	
	reset	Reset Input	
	avalon_slave_0	Avalon Memory Mapped Slave	
	conduit_end	Conduit	**ip_servomoteur_0_conduit_end**

Puis, on instancie le code en VHDL *DE0_nano_SoC_ADC.vhd* avec ce qu'on a obtenu dans le Qsys en utilisant le *generate.*

```
begin
    u0 : component DE0_NANO_SOC_QSYS
        port map (
            adc_ltc2308_conduit_end_CONVST      => ADC_CONVST,      --      adc_ltc2308_conduit_end.CONVST
            adc_ltc2308_conduit_end_SCK         => ADC_SCK,         --                             .SCK
            adc_ltc2308_conduit_end_SDI         => ADC_SDI,         --                             .SDI
            adc_ltc2308_conduit_end_SDO         => ADC_SDO,         --                             .SDO
            clk_clk                             => FPGA_CLK1_50,    --                                clk.cl
            ip_servomoteur_0_conduit_end_export => GPIO_1(1), -- ip_servomoteur_0_conduit_end.export
            led_external_connection_export      => LED,       --       led_external_connection.export
            --pll_sys_locked_export               => CONNECTED_TO_pll_sys_locked_export,   --              pll_sys
            --pll_sys_outclk2_clk                 => CONNECTED_TO_pll_sys_outclk2_clk,     --            pll_sys_
            reset_reset_n                       => KEY(0),          --                 reset.reset_n
            sw_external_connection_export       => SW       --       sw_external_connection.export
        );
```

B. Programmation logicielle et test de l'IP Servomoteur

Maintenant, on doit afficher l'angle tourné par le servomoteur qui dépend de la position entrée dans le console en utilisant un code en c.

Dans ce cas, on utilise deux fonctions pour lire et écrire la valeur de l'angle et la position, ce sont *IORD_AVALON_ANGLE* et *IOWR_AVALON_ANGLE* que nous avons définie dans le fichier *servomoteur.hpp*.

```
#ifndef SERVOMOTEUR_HPP_
#define SERVOMOTEUR_HPP_

#define IORD_AVALON_ANGLE(base) (*((unsigned int*)(base)))
#define IOWD_AVALON_ANGLE(data,base) *((unsigned int*)(base)) = (data)


#endif /* SERVOMOTEUR_HPP_ */
```

Après, dans le fichier *servomoteur.cpp,* on utilise les deux fonctions. D'abord, on déclare la variable *position.* Puis, dans la boucle infini, on utilise la fonction *IOWR_AVALON_ANGLE* pour écrire la valeur position dans *AVALON_SERVOMO_0_BASE*. Ensuite pour chaque fois, on incrémente la position afin de mettre le servomoteur tourner petit à petit jusqu'à 180 degrés. A la fin, on utilise le *printf* pour afficher la valeur du *position.*
Malheureusement, la valeur de *position* est un peu bizarre.

```c
int main(){
    volatile unsigned int position;

    //volatile unsigned int *avalon_servomo_addr=NULL;
    printf("Commence la mesure\n");
    //avalon_servomo_addr = mmap(NULL,FPGA_REGS_SPAN, (PROT_READ | PROT_WRITE),MAP_SHARED, fd, HW_REGS_BASE);
    while(1){
        usleep(10000);
        IOWD_AVALON_ANGLE((position&0xFF),AVALON_SERVOMO_0_BASE);

        position ++;
        if (position > 180){
            position = 0;
        }
        printf("\POSITION = %d\n",position);
    }
}
```

V. Conclusion

En conclusion, à partir de ce mini-projet, on comprend bien la marche du capteur HC SR04 et le servomoteur. Savoir programmer le processus pour calculer la distance entre l'obstacle et le capteur à l'aide des signaux envoyés et reçus par le capteur. De plus, on sait comment commander la position du servomoteur en utilisant les interrupteurs externes(switch). On sait aussi comment implémenter le bus Avalon à partir du code en VHDL et le qsys dans la carte FPGA à base de Nios2. On a aussi appris d'utiliser eclipse pour écrire le code en c afin d' afficher les valeurs dans le console, dans cette partie, on a réussi d'afficher la valeur de distance mais pas les valeurs de position.