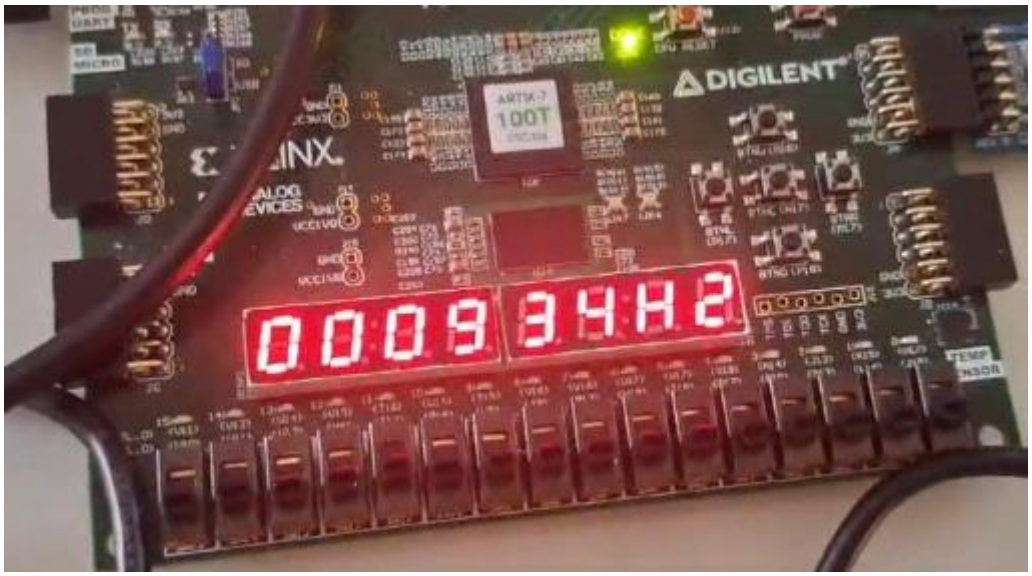
UE: Digital electronics

Digital Electronics Project



KADDOURI Nassira

AITELHADJ Yassmine

Département: EEEA

Major: Electronics

Year: 2024

Groupe: Jk

Introduction

The objective of this project is to design a complete instrumentation chain, covering the acquisition and digital signal processing. The project aims to measure the frequency of a real-time input signal using the Artix-7 FPGA from Xilinx.

The requirements specify that the input signal will have a frequency range between 10 kHz and 100 kHz. A high-pass filter with a cutoff frequency of 10 kHz must be implemented to eliminate low-frequency components. Additionally, the measured frequency should be displayed on the 7-segment display provided with the Artix-7 board.

The proposed architecture for the project is illustrated in Figure 1.

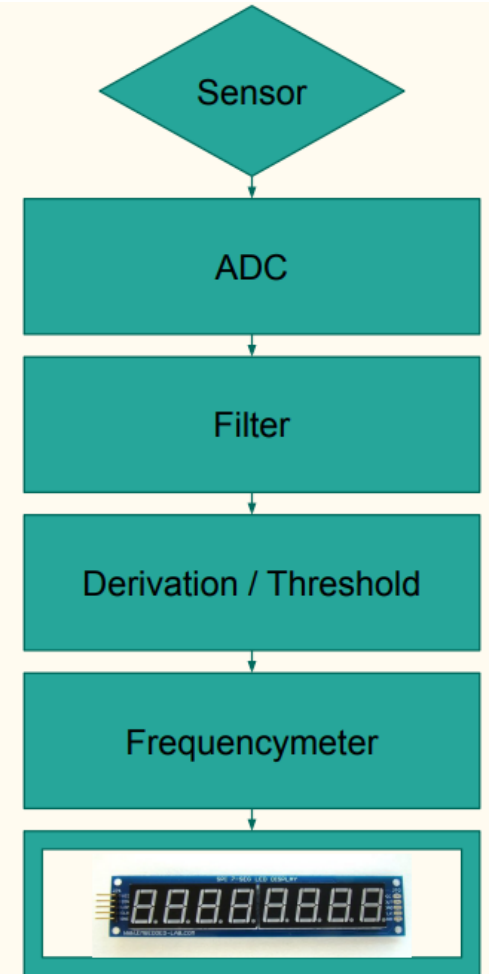


Figure 1 : Organizational chart of the project

1. Data acquisition and validation of the ADC

In this part, we validated the functionality of the integrated ADC in the Artix-7, where we implemented our frequency meter project. To achieve this, we used a DAC based on an R2R ladder structure connected to one of the PMOD connectors. This type of DAC, being passive, was simple to implement and required feeding with digital data refreshed at the appropriate rate, matching the ADC's sampling rate. We first tested the DAC by feeding it a ramp signal, which allowed us to successfully display a sawtooth signal of 255us period at its output, confirming its proper operation.

1.1 Analogic to digital conversion

The goal of the first part is to convert an analog signal into a digital signal. This allows us to process the signal and retain only the characteristics necessary for frequency analysis. The initial approach involves using the FPGA's integrated converter, called the XADC. Using the Xilinx Analog-to-Digital Converter (XADC) Wizard in LogiCORE IP, an HDL file was generated to configure the converter.

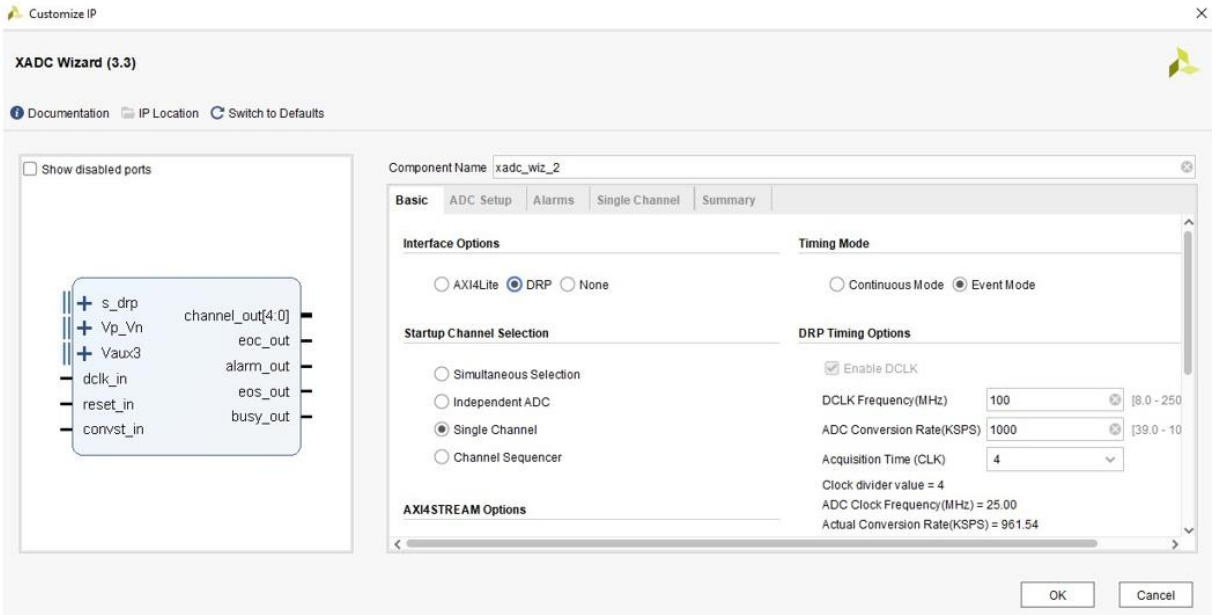


Figure 2: configuration of the XADC

According to Figure 2, the DRP interface (default) is initially considered. The Startup Channel Selection is configured for single-channel mode, allowing us to monitor only one channel. The Timing Mode is set to Event Mode, meaning an external trigger is required to start the conversion. The ADCCLK is derived from the DRP clock and must fall within the range of 4–26 MHz. The Vivado IDE calculates the clock divider based on the desired conversion rate, which is set to 1 Msps, and the DCLK frequency of 100 MHz.
The transfer function can be obtained from the XADC datasheet. It shows that a 12-bit ADC converter is used with an amplitude of 1V. Code transitions occur in successive values of the LSB (Least Significant Bit), where LSB = 1V/1012 = 244 µV.

We aim to use the input signal in differential mode, which requires always having a positive input (Vp) and a negative input (Vn). We will use the Digilent PmodTPH2 module, which allows us to apply the differential input to the ADC.

1.2. Digital to analogic conversion

To validate the previous converter, we focus on the inverse conversion. We will use an R-2R DAC.

Unfortunately, the Pmod R2R only operates with 8 bits. Therefore, we will use only the 8 most significant bits (MSB) from the ADC output register, as the MSBs contain the shape of the signal, while the least significant bits (LSBs) are mainly for precision.

To properly verify the output, the DAC needs to be fed, and the digital data must be converted. This conversion should ideally occur at the same sampling rate as the ADC. Therefore, we will create a timing block (tempo) to synchronize them.

After combining them, we generate a 10 khz signal that has positive and negative sinusoids, give it as input of the ADC through the PmodeTPH2 and visualize the output of the DAC in the oscilloscope. And Indeed, we fine that the input of our system and the output are identical as shown in the following figure, thus the validation of the proper functionality of the ADC.

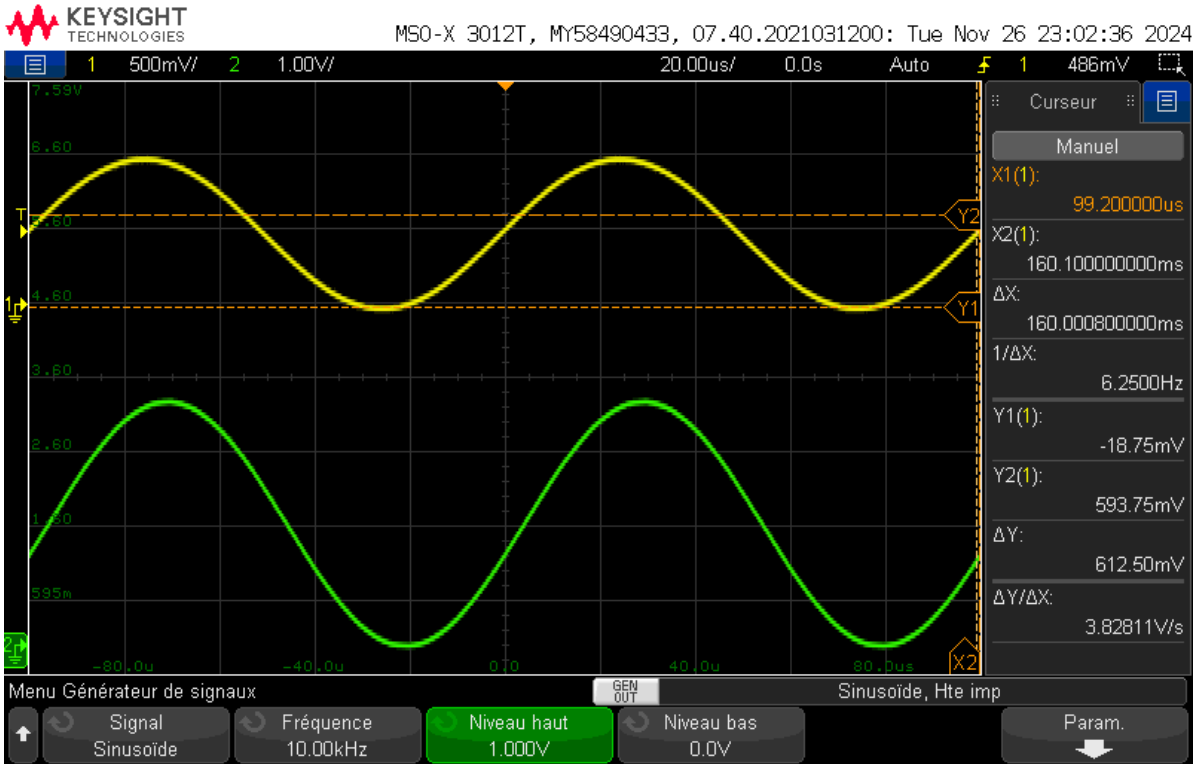


Figure 3: validation of the ADC with reverse conversion

2. Signal conditioning

2.1 Filter

We want to design a first-order high-pass filter with a cutoff frequency of 10 kHz.

The transfer function of such filter:

$$H(p) = = \frac{p\,\tau}{p\,\tau + 1}$$

Where: τ is the time constant of the filter, related to the cutoff frequency f_c by $\tau = \frac{1}{2\pi f_c}$.
we start by relating the output Y(p) and input X(p) using the transfer function:

$$Y(p) = H(p)X(p) = \frac{\tau p}{\tau p + 1} X(p)Y(p)$$

Next, we multiply both sides by $(\tau p + 1)$ to eliminate the denominator:

$$(\tau p + 1)Y(p) = \tau p X(p)$$

Taking the inverse Laplace transform of both sides, where p corresponds to $\frac{d}{dt}$, we get the differential equation:

$$\tau \frac{d}{dt} y(t) + y(t) = \tau \frac{d}{dt} x(t)$$

When we discretize time, we set x_k as the input at the instant t_k and y_k as the output at the instant t_k, where $t_k = t_0 + k \cdot h$ and h is the time step.

We then obtain $\tau \left(\frac{y[k]-y[k-1]}{h}\right) + y[k] = \tau \left(\frac{x[k]-x[k-1]}{h}\right)$

Rewriting this equation gives:

$$y_k = \frac{1}{1 + 2\pi f_c h}(x_k - x_{k-1} + y_{k-1})$$

We can replace $\frac{1}{1+2\pi f_c h}$ by K, if $f_c = 10\ khz$ and $h = \frac{1}{Fs} = 10^{-8}$
Then $K = 0.9409$

We were able to implement the recursive equation of the filter in VHDL by creating two blocks: one for **memory** and one for **calculation**:

Memory Block: This block stores the previous values of the input and output signals x_{k-1} and y_{k-1} , which are needed for the recursive calculation.
Calculation Block: This block performs the filter's recursive computation using the previous equation to calculate the current value of the output.

We decided to encode each number using 9 signed integer bits and 13 fractional bits, for a total of 22 bits (21 down to 0). The 9 signed integer bits correspond to the 8 bits of the input signal

plus 1 sign bit. The 13 fractional bits allow us to ensure a precision up to the ten-thousandth place (with a conversion error smaller than $10^{\{-4\}}$. For example, $x_k = 45.687$ would be encoded as "0010 1101, 1010 1011 0000 0".

```
14  entity Filtre is
15    Port (
16          clk: in std_logic;
17          rst: in std_logic;
18          Signal_Tempo_1us: in std_logic;
19          x_p: in std_logic_vector(7 downto 0);
20          y_p: out std_logic_vector(7 downto 0)
21          );
22  end Filtre;
```

We Displayed the output of the filter in the oscilloscope and measured its cutoff frequency with the method of 7/5 caree, we found $f_c = 8.5$ khz, close to 10 khz. This difference might be induced by the limited precision of the calculation or the discretization of data.

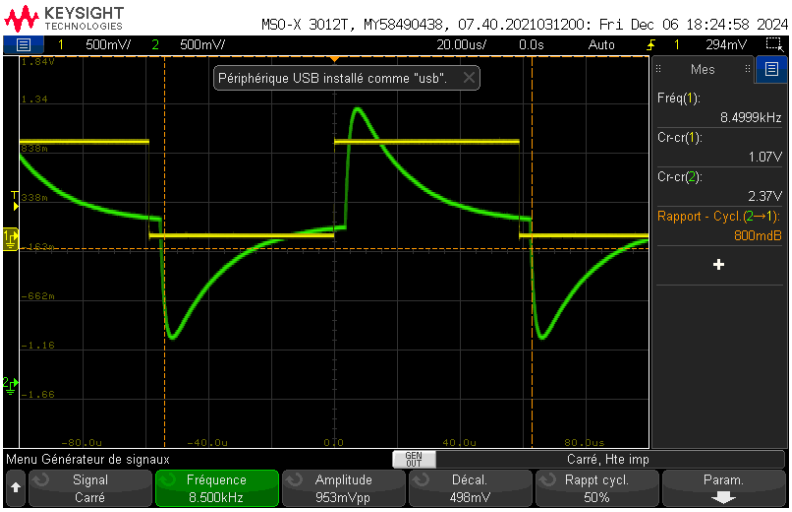


Figure 4: response of the filter to a square shaped signal

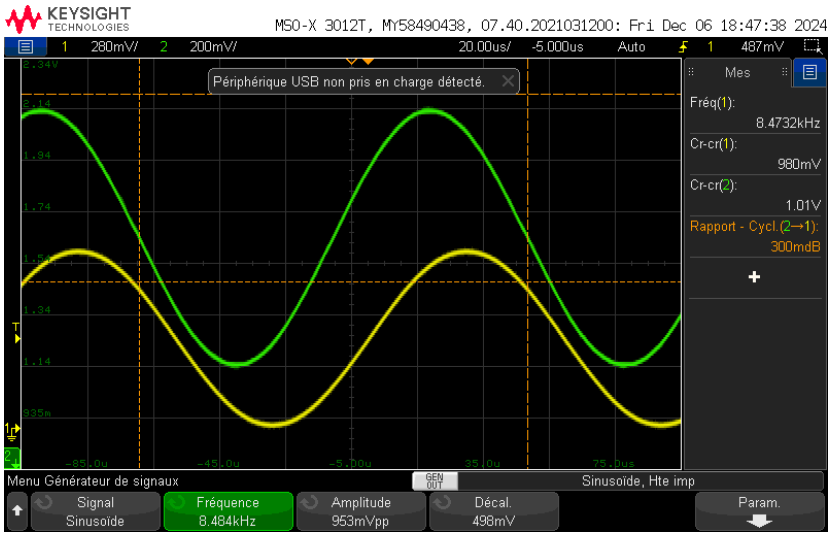


Figure 5: response of the filter to a sinusoidal Input

2.2 Differentiator (Derivateur)

We are now tasked with implementing a differentiator. The system input is more like a triangular signal, and the differentiator will help transform the slopes of the triangle into constants, which will make it easier to detect the beginning of a period.

To implement the differentiator, we need to perform the following operation:

$$\frac{\mathrm{dx(t)}}{\mathrm{dt}}$$

By discretizing this, we obtain:

$$(t) = \frac{x_k - x_{k-1}}{h}$$

To avoid a complicated multiplication by $\frac{1}{h}$ that only influences the amplitude of the signal, while keeping the filter's output significant, we propose replacing the factor $\frac{1}{h}$ with 4, which gives the following equation:

$$yk = 4 \cdot (xk - xk - 1)$$

Thus, the differentiator principle becomes a simplified version of the filter.

```
34  entity Derivateur is
35      Port (
36              clk: in std_logic;
37              rst: in std_logic;
38              Signal_Tempo_1us: in std_logic;
39              x: in std_logic_vector(7 downto 0);
40              y: out std_logic_vector(7 downto 0)
41      );
42  end Derivateur;
```

To test this differentiator, we applied a triangular signal with a 100% duty cycle (sawtooth signal). The output waveform, as observed on the oscilloscope, is shown in Figure 6.

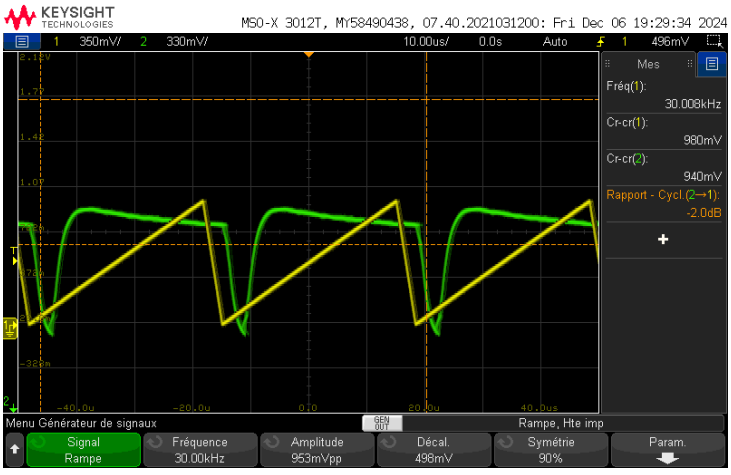


Figure 6: response of the derivative to a sawtooth signal

The output is a constant that corresponds to the slope of the sawtooth signal given in the input, which proves that the differentiator effectively outputs the derivative of its input.

2.3. Thresholding (Esseuillage)

This part is very simple. We just need to remove the offset that we added in order to adapt our output signal's representation to the DAC's encoding, then we should compare the current value of our signal to 0. 0 is our threshold.

This Bloc will provide as with a sort of pulses whenever the input signal goes beyond 0. This output is going to help us calculate the period of the signal easily.

```
33  entity Detect_Front is
34    Port (
35              clk: in std_logic;
36              rst: in std_logic;
37              Signal_Tempo_1us: in std_logic;
38              x: in std_logic_vector(7 downto 0);
39              y: out std_logic
40      );
41  end Detect_Front;
```

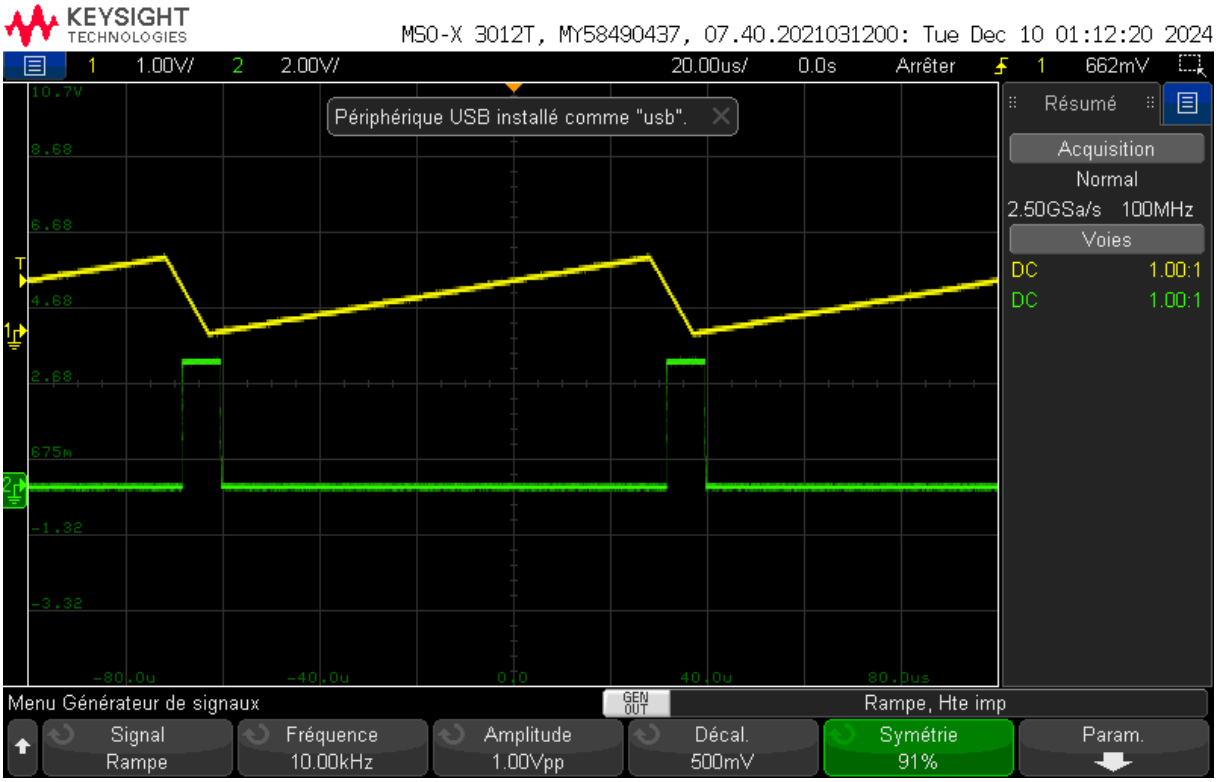


Figure 7: Output of the thresholding

3. Frequencymeter

3.1 Period counter

To measure the signal frequency, we count the number of sampling periods (Te = 1 µs) between two consecutive signal periods. The counter initializes at the start of a period and outputs its final value continuously, as shown in the finite state machine in Figure 8.

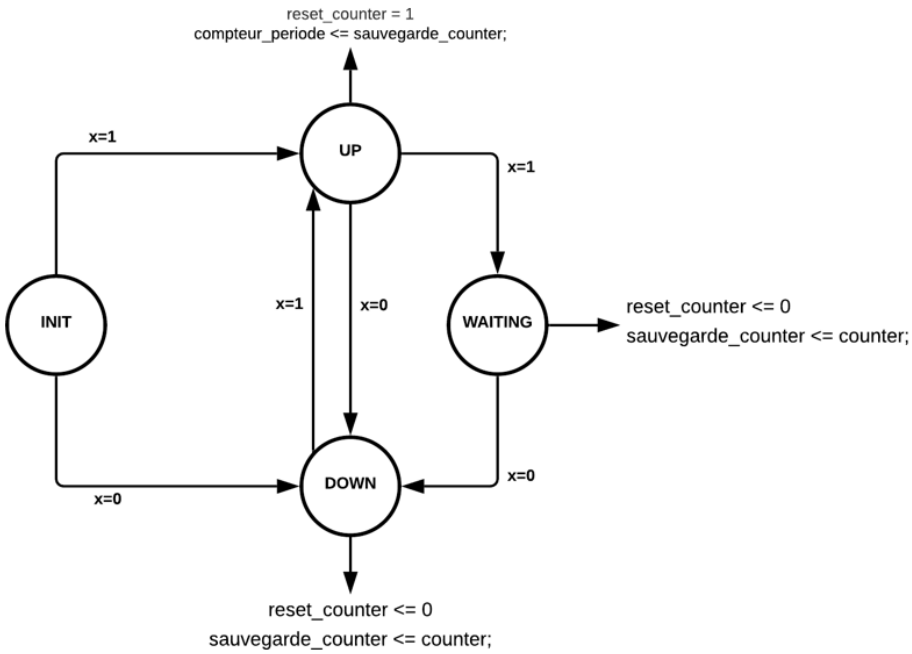


Figure 8: FSM of the period counter

This period counting block needs as inputs the 1us timer and the frequency pulse (binary signal generated with the previous block of thresholding), as shown in the following entity port.

```
entity Period_Counter is
    Port (
            clk: in std_logic;
            rst: in std_logic;
            Signal_Tempo_1us: in std_logic;
            pulse_frequency: in std_logic;
            compteur_periode: out std_logic_vector(12 downto 0)
    );
end Period_Counter;
```

With a testbench we create a pulse signal and simulate to make sure that the period counting block outputs the same period we set for the signal. 399 us in this case.

Figure 9: testbench output of period counter in response to a 399 us period pulse in the input

3.2 Divider

Knowing the number of sampling periods during one signal period, we can express the period:

$$T_{signal} = k \cdot Te$$

here T_{signal} is the signal period, T_e is the sampling period, and k is an integer. To calculate the frequency:

$$f = \frac{f_e}{k}$$

For this, we implement a divider based on the Euclidean division, represented in the following FSM.

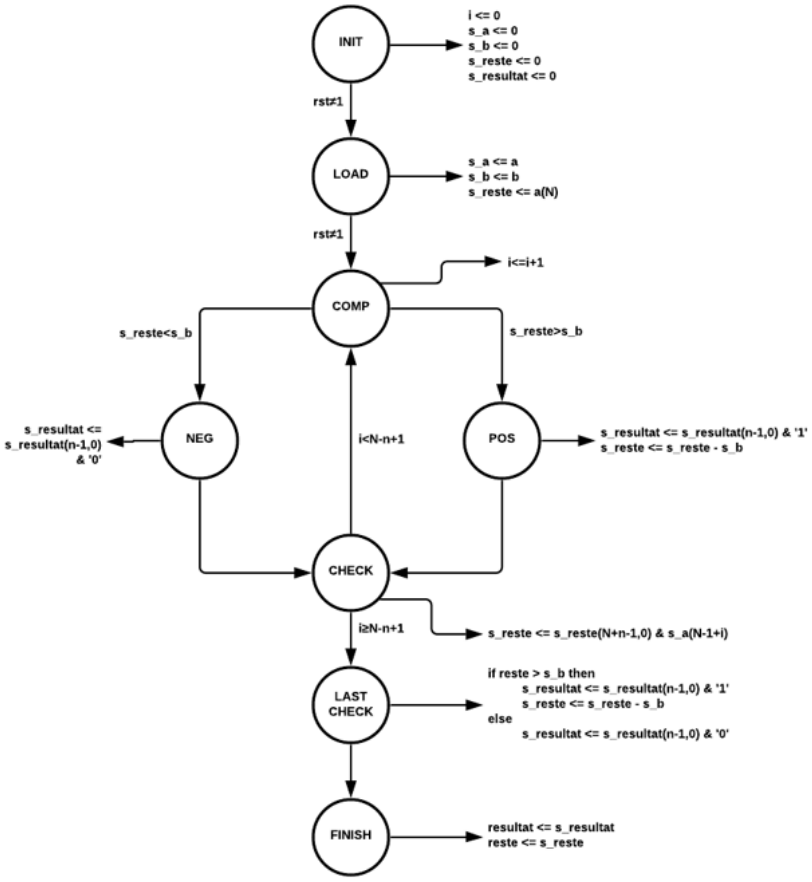
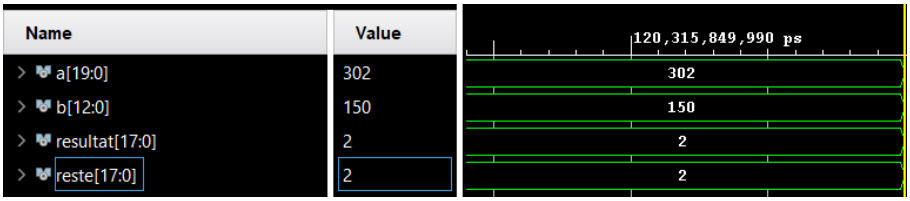


Figure 10: FSM of the divider block

Here is the entity port that describes the implementation of this block:

```
34  entity Diviser is
35      Port (
36            clk: in std_logic;
37            rst: in std_logic;
38            a: in std_logic_vector(19 downto 0);
39            b: in std_logic_vector(12 downto 0);
40            resultat: out std_logic_vector(17 downto 0);
41            reste: out std_logic_vector(17 downto 0)
42            );
43  end Diviser;
```

And here is a testbench where we gave 302 as value for a and 150 for b, the block divides a over b and it outputs a result of 2 and a remain of 2:



4. 7-segments display

The display block consists of two components: a BCD (Binary Coded Decimal) to 7-segment converter, which lights up the corresponding LEDs to display the appropriate digit, and a multiplexer that selects the power of 10 (units, tens, hundreds, ...) to be displayed.

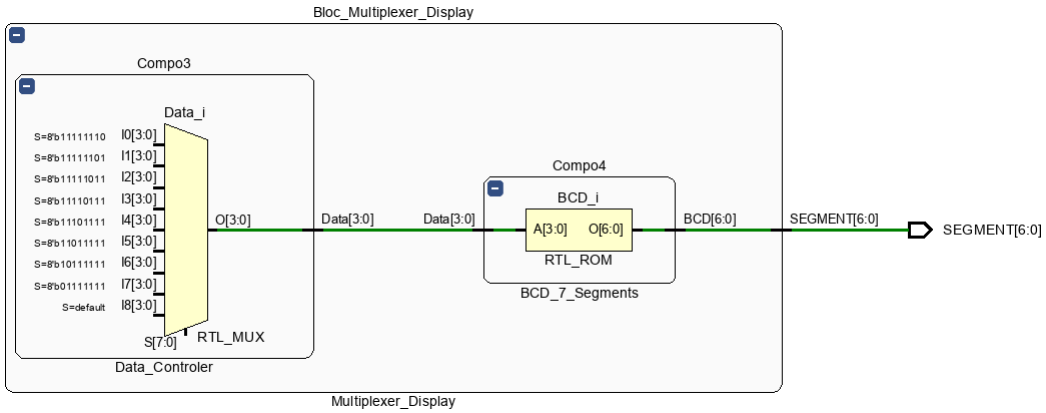


Figure 11: Display bloc

4.1. Clock devider

We aim to activate the anode corresponding to the desired power of 10. To achieve this, we need to display them successively at a certain frequency, here 1 kHz.
We start by creating a clock divider (clock_divider) that provides a "clk" signal at 1 kHz. The bloc generates a pulse every 1 ms, lasting for a one clock cycle.

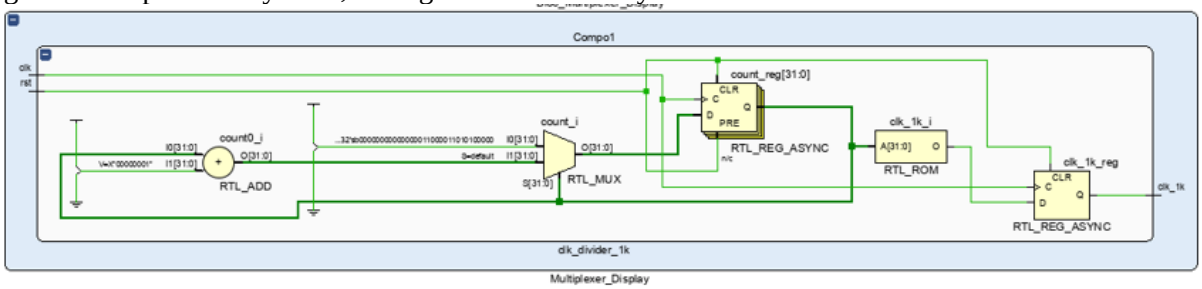


Figure 12: clock divider bloc

4.2 AN counter

Next, the correct power of 10 must be selected for display. The AN_Counter component will select the anode to activate every 1 ms to display the corresponding power of 10.

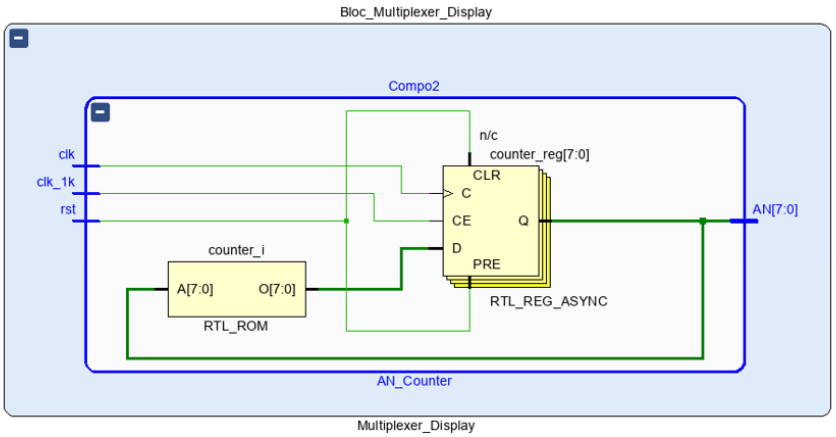


Figure 13: AN counter bloc

A simple testbench can prove the correct functioning of the counter as follows

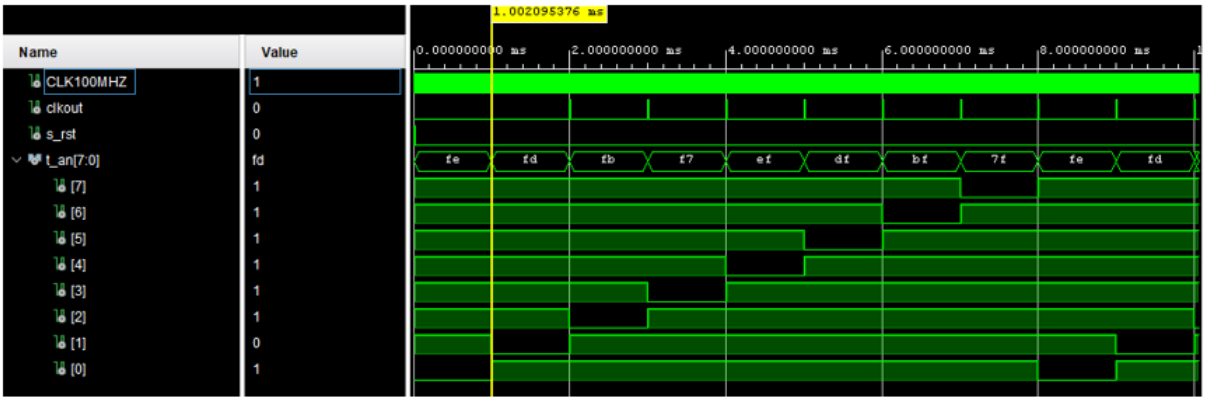


Figure 14: simulation of AN counter bloc

4.3 Data controller

The next block, the "data_controller," simply connects the correct power of 10 to the appropriate 7-segment display. It also determines which display uses the decimal separator, by evaluating if the frequency is in Hz or KHz and what digits are equal to 0.

4.2 Binary to Binary-Coded Decimal (BCD) Converter

The final block of the display is responsible for converting a binary value into decimal. The goal is to separate each part of the number to display it on the correct BCD, allowing us to show the units, tens, and hundreds of the values. The state machine is shown in the following figure.

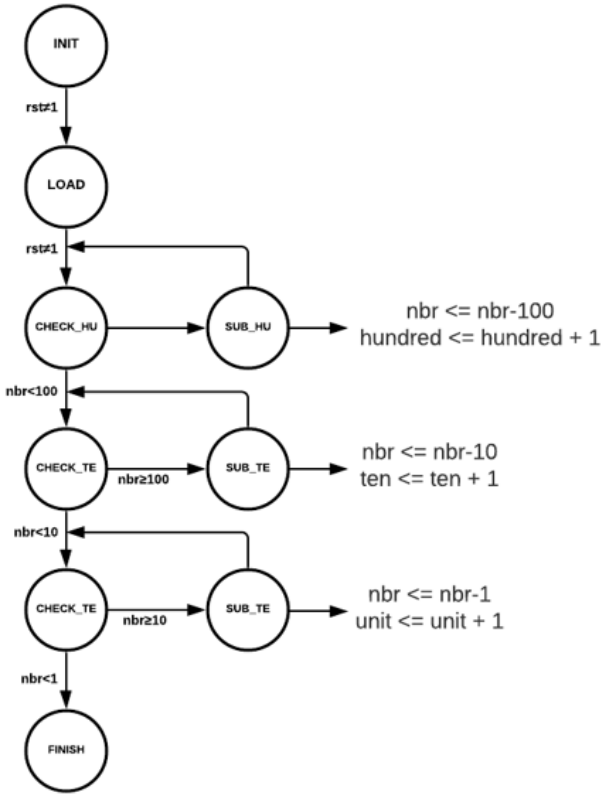


Figure 15: FSM of the binary to BCD converter

The following image shows the functioning of the previous blocs, where we input a signal with a certain frequency and we can see the display of the frequemcy on the 7 segments displayer.

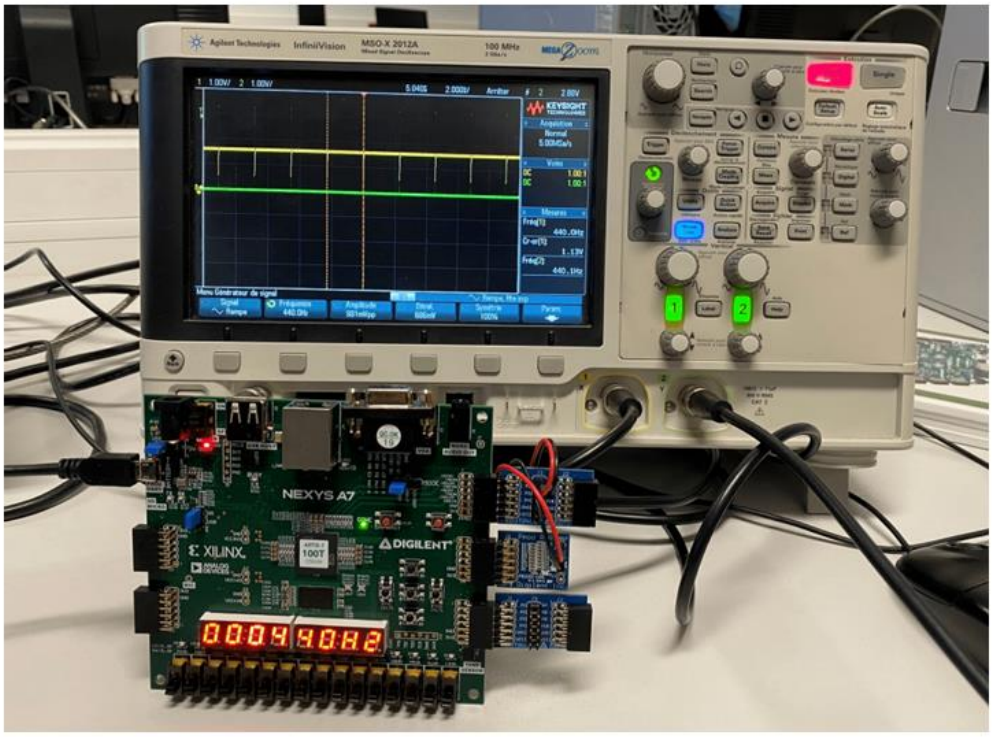


Figure 16: frequency display on 7 segments

Conclusion:

In this project, we applied our VHDL skills to a signal processing application. We successfully developed an acquisition chain to display the frequency of a signal on an FPGA board, adhering to a predefined set of specifications. This experience placed us in a situation similar to those encountered in the professional world.

We would like to express our heartfelt thanks to the entire guiding and tutoring team for their invaluable support throughout this project.