

Mini-Projet sur SoC-FPGA:

Capteur ultrason

YANG Liyun
JIN Qianhui

EISE4

Année: 2020-2021
Enseignement: M.Douze

Sommaire

I.	Introduction -----	3
II.	Réalisation d'IP -----	3
III.	Test de l'IP sur carte -----	7
IV.	Intégration de bus avalon -----	11
V.	Programmation logicielle et test de l'IP -----	15
VI.	Conclusion -----	18

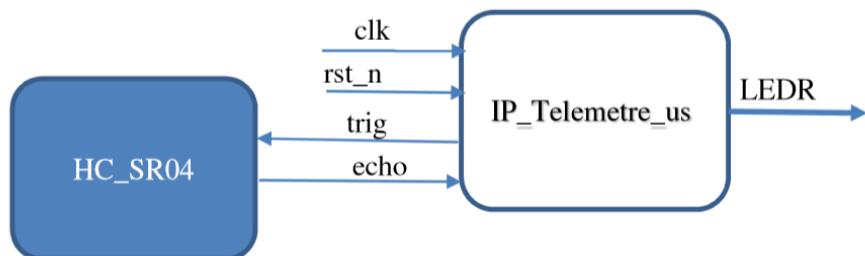
I. Introduction

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 avons deux mini-projets, ce projet IP télémètre est notre premier mini-projet. Le but de ce mini-projet est de détecter des obstacles à l'aide d'un télémètre ultrason et d'afficher la distance en centimètre sur LEDs de FPGA et sur le terminal. Afin d'atteindre cet objectif, nous devons réaliser un code en vhdl qui décrit la marche du IP télémètre ultrason, puis ajouter de bus avalon pour implémenter l'interface du IP télémètre avec le bus avalon dans la carte NIOS2 à l'aide de Qsys. A la fin, il faut réaliser un code en c qui permet de lire la valeur de distance et l'afficher dans le terminal.

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. Si 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.

II. Réalisation d'IP

a) Code d'IP sous vhdl



Selon le schéma au dessus, on a les port 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', si **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'incrémentera, **cpt_echo** est mis à zéro.

Lorsque le signal **Echo** est au niveau '0', **cpt_echo** et **dis** sont tous mis à 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 incrémantation, 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);
```

b) 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 instancié les signaux et crée d'abord le processus de l'horloge:

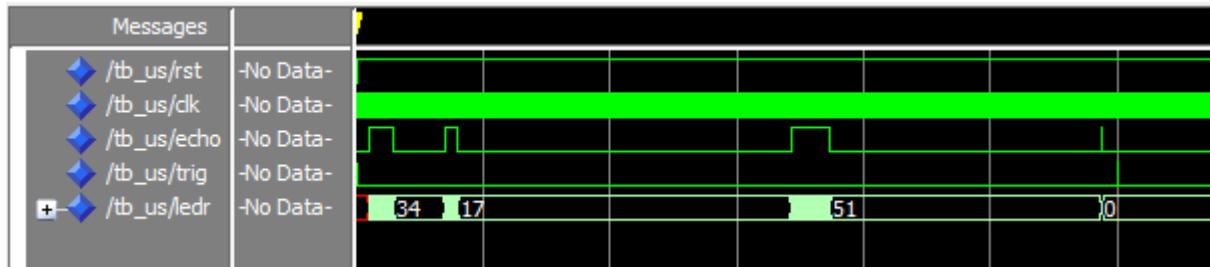
```
begin
--Instancier uut
uut : telemetre_us port map(
    clk =>clk,
    rst =>rst,
    echo =>echo,
    trig =>trig,
    LEDR =>LEDR
);
--clock process
clk_process : process
begin
    clk<= '0';
    wait for clk_periode/2;
    clk<= '1';
    wait for clk_periode/2;
end process;
```

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
    echo <= '0';
    wait for 4 ms;
    echo <= '1';
    wait for 1 ms; -- distance=34
    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;
```

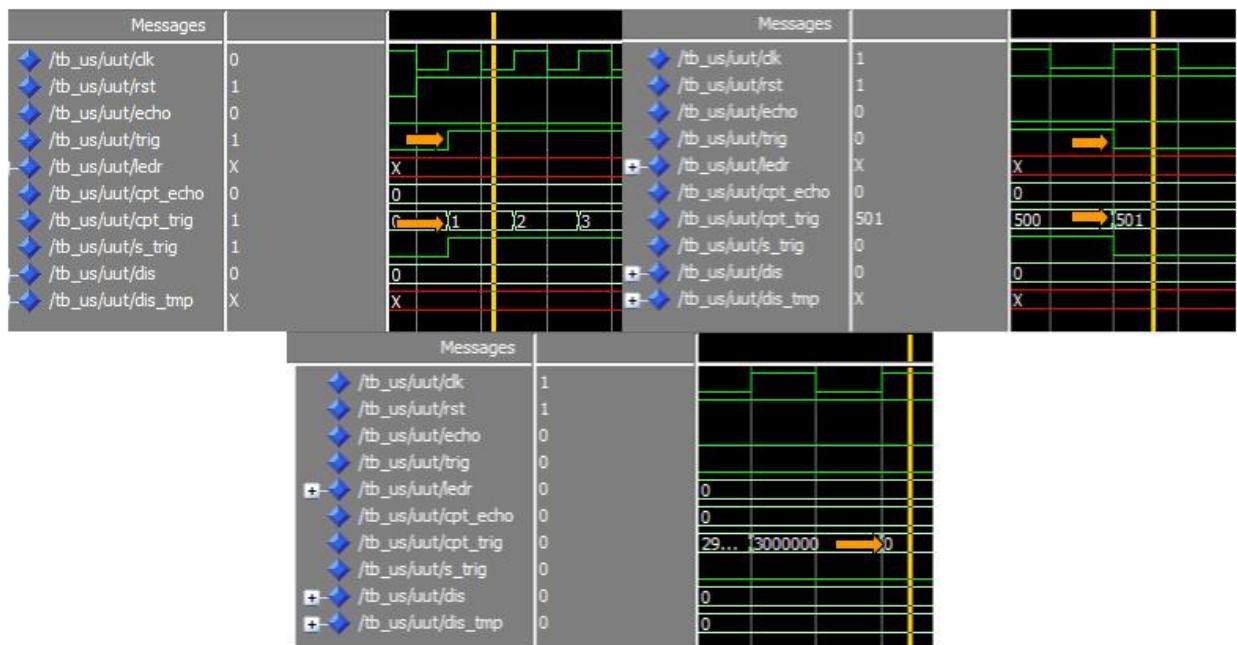
Ces signaux Echo sont dans la première période du signal **Trig**, On a la simulation dans la figure en dessous, et vous explique en détail.

La simulation globale:

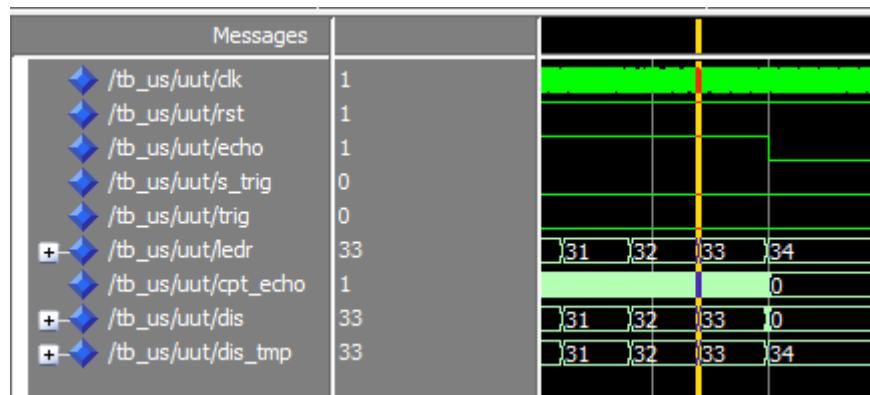


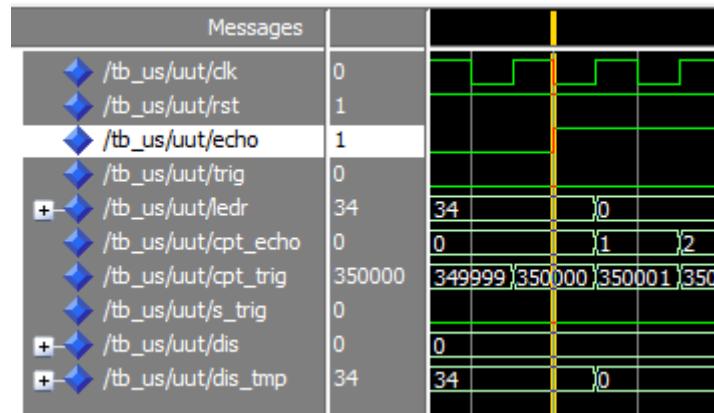
Le calcul déclenche bien dès que le signal **Echo** est au niveau “high”, **LEDR** a la valeur de la distance pendant la présence de **Echo** et retenir la valeur de distance jusqu'à un nouvel signal **Echo**. Or le dernier **Echo** est très petit (100 ns) la distance est d'environ 0.

Les détails:

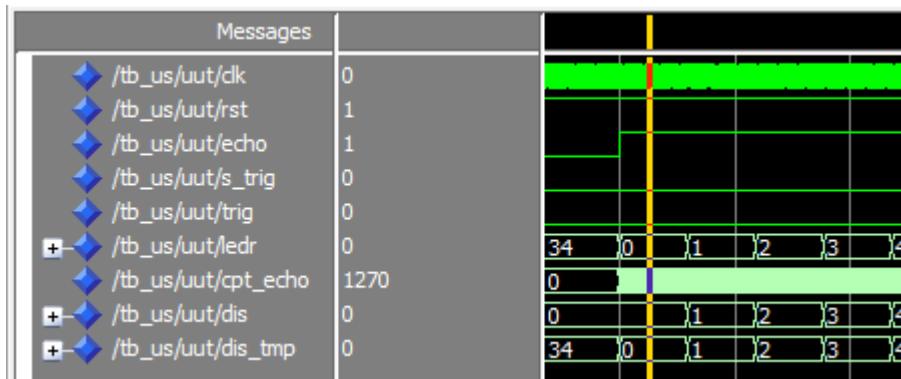


On voit bien que le signal **trig** finit bien lorsque **cpt_trig** est 500 (10 us), et commence bien si **cpt_trig** est 1

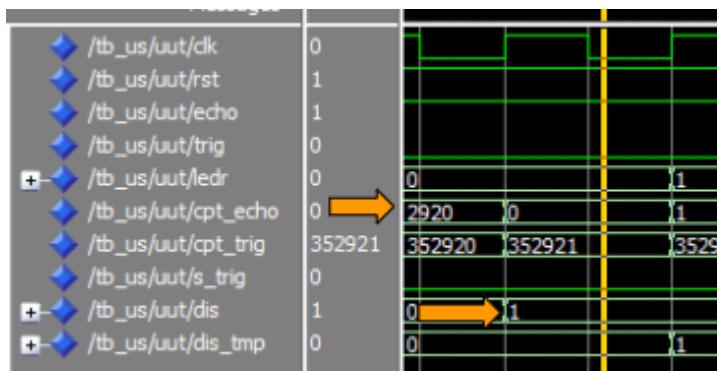




On voit bien que **cpt_echo** s'incrémente quand **Echo** est présent avec **clk** est à '1'. **cpt_echo** est aussi mise à zéro lorsque **Echo** est fini. Ainsi, **LEDR** maintient la valeur de distance s'il n'y a pas de signal **Echo**.



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 mis à zéro, dans ce cas la distance s'incrémente.

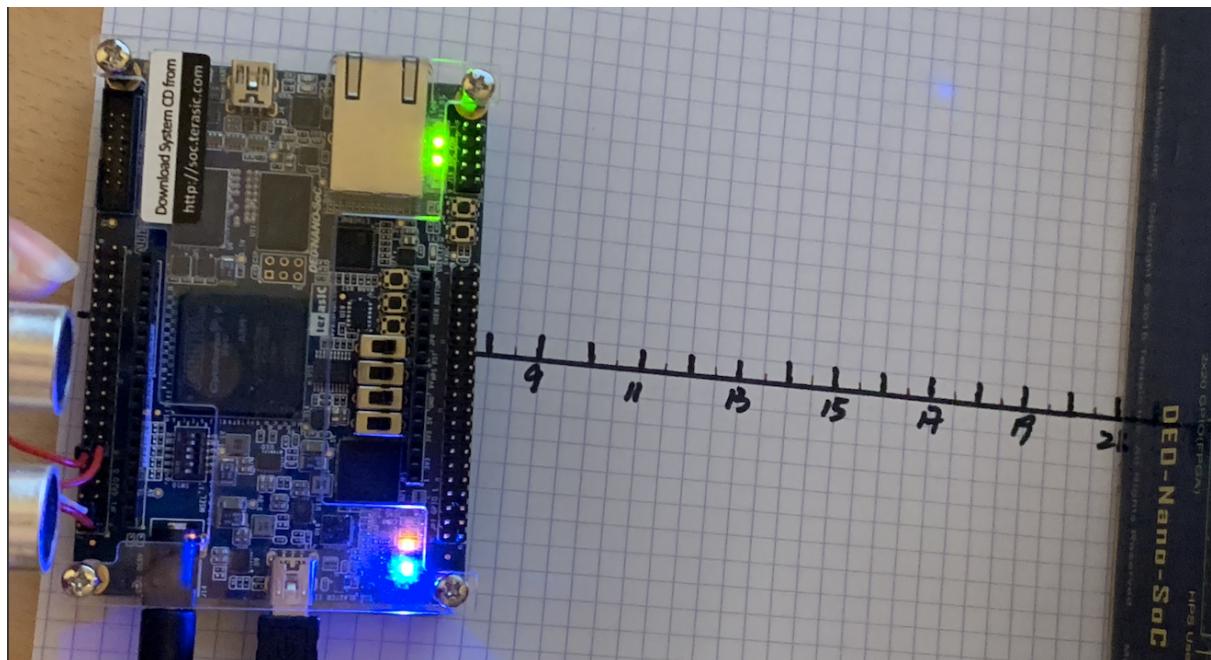
Messages		
◆ /tb_us/uut/dk	0	[Wavy green line]
◆ /tb_us/uut/rst	1	[Solid green line]
◆ /tb_us/uut/echo	0	[Solid green line]
◆ /tb_us/uut/s_trig	0	[Solid green line]
◆ /tb_us/uut/trig	0	[Solid green line]
+ ◆ /tb_us/uut/ledr	0	51 0
◆ /tb_us/uut/cpt_echo	5	0 [Binary sequence] 0
+ ◆ /tb_us/uut/dis	0	0
+ ◆ /tb_us/uut/dis_tm	0	51 0

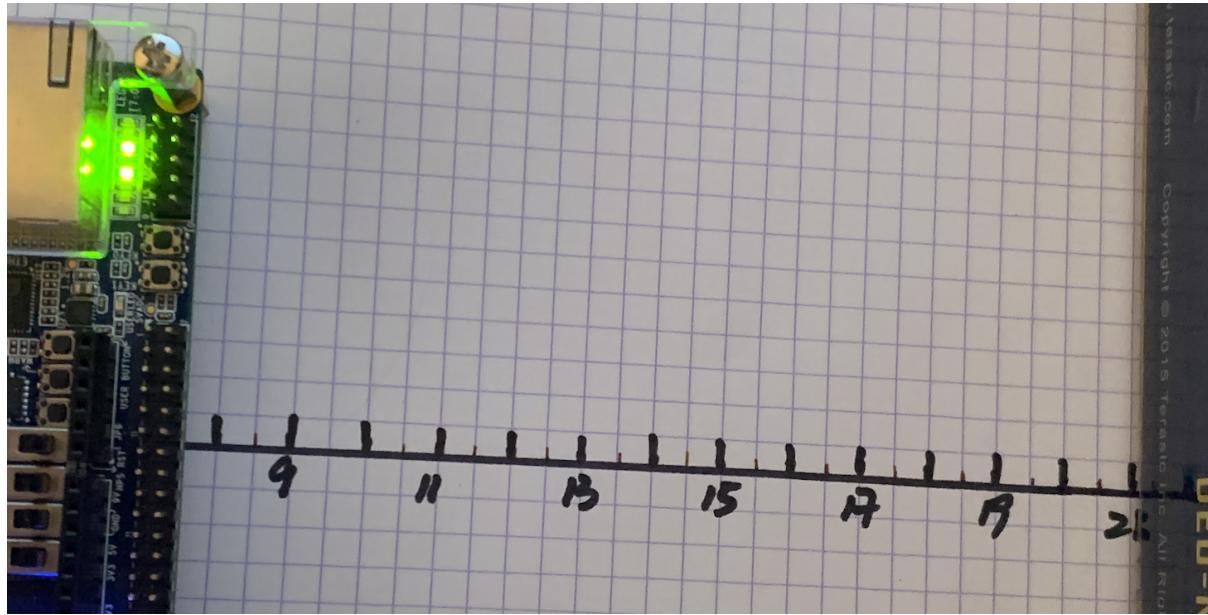
cpt_echo fonctionne même si **Echo** est petit (100 ans), mais il est mis à zéro aussi quand **Echo** est fini.

III. 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.

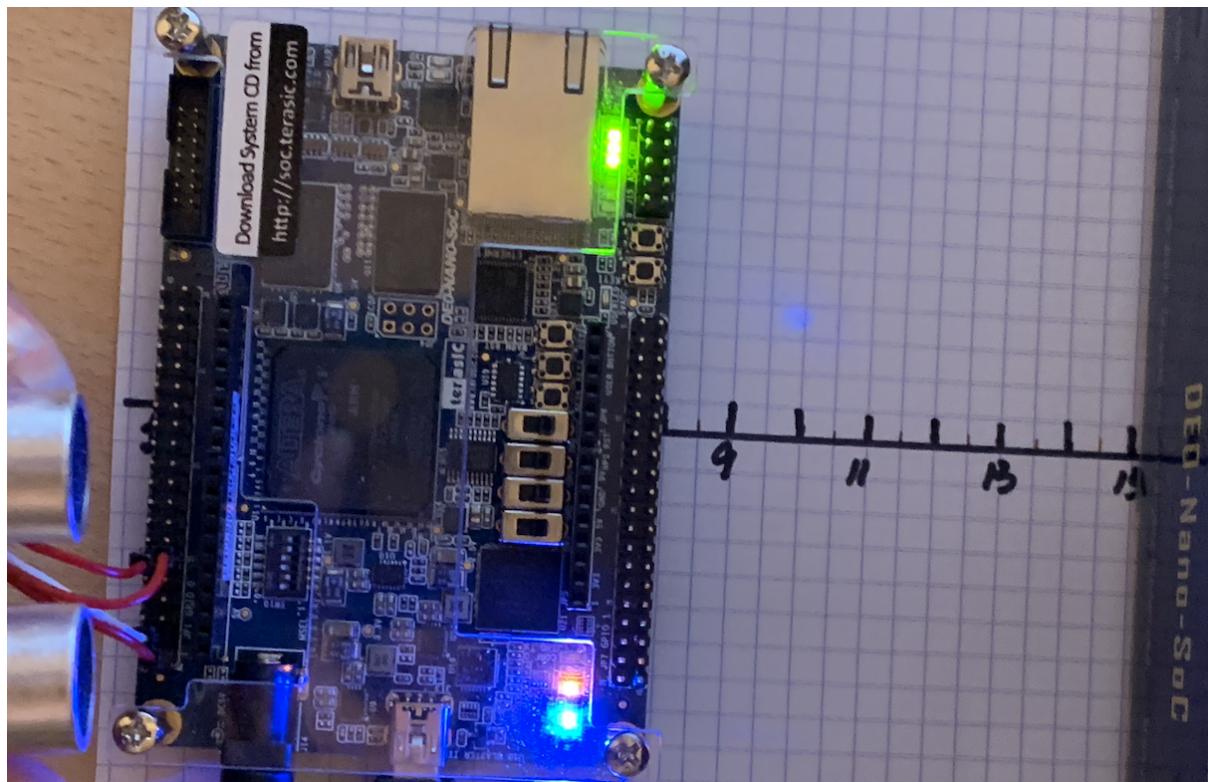
Voici les photos de nos tests :





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.

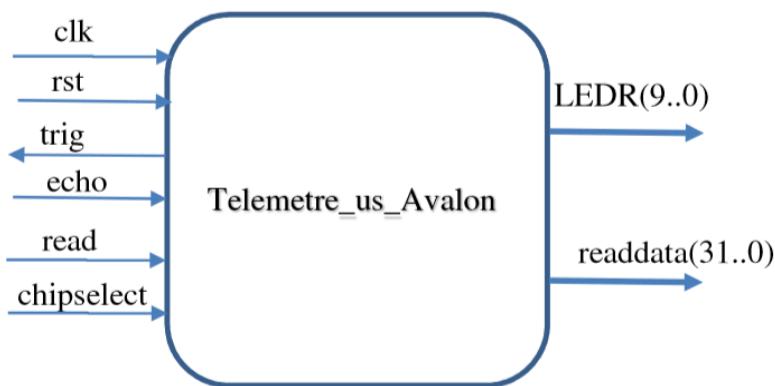
On fixe aussi le capteur à 1 cm, puis on rapproche la boîte au capteur avec une distance de 14 cm.



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.

IV. Intégration de Bus avalon

a) 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));
```

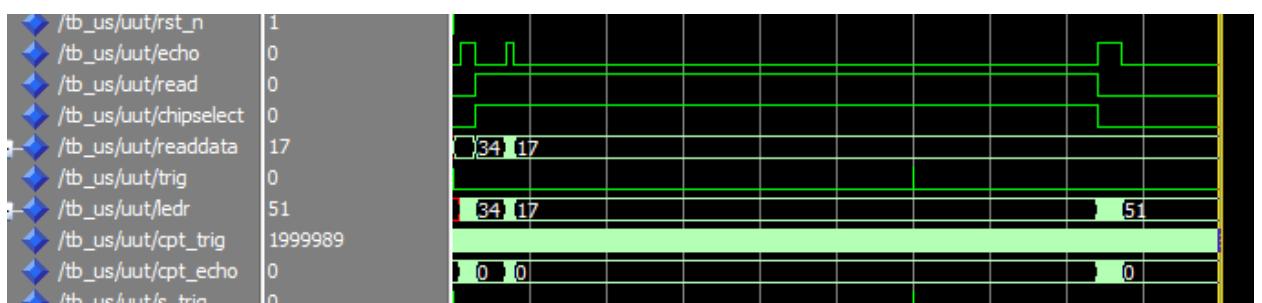
b) Test

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**.

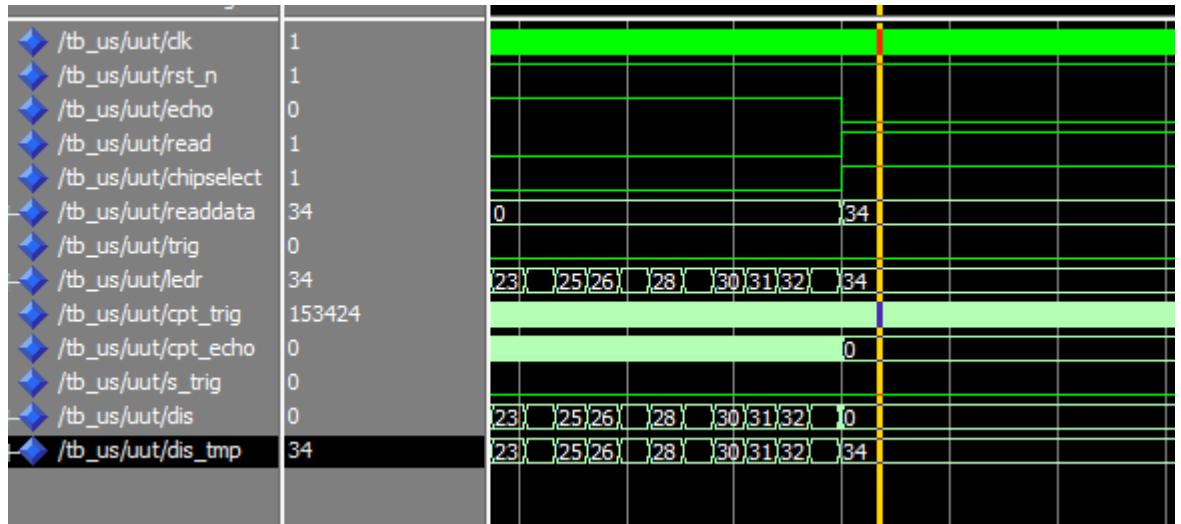
Voici le code pour la simulation:

```
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;
```

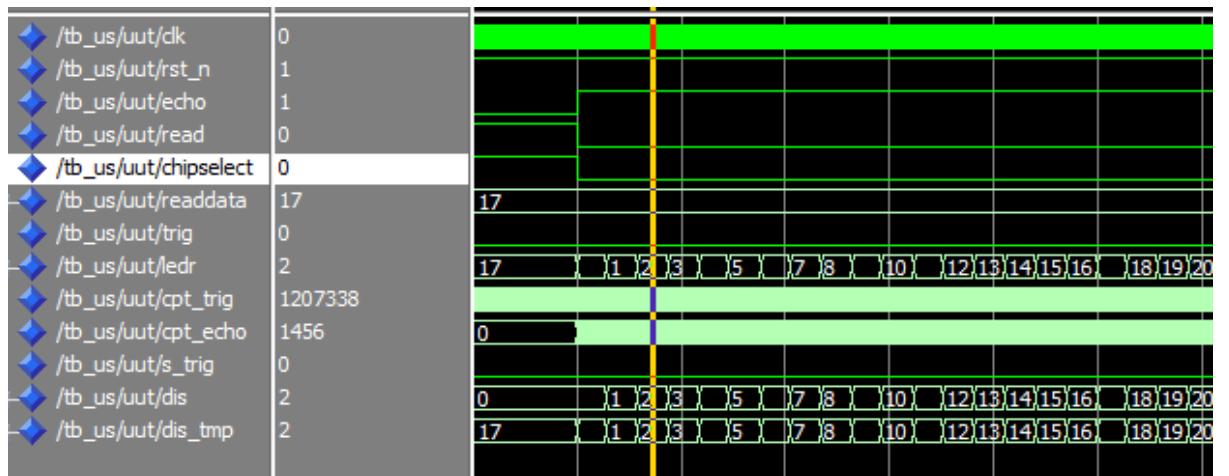
Voici la simulation global:



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***.



c) 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:

Name
► avalon_slave_0 Avalon Memory Mapped
└─ chipselect [1] chipselect
└─ read [1] read
└─ readdata [32] readdata
<<add signal>>
► clock Clock Input
└─ clk [1] clk
► led_pwm Conduit
└─ LEDR [8] writedata
└─ echo [1] beginbursttransfer
└─ trig [1] writeresponsevalid
<<add signal>>
► reset Reset Input
└─ rst_n [1] reset_n
<<add signal>>
<<add interface>>

Ensute 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_SDIN             => ADC_SDIN,
        adc_ltc2308_conduit_end_SDO              => ADC_SDO,
        clk_clk                                => FPGA_CLK1_50,
        --p11_sys_locked_export                 => CONNECTED_TO_p11_sys,
        --p11_sys_outclk2_clk                  => CONNECTED_TO_p11_sys,
        reset_reset_n                           => KEY(0),
        sw_external_connection_export           => SW,           -- SI
        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)  -- --
    );

```

V. Programmation logicielle et test de l'IP

a) Code en c

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

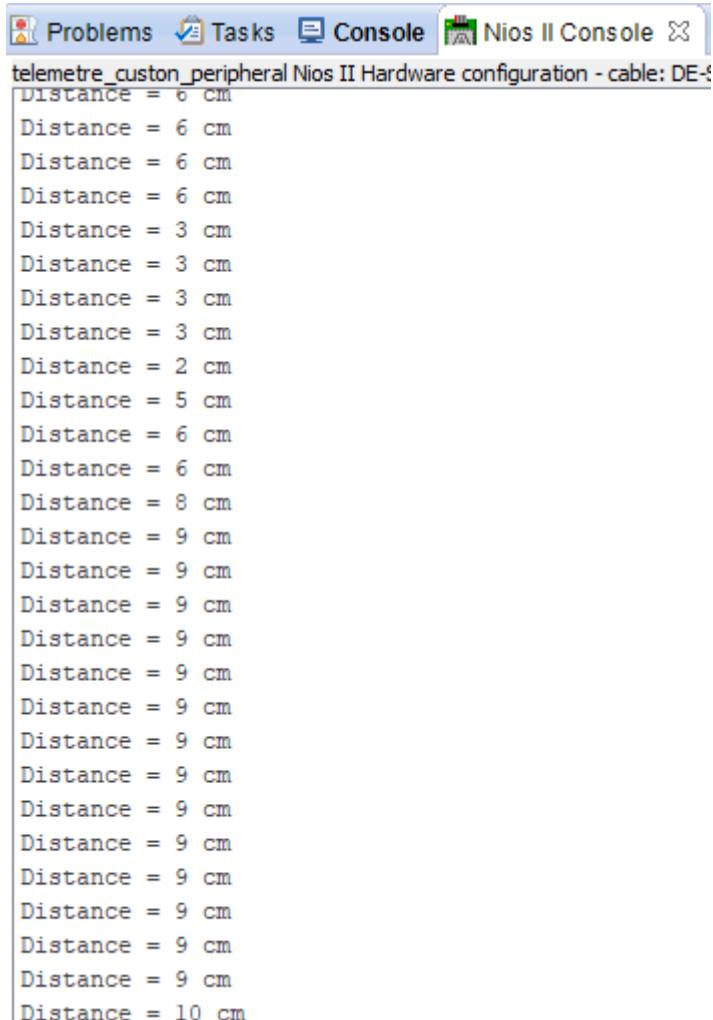
Alors, pour lire et écrire la valeur de distance, on a besoin deux fonctions, ce sont *IORD_AVALON_DISTANCE* et *IOWR_AVALON_DISTANCE* que nous avons définie dans le fichier *avalon_telemetre_mesure.h*. La fonction *IORD_AVALON_DISTANCE* permet de lire la valeur de *base*, et la fonction *IOWR_AVALON_DISTANCE*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

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

Ensute, 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*.

```
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);
}
```

b) Simulation en comparant la valeur afficher et la valeur lire sur la carte FPGA

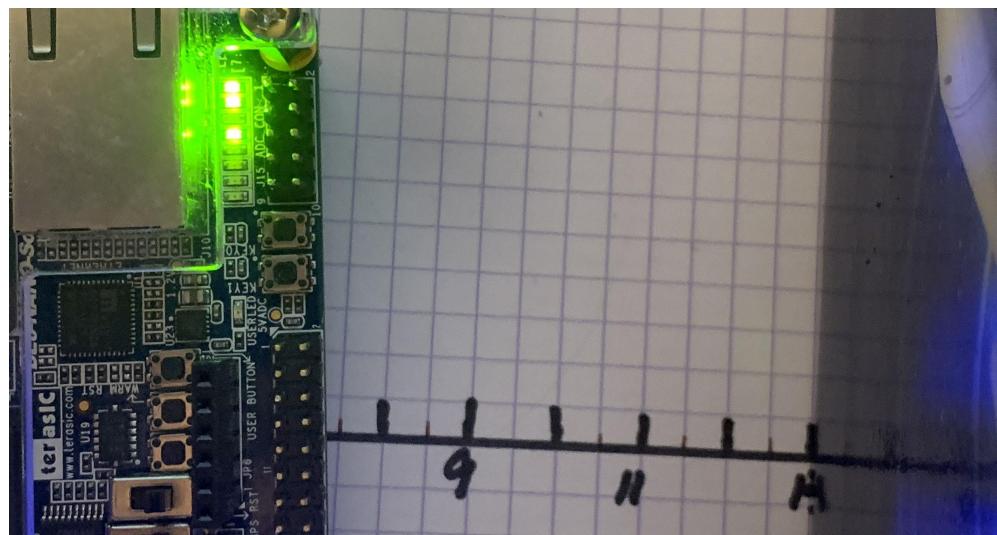
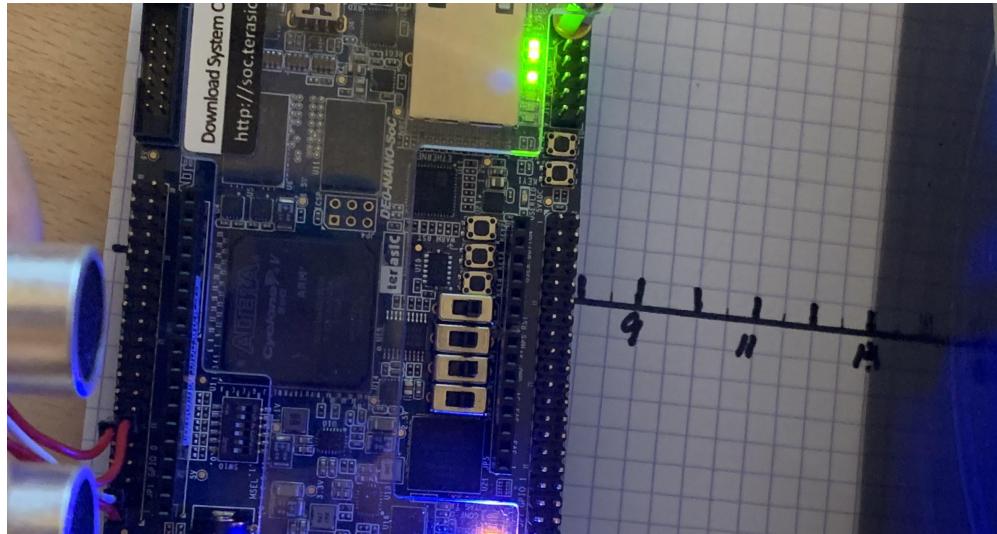


The screenshot shows a terminal window titled "Nios II Console". The title bar also includes "Problems", "Tasks", and "Console". The main area of the window displays a series of text lines: "telemetre_custom_peripheral Nios II Hardware configuration - cable: DE-4", followed by multiple lines of "Distance = 6 cm", then "Distance = 3 cm", and finally a long sequence of "Distance = 9 cm" followed by one "Distance = 10 cm".

```
telemetre_custom_peripheral Nios II Hardware configuration - cable: DE-4
Distance = 6 cm
Distance = 6 cm
Distance = 6 cm
Distance = 3 cm
Distance = 3 cm
Distance = 3 cm
Distance = 3 cm
Distance = 2 cm
Distance = 5 cm
Distance = 6 cm
Distance = 6 cm
Distance = 8 cm
Distance = 9 cm
Distance = 10 cm
```

La figure à côté 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:



```
Problems Tasks
telemetre_custom_peripheral
Distance = 11 cm
```

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 fournir aussi une démonstration en vidéo:

<https://drive.google.com/file/d/1Mj8LQgENV5PU0d4GaQvkifL-fFs6yTy6/view?usp=sharing>

VI. Conclusion

A partir de ce mini-projet, on comprend bien la marche du capteur HC SR04, et comment programmer le processus pour calculer la distance entre l'obstacle et le capteur à l'aide des signaux envoyés et reçus par le capteur. En plus, on sait comment implémenter le bus avalon à partir du code en vhdl et le qsys dans la carte FPGA à base de Nios2. Puis, on a aussi étudié à utiliser eclipse en écrivant un code en c afin d'afficher la distance dans le console.