UMONS
Université de Mons

Faculté Polytechnique

Hardware and Software

DE1-SoC : Servomotor control

Idriss FATTAHI
Valentin HONOREZ
Group: G2-DE1-Servo

POLYTECH MONS

Under the supervision of the Professor
Carlos VALDERRAMA

Academic Year 2019-2020

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

Contents

Introduction

This project is part of the course Hardware/Software Platforms at the polytechnic faculty of Mons. The objective of this report is to show to the readers a way to control a servo-motor with the DE1-Soc board.

The interest of this project is to apply known techniques and looking for existing solutions because people in real life will also prefer to keep their comfort zone too. We are not here to reinvent the wheel to create a car but to show to the readers than we can handle an electronical project from start to end.

It is important to report than the entire duration of the project happened during the sanitary crisis linked to Covid-19, so were unable to test our codes on real equipment and do a tutorial of how to do it.

All our codes are available on GitHub[1], even if it wasn't asked to put our codes on it, we would have done it. Eventually, nowadays, GitHub is our resume to be hired in a company so it's really important to demonstrate the skills that we have learned.

[1]https: //github.com/ValentinHonorez/HardwareSoftwarePlatformsServo

1 Project Goal

The main goal of the project is to be able to change the position of a servomotor by controlling the switches on the DE1-SoC board.

To achieve that, we must follows some objectives which are the following :

1. Collect information needed to develop an application and evaluate the existing component by doing research on technical documents.
2. Understand technical documents and pull out the right information.
3. Improve our VHDL programming language skills by coding algorithms which are necessary for our project.
4. Understand the behaviour and the implementation of already existing work
5. Develop a hardware driver in VHDL to configure and interact with the peripheral
6. Implement a test bench of the hardware driver to study its behaviour.
7. Develop an application in VHDL for interacting with the environment and use the driver to get/receive data from the peripheral.
8. Implement the test bench of the application to virtually simulate its behaviour and interaction with the environment.

The Hardware driver and the application will be explained in details in the chapter 5.4 because it's the core of our codes.

2 Theoretical background

2.1 Servomotor

A servomotor is a motorized system able to reach predetermined positions and to maintain them if needed. The servomotor is composed at the same time of the motor, gears and all the electronics needed for controlling the motor. This kind of motor is controlled by a PWM signal.

2.2 PWM signal

A PWM signal is a binary signal which can be interpreted as an analogical value. The principle is to create a signal with 0 and 1 values and having a fixed period. This signal will be characterized with a certain duty cycle, which is the fraction of one period in which the signal is active (equal to one). The mean value of this signal will be an analog quantity equal to the duty cycle multiplied by the maximum amplitude of the signal. The duty cycle is given by this formula :

$$DC = \frac{T_a}{T}$$

with T the total period and T_a the time during which the signal is equal to one. Figure 2.1 shows us an illustration :

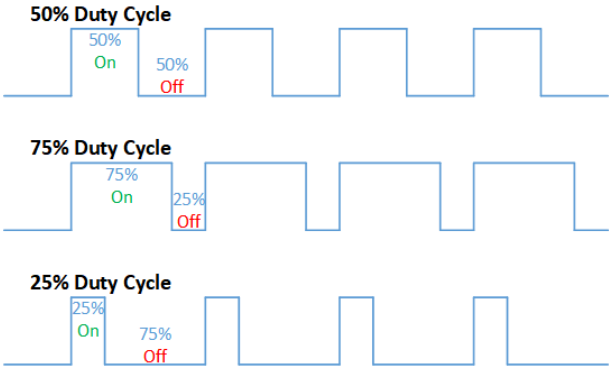

Figure 2.1: Duty cycle

3 Tools used

The name of the course "Hardware/Software platforms" really describes this part. It means that anything can be implemented in different ways. We just have to choose the language, the development tool and the targeted microprocessor.

3.1 Programming language

The programming language used in this project is VHDL (Very-High-Speed Integrated Circuits). It is a hardware description language used in electronic design automation to visualize and describe digital/analog/mixed-signal systems. The language has a high level of abstraction and have some modelling styles (Dataflow, structural , behavioral) based on the type of concurrent statement used.

One of the strong point of VHDL resides in its executable nature: a specification described in VHDL can be verified by simulation, before the design in real life is completed. (gates and wires). It is also independent of the technology used so it simulate the behaviour on computer without even having programmed the processor. Another advantage is that VHDL is a typed language.

3.2 Software

The development tool for this project is Altera Quartus II by Intel. its includes a VHDL implementation, a visual editing of logic circuits. It's a big tool to design FPGAs. To simulate our VHDL codes , ModelSim is also needed. It provides a complete simulation and debugging environment for complex designs in FPGA and it supports several description languages, including VHDL. For this project ModelSim will be used in conjunction with Intel Quartus and not independently.

3.3 Hardware

Normally, for our project, we should have used the DE1-Soc board to implement the application.

4 Tutorial

To easily reproduce all the results obtained to the project and/or to start on the basis of our work, a tutorial is proposed in this chapter. As mentioned before, due to the sanitary crisis linked to Covid-19, we were incapable to implement on the DE1-Soc board. Hence, the tutorial will focus on Quartus II for the following aspects: the installations, the Project Creation and how to simulate the codes.

4.1 Installations

To install Quartus, the following steps are required:

1. Create an Intel account
2. Select the Lite edition and the 18.1 release
3. Download the Quartus Prime and ModelSim-Intel FPGA Edition. (figure 4.1)
4. Download Cyclone V device support
5. Launch the Device installer on your desktop and put the cyclone V file in the download directory.

Note: Do not put spaces in directory names, this can cause problems.

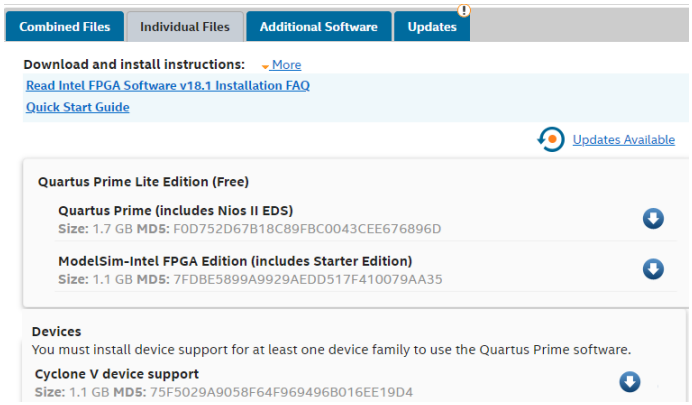

Figure 4.1: Download

4.2 Project Creation

To create a new project on Quartus, the following steps need to be followed:

1. Launch Quartus II
2. Click on file : "New Project Wizard..."
3. On the new window displayed (fig 4.2):
   (a) Write the correct folder and name of the project.
   (b) Click on "Next"
   (c) Click on "Empty project"
4. Click "Next" on the "Add files" window
5. On the "Family, Device and Board Settings" window (fig 4.3), select the right family of component and then “Next”. The component used in this project is a Cyclone V 5CGXFC7C6F23I7.
6. On the "EDA Tool Settings", select "ModelSim-Altera" as Tool name and "VHDL" as format. (fig 4.4)

The project is created, you have to add files to it. There are two choices :

1. Create a new file by selecting "New", "File" and "VHDL File". (fig 4.5)
2. Add an already created file by selecting again "New project Wizard" and add the name of the existing file you want to add.

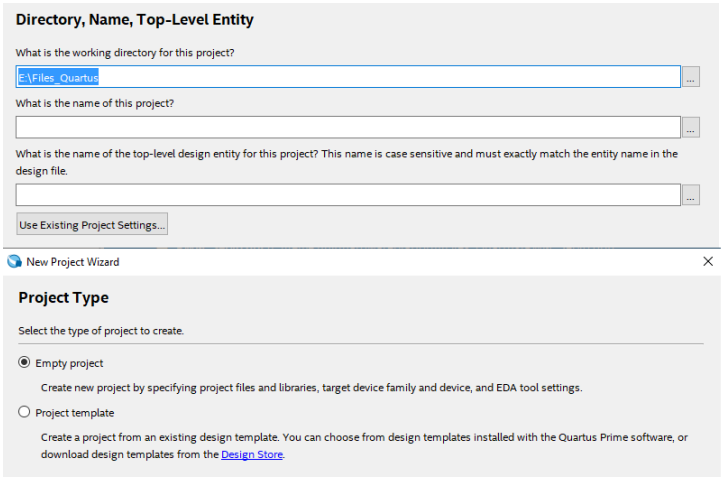

Figure 4.2: Directory, Name, Top-Level Entity

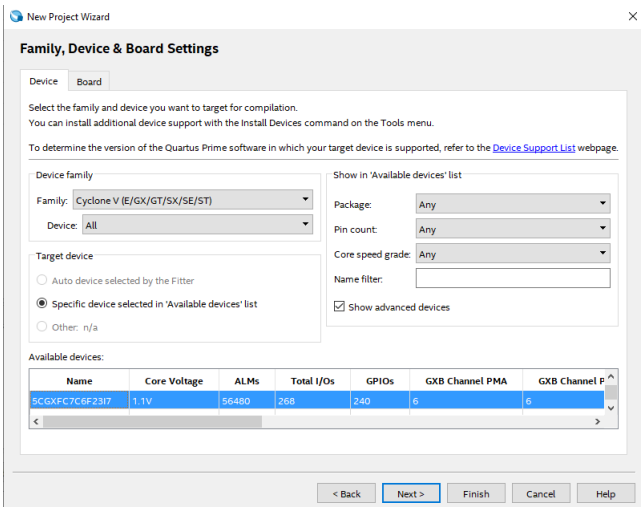

Figure 4.3: Family, Device and Board Settings

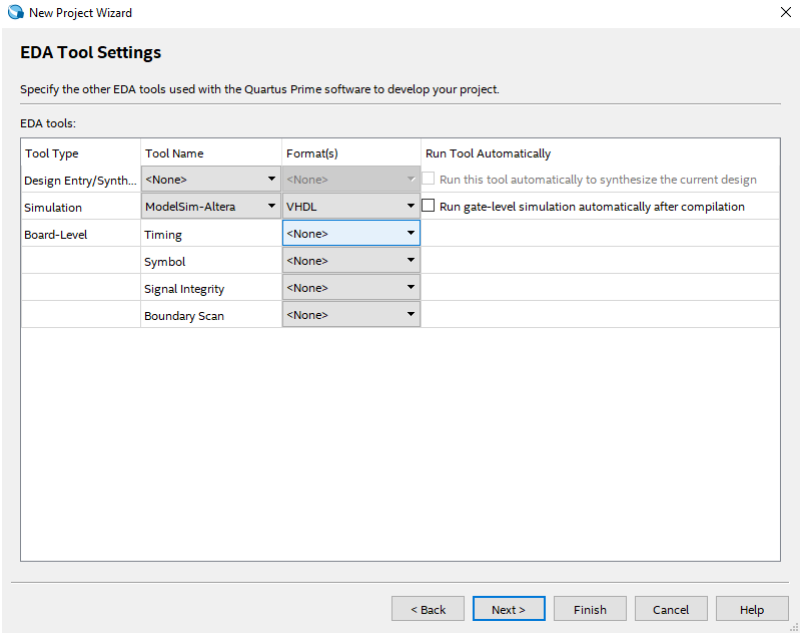

Figure 4.4: EDA Tool Settings

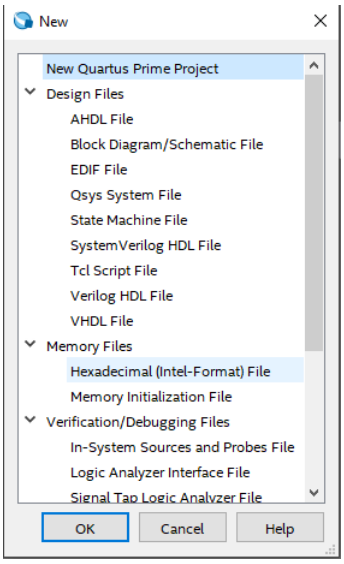

Figure 4.5: New file

4.3 How to Simulate

To simulate the VHDL files on Quartus, the following steps need to be done:

1. Set the main code to the top-level entity by clicking right on the file (fig 4.6)
2. Click right on the test bench file, select properties and change the type file in "VHDL Test Bench File". (fig 4.7)
3. Click on Assignments, EDA Tool Settings, Simulation and modify the output directory by writing the modelsim directory. (fig 4.8)
4. In the same panel, click on Test Benches:
   (a) Create a new Test bench
   (b) Copy the name of the VHDL file and paste it on the Test bench name.
   (c) Paste it also on the File name and press add.
5. Click on Processing ans start the compilation
6. Click on Tools, Run Simulation Tool, RTL simulation

After these steps, you will have an interface to simulate the test bench. Just press stop to stop the simulation and analyze it ! (figure 4.9)

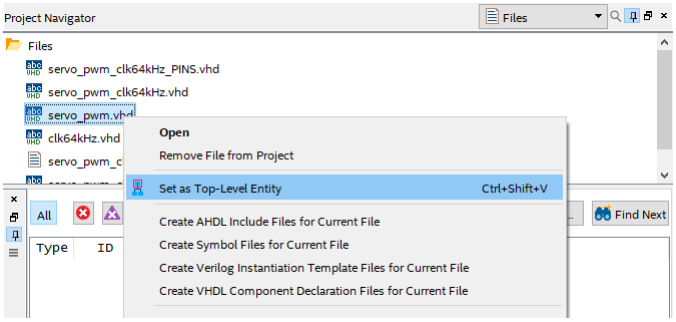

Figure 4.6: Top-level entity

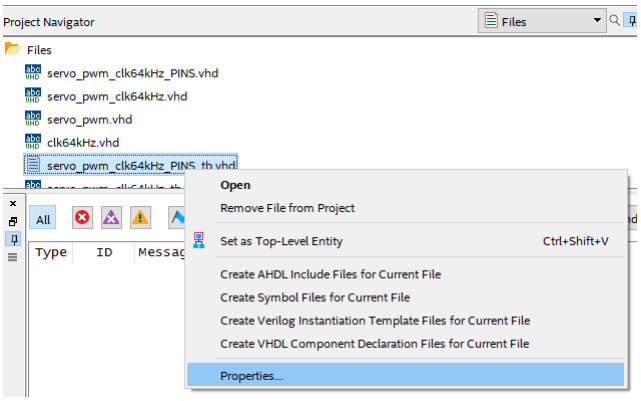

Figure 4.7: VHDL Test Bench File

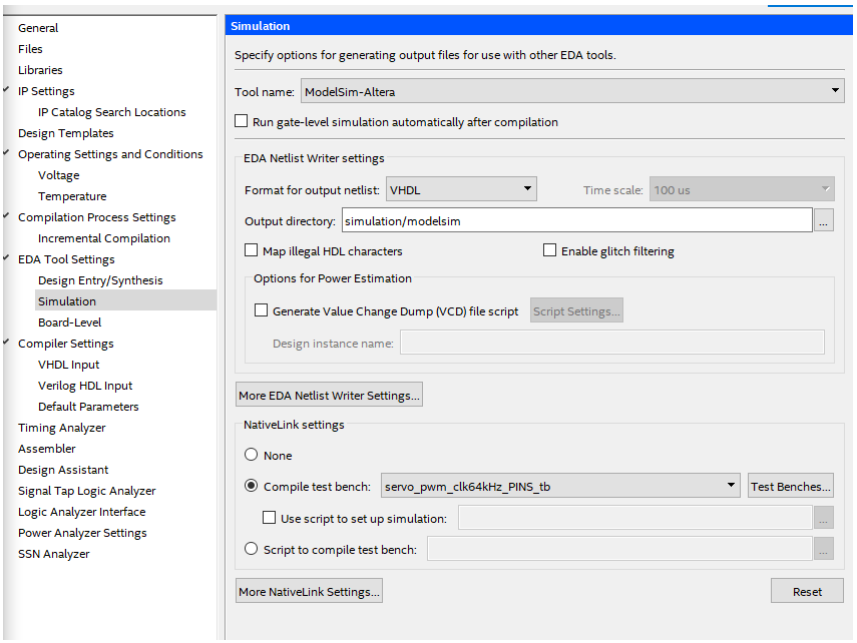

Figure 4.8: EDA Tool Settings

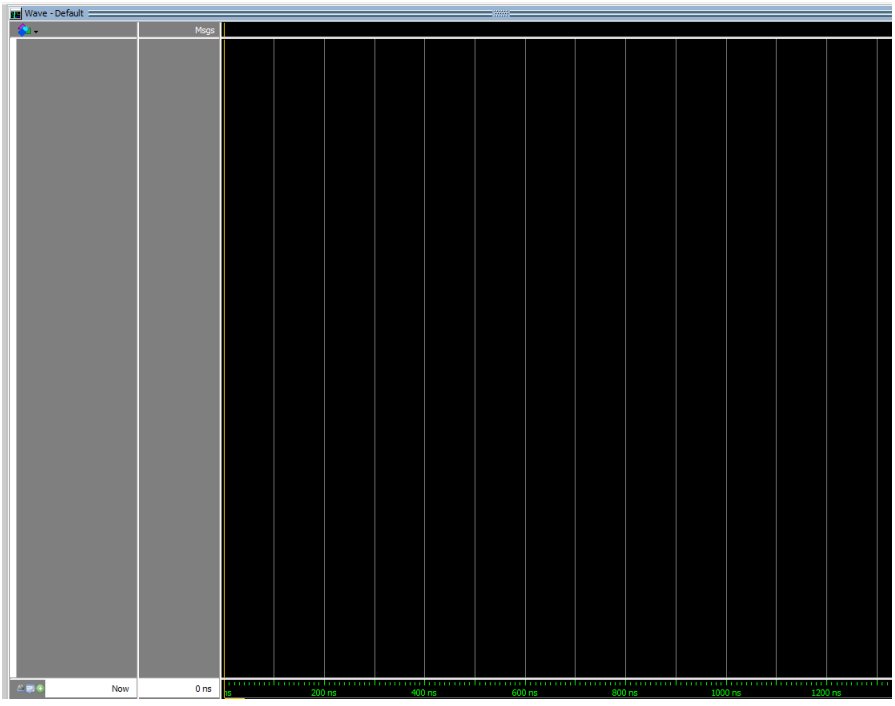

Figure 4.9: Simulations interface

5 VHDL codes explanations

5.1 Frequency divider

In the first place, we have to synthesise a frequency divider. Indeed, the altera board that we have in our possession can generate a clock of 100 MHz but we need much a lower frequency. The clock that we want to generate is a 64 kHz frequency clock. In fact, the pulse width range of our PWM goes from 0.5 ms to 2.5ms. Thus, the range of operation is 2ms. Furthermore, our servomotor can take 128 positions. So, by dividing the number of positions by the range of operation, we obtain our 64 kHz clock.

In this code (see Figure 6.1), we have as inputs the original clock and the reset, and as output the new clock. We have also created a temporal signal who will serve as an intermediary to create the new clock and a counter which goes from 0 to 780. When the clock state changes, we increment the counter by one. Once the counter reaches 780, we change the state of our temporal signal and we reset the counter at 0 value.

5.2 Control of the servo with PWM

As we said in previous sections, we have to vary the width of the pulse in order to vary the angle of the servomotor. We find a kind of bijection between the pulse width and the desired position.

In this code (see Figure 6.2 and 6.3), as inputs, we have the clock, the reset and the desired position which is a vector of 7 bits (128 positions). As output, we have the servo signal, we will serve to control the servomotor. A counter of 11 bits will be generated to allow to count till 1279. We obtain this value because we want to know how many samples we need with our 64KHz clock to browse a period. Indeed, we have a clock of 64KHz, so we will have 64 samples every 1ms. Knowing that the time between two pulses is 20ms, we have 64*20 samples during on period, which is equivalent to 1280 samples. We will also use a temporal signal (pwmi) of 8 bits which is used to generate the PWM pulse. To create the servo signal, we compare the counter value with the temporal signal value. When the counter is smaller than the pwmi, then put the servo value to 1, otherwise you put it to 0. As said, we want as minimum value of the servo signal to be 0.5 ms. We get this value to

0.5 ms by putting the minimum value of the pwmi signal to 32. Indeed, if we put the pwmi to 32, we can easily get the value of 0.5ms knowing that if we count till 1280, a period of 20ms is elapsed :

$$\frac{32}{1280} = \frac{x}{20ms} \Longleftrightarrow x = 0.5ms$$

In the same way, we can extract the maximum value which is :

$$\frac{32 + 128}{1280} = \frac{x}{20ms} \Longleftrightarrow x = 2.5ms$$

Our servo motor goes then from 0.5ms to 2.5ms of pulse width. When the counter reaches 1280, we have to reset it to 0 to start a new period.

5.3 Main function

The main function will assemble the clock that we have generated with the pwm signal to control the servo. The codes are available in Figure 6.3, 6.4 and 6.5.

5.4 Interface

Finally, we have to connect our signals with the board. As we can see in the code (Figure 6.6 and 6.7), we have several board inputs. We can distinguish :

1. clock input : Accept the clock of 64kHz
2. KEY[0 to 3] : Corresponds to the buttons on the board.
3. SW[0 to 7] : Correspond to the switches on the board.
4. LEDR [0 to 9] : Corresponds to the leds on the board
5. GPIO [0 to 3] : Corresponds to the input/outputs of the board

The GPIO(0) is used to connect the ground, GPIO(1) is used to connect the servo signal and GPIO(2) is used to connect the DC supply (5V).

The SW vector will be used to indicate the position that we want in binary. For example, if we want the servomotor to stay at 0 degrees, we will put all the switches on 0.

The LEDR vector is used to show graphically which position is allocated. If we look at the example taken for the SW vector, all the LEDS would be down.

The KEY input will be used to activate the process. If we press on the button 1 (KEY(1)), then we change the position of the angle according to the states of the switches. Otherwise, if we press on the button 0 (KEY(0)), then we reset the process.

6 Simulations

The simulation that you will see in this section is divided in two parts. First, we we will simulate the generation of the servo signal for different desired positions. Second, we will simulate the interface human/machine. In other words, we will simulate the way that we will interact with the servomotor. In this, we will change the positions of the switches present on the board.

6.1 Clock generation

To verify the good generation of our signal of the signal, we have a made a simulation for that :

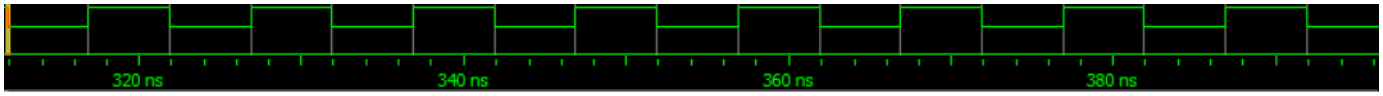

Figure 6.1: 100 MHz Altera Board clock

We can see that our original signal has a period of 10 ns. The frequency is thus of 100 MHz. This signal will pass through the frequency divider to produce a clock signal of 64 kHz.

6.2 Servo signal generation for different positions

To test the good generation of our servo signal, we will test is for the 2 extremes positions and one position between them. First, for a desired position of 0 degrees, the position in binary is '0000000'. This position equals to a servo signal of 0.5 ms of pulse width (see section 5.2) and that is what we can observe in the following figure :

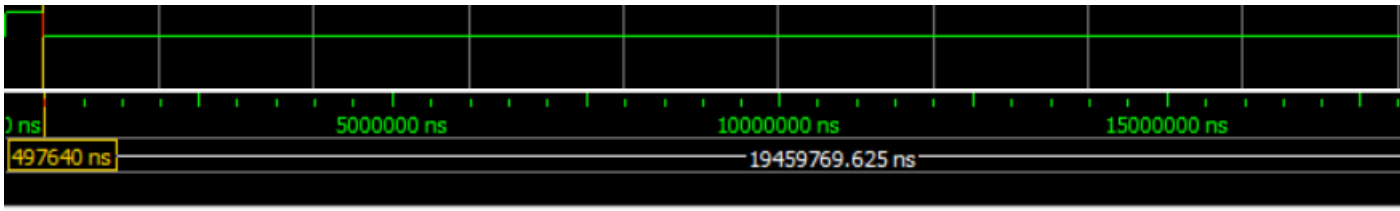

Figure 6.2: Servo signal for a position at 0 degrees

You can also see that the total period is 20 ms, which confirms what we already told in the previous sections. If we want our servomotor to make a full turn (360 degrees), we have to apply to the servomotor a signal of 2.5 ms (see section 5.3 for the computations). This value can be observed in the next figure :

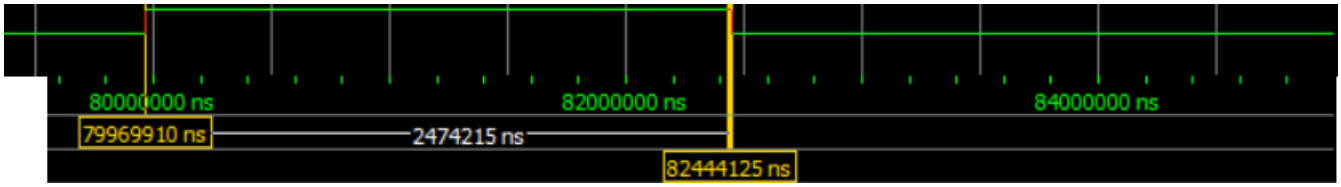

Figure 6.3: Servo signal for a position at 360 degrees

Finally, we will take a value between 0 degrees and 360 degrees. We have chosen the position "0101000" which is the 40th position. We know that there is 128 position for 1 round, that means that the 40th position corresponds to :

$$\frac{40}{128} = \frac{position(^\circ)}{360} \iff position(^\circ) = \frac{40}{128} \cdot 360 = 112.5^\circ$$

We also know that the servo time is, using the same reasoning than in the section 5.3 :

$$\frac{32+40}{1280} = \frac{x}{20ms} \iff x = 1.10ms$$

We can observe this in the following figure :

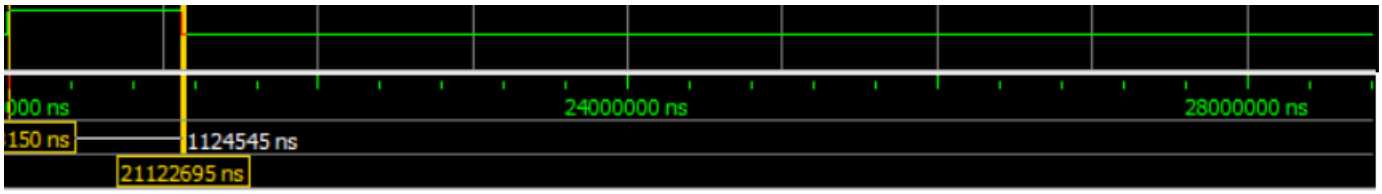

Figure 6.4: Servo signal for a position at 112.5 degrees

6.3 Interface

For the interface, we will make a simple test : we will first change the values of the switches. After that, we will wait 3 x PWM half period and then we press the button. Then, we will wait 1 additional half period to release the button. When this button is released, we change the value of the switch. The PWM value is then modified.

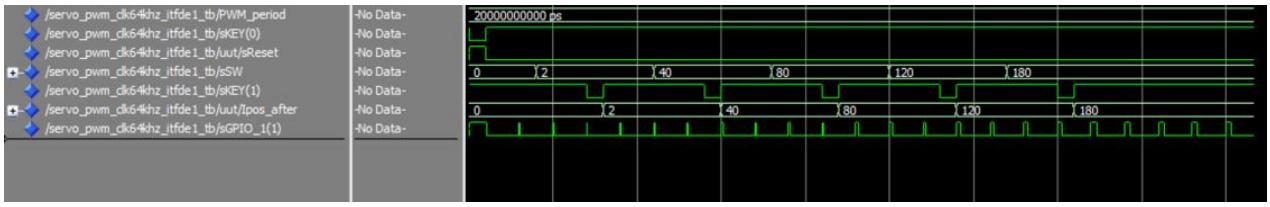

Figure 6.5: Behaviour of the interface

7 Conclusion

Despite current sanitary conditions which didn't unable us to test our codes on a board and with real conditions, this project allows us to deal entirely with an electronical project.

As I said in the chapter 3, the name of the course means than anything can be implemented but it depends of multiples variables like the language, the development tool, the hardware used, etc.. A big advantage that we can get from that is that our work can be translated by other people to another language like Python, java, an other development tool like IDE Arduino, another microprocessor and so on. They will lose less time to understand what they have to pay attention to handle this work.

We hope that all this work will help the readers to control a servomotor in the real life.

Annexes

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

-- Construction of the clock of 64KHz
entity clk64kHz is
    Port (
        clk    : in  STD_LOGIC;
        reset  : in  STD_LOGIC;
        clk_out: out STD_LOGIC
    );
end clk64kHz;

architecture Behavioral of clk64kHz is
    signal temporal: STD_LOGIC;
    signal counter : integer range 0 to 780 := 0;
begin
    freq_divider: process (reset, clk) begin
        if (reset = '1') then
            temporal <= '0';
            counter  <= 0;
        elsif rising_edge(clk) then
            if (counter = 780) then
                temporal <= NOT(temporal);
                counter  <= 0;
            else
                counter <= counter + 1;
            end if;
        end if;
    end process;

    clk_out <= temporal;
end Behavioral;

```

Figure 7.1: Frequency divider VHDL code

```vhdl
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

-- PWM signal to control the servo

entity servo_pwm is
    PORT (
        clk   : IN  STD_LOGIC;
        reset : IN  STD_LOGIC;
        pos   : IN  STD_LOGIC_VECTOR(6 downto 0);
        servo : OUT STD_LOGIC
    );
end servo_pwm;


architecture Behavioral of servo_pwm is
    -- Counter, from 0 to 1279.
    signal cnt : unsigned(10 downto 0);
    -- Temporal signal used to generate the PWM pulse.
    signal pwmi: unsigned(7 downto 0);
```

Figure 7.2: Control of the servo with PWMr VHDL code : Part 1

```vhdl
begin
    -- Minimum value should be 0.5ms.
    pwmi <= unsigned('0' & pos) + 32;
    -- Counter process, from 0 to 1279.
    counter: process (reset, clk) begin
        if (reset = '1') then
            cnt <= (others => '0');
        elsif rising_edge(clk) then
            if (cnt = 1279) then
                cnt <= (others => '0');
            else
                cnt <= cnt + 1;
            end if;
        end if;
    end process;
    -- Output signal for the servomotor.
    servo <= '1' when (cnt < pwmi) else '0';
end Behavioral;
```

Figure 7.3: Control of the servo with PWM VHDL code : Part 2

```
1  library IEEE;
2
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  use IEEE.NUMERIC_STD.ALL;
6
7
8
9
10 -- assemble the clock and the pwm to control the servo
11
12
13
14
15 entity servo_pwm_clk64kHz is
16
17     PORT(
18
19         clk  : IN  STD_LOGIC;
20
21         reset: IN  STD_LOGIC;
22
23         pos  : IN  STD_LOGIC_VECTOR(6 downto 0);
24
25         servo: OUT STD_LOGIC
26
27     );
28
29 end servo_pwm_clk64kHz;
30
31
32
33 architecture Behavioral of servo_pwm_clk64kHz is
34
35     COMPONENT clk64kHz
36
37         PORT(
38
39             clk    : in  STD_LOGIC;
40
41             reset  : in  STD_LOGIC;
42
43             clk_out: out STD_LOGIC
44
45         );
46
47     END COMPONENT;
48
```

Figure 7.4: Main function : Part 1

```
    COMPONENT servo_pwm
        PORT (
            clk   : IN  STD_LOGIC;
            reset : IN  STD_LOGIC;
            pos   : IN  STD_LOGIC_VECTOR(6 downto 0);
            servo : OUT STD_LOGIC
        );
    END COMPONENT;

    signal clk_out : STD_LOGIC := '0';



begin
    clk64kHz_map: clk64kHz PORT MAP(
        clk => clk,
        reset => reset,
        clk_out => clk_out
    );

    servo_pwm_map: servo_pwm PORT MAP(
        clk => clk_out,
        reset => reset,
        pos => pos,
        servo => servo
    );
end Behavioral;
```

Figure 7.5: Main function : Part 2