POLITECNICO DI BARI

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING

Master's Degree in Automation Engineering

REAL TIME IMPLEMENTATION OF TRAPEZOIDAL TRAJECTORY

Professor:
Dr. Eng. Martino De Carlo

Candidates:
Bruno Gianluca
D'Arcangelo Luca
Zinfollino Vincenzo

ACADEMIC YEAR 2023–2024

Contents

Introduction

The speeding up of the technological evolution and the reduced time to market that the industries have to face off, have meant that in the last decade the industry has increasingly relied on FPGAs, introducing them more and more pervasively in high-end products. From medicine to aerospace, FPGAs play a fundamental role in ensuring very high performance and constant repeatability, even in adverse and mission-critical conditions. Thanks to their infinite flexibility they are widely used in the Automation sector, as they naturally respond to some crucial needs of the sector, for example to mention some fundamental characteristics:

- **High Processing Speed:** FPGAs can perform parallel processing at very high speed, enabling real-time control of motors and actuators. This capability is critical for applications that require fast and precise responses, such as controlling industrial robots or CNC machines.
- **Flexibility and customization:** FPGAs can be reprogrammed to suit different applications and control algorithms. This flexibility allows you to optimize the control system for specific operational needs, improving the efficiency and precision of the movement.
- **Predictable latency:** Thanks to their hardware architecture, FPGAs offer very low latency compared to traditional microcontrollers or processors. Unlike microcontrollers where the execution time of a generic instruction is random, in FPGAs since the code is reproduced in hardware we can measure the latencies exactly.

In our project we wanted to explore the possible applications of an FPGA in the field of motion control. As previously mentioned, FPGAs are widely used in CNC machines and manipulators, to guarantee high-precision synchronous movements on multiple axes without deteriorating speed and flexibility. This is possible by carrying out the real-time trajectory calculation for each axis. For any micro controller this is not an entirely simple task, since however fast the execution may be the execution of the instructions will always be sequential and not parallel, a solution could be to use multiple micro controllers in parallel, but in that case you would have a increase in complexity, due to the need for communication and an increase in costs. To avoid this, FPGAs are used which guarantee parallel management of the code by design.
One of the most used trajectories in the field of motion control is the trapezoidal trajectory since it does not require too complex calculations and this guarantees a predictable and repeatable implementation capable of being implemented in real time. A trapezoidal trajectory is characterized by its distinct acceleration, constant velocity, and deceleration phases, which together form a trapezoidal shape when velocity is plotted over time. This trajectory is used to move the shaft of the motor from one position to another efficiently and smoothly, keeping in consideration the constraints on maximum acceleration and velocity.In motion control systems, implementing a trapezoidal trajectory involves specifying parameters such as the desired start and end positions, maximum velocity, and maximum acceleration. The control system then calculates the velocity set-point in real time for each phase to ensure smooth and efficient motion.

In order to demonstrate the possibility of implementing the trapezoidal trajectories using an FPGA, for the development of our project we decided to use a stepper type motor, which will be moved by using a trapezoidal trajectory. Stepper motors, due to

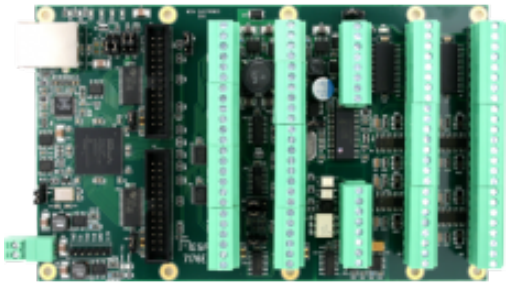

Figure 1: The 7I76E ia a remote FPGA card based on Xilinx hardware, with an Ethernet interface designed for interfacing up to 5 Axis of step &dir step motor or servo driver. This board is a commercially available solution to develop motion controlled systems, using Linux CNC distros

the way they are built, require an acceleration and deceleration profile to ensure that the movement is implemented correctly. Unlike DC motors where you just need to ramp up or down the voltage (increasing or decreasing the duty cycle of the PWM) to vary the velocity of the shaft. Instead,for moving the stepper correctly, the frequency of the PWM signal that we supply to the Driver must be varied accordingly to the velocity that we want.

Our project was divided into several phases:

- **Derivation of the analytical relationships:**In this phase we have determined, using pen and paper, the analytical relationships that exist between the trajectory that we want implement and the signal that must be provided to the driver. In this phase, several Matlab scripts were created to verify the correctness of the relationships that we found.

- **Design of the main components:** In this phase the main components to be created and their interfaces have been defined. It was decided to create a modular system composed of several parts

  - **Frequency Modulator:** Represents a generalized PWM generator. We decided to implement a PWM generator that is as general as possible so that we can use it, if necessary, we can modify both the modulating and the carrier signal.It takes the calculated speed set-point as input and implements it accordingly to the driver specification, in the form of a square wave signal.
  - **Trajectory generator:** Implements a finite state machine within which the speeds to be supplied to the motor instant by instant are calculated. Since this module is also as general as possible, it is possible to modify all the core parameters of the trajectory as desired, such as the maximum speed and acceleration that the trajectory must respect.

- **Implementation:** In this phase we implemented all the components that we have designed using VHDL.One of the most difficult challenges in the implementation

phase was the management of the timing between the components. For correct management, several internal clocks were generated which powered different modules. The timing problems were really difficult to identify but thanks to the help of Multisim and the analytical validation carried out with Matlab previously we were able to recognize them during the debug phase.

- **Validation:** The last phase was the validation, where we determined a trajectory in simulation, using both Multisim and Matlab scripts we verified that the movement was correct and above all had good repeatability.

Case Study

Trapezoidal velocity profile

To control a stepper it is necessary to impose a speed profile in order to manage the speed of the stepper during the movement. A **trapezoidal trajectory** was used, which is composed by three different parts:

- **constant acceleration:** v_c increases from 0 to a maximum velocity value v_{max}
- **costant velocity:** v_c remains constant at the value v_{max}
- **constant deceleration:** v_c decreases from v_{max} to 0.

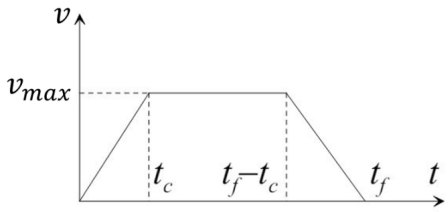

Figure 2: Trapezoidal velocity profile

To impose maximum speed values v_{max} and acceleration allowed a_{max}, it is necessary to consider the following equations:

$$\begin{cases} t_c = \dfrac{v_{\max}}{a_{\max}} & \text{acceleration time} \\ h = v_{\max}\,(t_f - t_c) = q_f - q_i & \text{distance travelled} \end{cases}$$

In this way, by fixing the values of v_{max} and a_{max} it is possible to calculate the final time:

$$t_f = (t_f - t_c) + t_c = \frac{h}{v_{max}} + \frac{v_{max}}{a_{max}} \tag{1}$$

In order for the total distance traveled h to be sufficient to allow the system to reach the v_{max}, keep the speed v_{max} constant and decelerate until reaching at 0, the following condition must hold:

$$h \geq \frac{v_{\max}^2}{a_{\max}} \tag{2}$$

obtained by imposing:

$$t_f \geq 2t_c \implies t_c \leq \frac{t_f}{2}$$

By imposing a trapezoidal speed profile, it is possible to obtain several advantages including:

- **avoid missing steps:** excessively sudden acceleration or deceleration can cause a loss of steps as the stepper would not be able to follow the microsteps provided. The trapezoidal trajectory helps avoid this type of problem by allowing the motor to accelerate and decelerate gradually.
- **reduction of mechanical stress:** Sudden acceleration or deceleration can impose significant mechanical stress on the engine and any system connected to it, and the trapezoidal trajectory helps distribute this stress evenly, reducing the risk of damage.

2.1.1 Special case: triangular trajectory

In the case of:

$$h < \frac{v_{\max}^2}{a_{\max}} \tag{3}$$

the maximum speed v_{max} is not reached. Consequently, the total distance h is not sufficient to allow the system to reach the v_{max} within the fixed time t_f. In this case a triangular velocity profile is used as shown in the following figure:

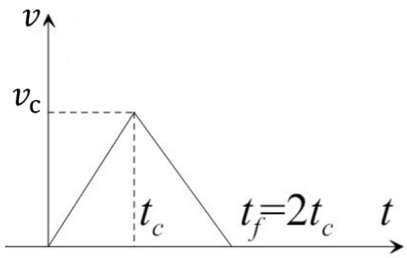

Figure 3: Triangular trajectory

In the current condition, the speed increases up to a value v_c and after reaching it it decreases with a constant deceleration without ever reaching the maximum speed v_{max}. The speed achieved in this case is equal to:

$$v_c = a_{max} \cdot t_c \tag{4}$$

PFM

It is not possible to directly drive a stepper motor using only an FPGA, as the FPGA output pins can provide a maximum of 3.3 V and a limited current, insufficient to adequately power the stepper. For this reason, it is necessary to use a motor driver that can provide the appropriate voltage and current to the motor. In our project, we chose to use the TMC2209 driver. This driver is specifically designed for controlling stepper motors, ensuring appropriate power and enabling precise and efficient motor control.

The type of driving signal to be applied to the motor driver that we have chosen to use is the PFM (Pulse Frequency Modulation) signal. This technique allows you to generate a pulse train based on the comparison between a periodic carrier signal and a constant modulating signal according to a driving strategy. In this way it is possible to modify the frequency of the pulses generated and applied to the driver in an appropriate manner and consequently vary the speed of the motor.

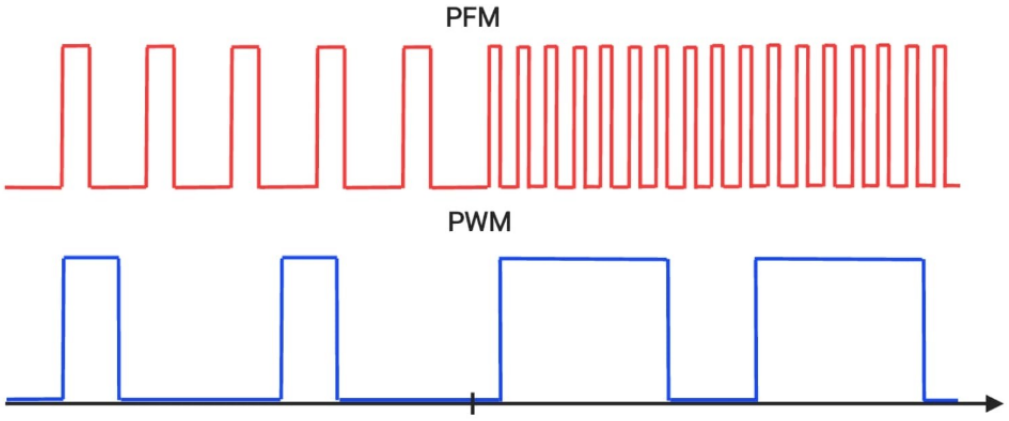

Figure 4: Comparison between PFM and PWM

PWM, on the other hand, modulates the pulse width to vary the duty cycle of the PWM signal, which is better suited for controlling the average power supplied to DC motors but does not provide the same level of precision in speed control for stepper motors as it is not possible to vary the carrier signal.

The frequency of the pulses will be varied appropriately based on the respective sections of the trapezoidal trajectory that we have implemented on the board. During the acceleration phase, the pulse frequency will gradually increase. In the constant speed phase, the frequency will remain stable, and during the deceleration phase, the pulse frequency will gradually decrease. This approach allows for smooth and precise control of motor motion, minimizing vibration and improving overall system efficiency.

2.2.1 ARR/CCR

The Auto-Reload Register (ARR) and the Capture/Compare Register (CCR) are critical to the generation of PFM (Pulse Frequency Modulation), or PWM (Pulse Width Modulation) signals needed to control the motor.

- **Auto-Reload Register (ARR):** defines the period of the signal, i.e., the time it takes for the timer to count up from zero to the value set in the ARR and then restart from zero. This value is updated to the frequency at which the motor driver is to operate.

$$ARR = \frac{f_{clk}}{PSC \cdot f_{microstep}} \tag{5}$$

- **Capture/Compare Register (CCR):** determines the duration of the high pulse within each period, i.e., the duty cycle of the signal. The duty cycle is the percentage of the period when the signal is high. In the project, a CCR was set to always equal ARR/4

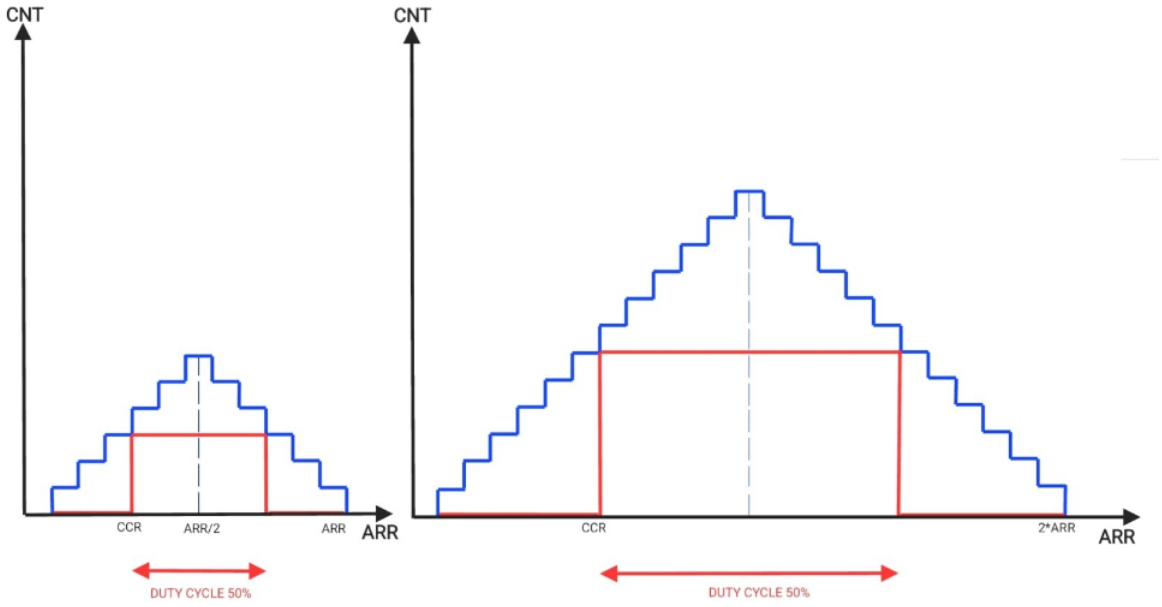

Figure 5: Periodic signal variation according to the value of ARR

During operation, the counter counts from 0 to the value of ARR. When the counter reaches the value of the CCR, the PWM signal goes from low to high (or vice versa, depending on the configuration) or stays at high until it exceeds the threshold value. The counter increases to ARR/2, and then decreases to 0, resetting the register so it can start again, thus generating a periodic signal. This produces to have a counter waveform corresponding to an equilateral triangle. The signal generated, on the other hand, will be a square wave, centered in the triangular signal, whose period is larger or smaller, depending on the value of ARR. The counter, unlike the ARR, updates with each clock stroke of the FPGA.

A speed control was implemented based on changing the value of ARR, since this corresponds to the desired speed (microstep/s). A zero ARR corresponds to a similarly zero

speed, while a higher value of ARR results in a slower speed, since the speed is inversely proportional to the period of the signal. Therefore, a high value of ARR corresponds to a lower frequency and thus a lower motor speed.

Hardware Implementation

Hardware components

A complete list of the components used in the project is shown below:

- DE10-Lite FPGA
- NEMA 17 stepper motor
- stepper motor driver TMC2209

3.1.1 DE10-Lite FPGA

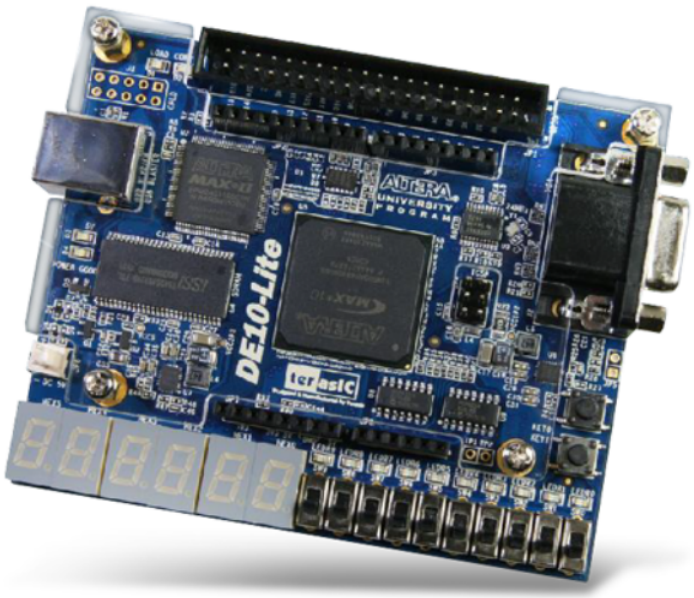

Figure 6: DE10-Lite FPGA

The DE10-Lite [2] board offers a robust hardware platform based on the Altera MAX 10 FPGA, ideal for cost-effective and energy-efficient solutions in control and data path applications. The MAX 10 FPGA guarantees industry-leading programmable logic, providing great design flexibility. It is particularly suitable for high-volume applications such as protocol bridging, motor control, analogue-to-digital conversion, image processing and portable devices.

Features of the board:

- **Altera MAX 10 FPGA**
  - MAX 10 10M50DAF484C7G
  - Integrated dual ADCs
  - 50K programmable logic elements
  - 1638Kbit M9K memory

  - 144 18 × 18 multiplier
  - 4 PLLs
- **Programming and Configuration**
  - On-board USB blaster (normal type B USB connector)
- **Memory Device**
  - 64MB SDRAM, x16 bits data bus
- **Sensor**
  - Accelerometer
- **Expansion Connectors**
  - One 2x20 GPIO connector(voltage levels: 3.3V)
  - Arduino Uno R3 connector, including six ADC channels
- **Switches/Buttons/LEDs**
  - 10 LEDs
  - 10 slide switches
  - 2 push buttons
  - Six 7-segments displa
- **Power**
  - 5 V_{DC} input

In the project was used:

- One switch SW to set the input angle, which can vary between two values.
- Two GPIO pins connected to the motor driver, PIN_V10 to turn the PFM signal high or low and the second, PIN_W10, to set the direction of movement.

3.1.2 NEMA 17 stepper motor

Stepper motors operate by dividing a full rotation into a large number of equal steps. The motor's position can be commanded to move and hold at one of these steps without any feedback sensor (an open-loop system), as long as the motor is carefully sized to the application. The rotor of the stepper motor is a permanent magnet, and it is surrounded by a stator with windings. By energizing these windings in a specific sequence, a rotating magnetic field is created, which pulls the rotor into alignment with the magnetic field of the stator. This sequence of energizing the windings allows the motor to move in precise increments, making stepper motors ideal for applications where precise positioning is required.

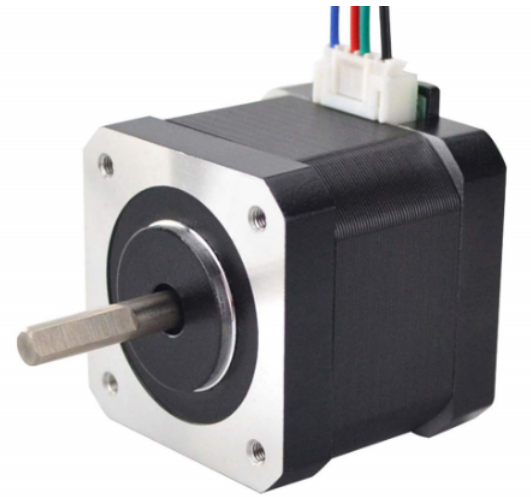

Figure 7: Image of the stepper motor used

The choice of the NEMA 17 motor, a popular option for many small robotics and automation applications, was motivated by several reasons, primarily because it offers a good balance between performance and cost.

One reason for choosing this motor is its strong torque despite its compact size and the precision control enabled by its high number of steps per revolution. This allows the motor to follow a precise trajectory given as input, even with a load attached.

Another important reason for choosing this model is the availability of a variant that includes a magnetic incremental encoder mounted directly above the stepper motor. This feature makes the entire motor system more compact. In applications requiring closed-loop control, all the control methods already implemented can be used, with the addition of the encoder readout. This makes it possible to improve the accuracy and reliability of the system without having to make significant changes to the existing control setup.

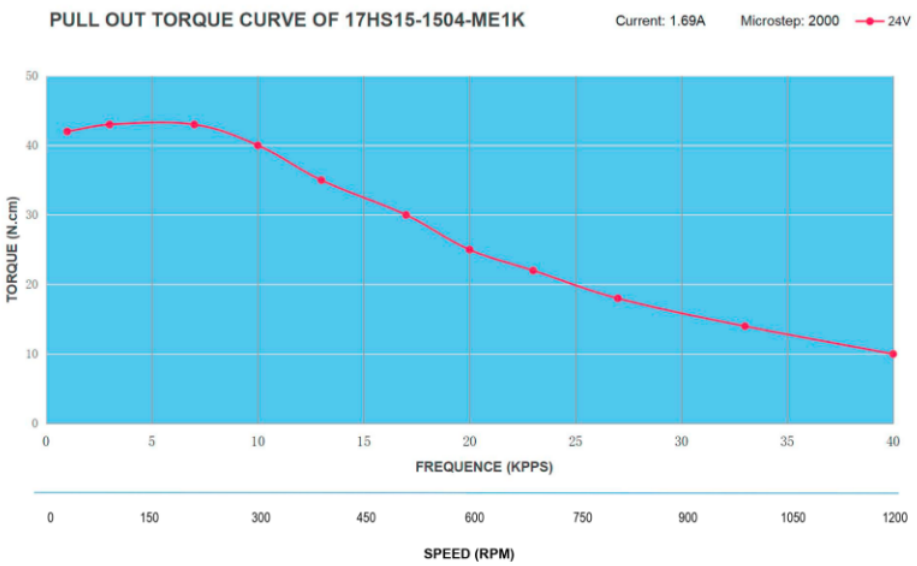

Figure 8: Torque Curve of the stepper Motor

3.1.3 Stepper motor driver TMC2209

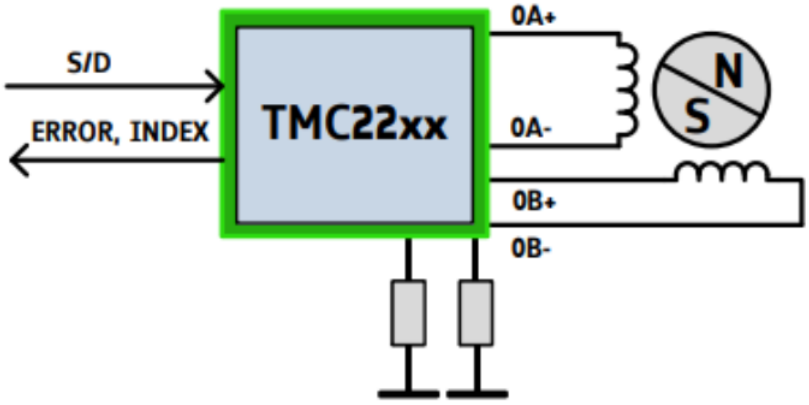

Figure 9: TMC2209 Driver

The motor driver chosen to control the motor is the TMC2209 due to its high performance, integrated protections, easy configuration, and broad compatibility. It is particularly suitable for applications that require precise and reliable control of stepper motors. Moreover, this driver allows microstepping, allowing higher resolutions than the number of motor steps by enabling intermediate step locations. These intermediate step locations are achieved by energizing the coils with intermediate current levels. To select one of the six possible step modes, the three dedicated resolution pins must be correctly connected. For more information, please refer to the driver datasheet [5]. In the project has been used the 1/16 step micro-stepping resolution, that allows to control the position of the motor with more precision because there are more available steps and allows to have enough range in terms of max speed affordable by the motor. In fact, Nema 17 has normally 200 steps for revolution, but with micro-steps, there will be 3200 micro steps for revolution.

$$Risolution = \frac{2\pi[rad]}{3200[microstep]} = 0,0019[rad/microstep] \tag{6}$$

From the datasheet, is known the maximum step frequency (at the maximum resolution of resolution in microsteps) so a lower frequency was chosen in the design.

$$f_{step} = \frac{f_{clk}}{2} = \frac{12MHz}{2} = 6MHz \tag{7}$$

To control the motor, only two pins are needed: one for the direction and one for the step. The direction is simple, logic level 0 is clockwise rotation, while logic level 1 is counterclockwise (according to the motor wires connections). To trigger a step on the step pin, a transition from 0V to 5V is required. Therefore, if a continuous rotation is desired, a square wave signal needs to be generated. However, with this driver is not possible to do a direct position control but only a velocity control, because it takes as input only the frequency of how many steps for seconds it has to do, and the direction. So, the frequency of this signal is proportional to the wanted speed. To impose the velocity, it has been necessary to use a counter timer for the PFM generation. Once the

desired velocity is known, there will be some operations to do for setting the properly the registers:

- **Auto-Reload Register (ARR):** responsible of the period of the waveform;
- **CCR: responsible of the duty cycle:** The duty cycle has always been set to 25% (a quarter value of ARR). It is set to 0% only to stop the motor.

Software Implementation

Software used

The project was built using the VHDL language on the Intel Quartus Prime software. However, it was necessary to use ModelSim to simulate the various components and to find more easily the errors that occurred during development.

A behavioral approach has been used, that is, a high level of abstraction, in order to make it as close as possible to an algorithm. In fact, in this way, if-then-else and case-when are used.

The creation of the velocity trajectory and the PFM have been described separately and then imported into the Top level entity file as components.

4.1.1 Quartus Prime

Intel Quartus Prime is a powerful design software for programmable logic devices. It supports VHDL and Verilog hardware description languages, enabling analysis and synthesis of designs. The software includes advanced tools for visual editing of logic circuits and simulation of vector waveforms. With an intuitive user interface and integration with other EDA tools, Quartus Prime optimizes workflow and reduces development time, making it ideal for designing and implementing complex digital circuits on Intel FPGAs.

- **VHDL Code:** The Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is a specialized language used to define the operations of electronic circuits, particularly those of digital nature. VHDL serves a dual role: it enables the design of hardware components and facilitates the creation of test entities to validate their functionality, thereby minimizing both development costs and time.

4.1.2 ModelSim

ModelSim is an advanced simulation software, widely used for functional verification of digital circuit designs. It supports the main hardware description languages, VHDL and Verilog, allowing engineers to test and verify their designs accurately. With ModelSim, various components of a digital circuit can be simulated, including logic gates, flip-flops, memories, and other complex devices. The software provides a powerful interface for simulation and debugging, facilitating error detection and correction through features such as waveform visualization, time analysis, and signal tracing.

4.1.3 Matlab

Through a Matlab script, it is possible to verify the correct implementation of the trapezoidal velocity profile, using the formulas described in the theoretical part . The same values have been used in the code in VHDL. The code sets up a for loop that iterates through each time interval until the trajectory is completed. In each iteration, the speed is calculated according to the current phase of the movement. At the end of the iterations, the velocity profile is plotted as a function of time using the plot function.

```
TTC    = 0.00016; % 1/6MHz
V_max = 100;
A_max = 10;
tc     = 10;       % V_max/A_max
h      = 1600;     % distance


ctf = ((h/V_max) + tc)/TTC; % Time required to complete the trajectory

count_acc_part = tc/TTC;                  % acceleration part
count_const_part = ctf - count_acc_part;  % constant part
count_dec_part = ctf;                     % decelaration part


y=1:1:ctf-1;

Timer_trajectory=1;
v= [];

for i=1:ctf-1

    if (Timer_trajectory> 0) && (Timer_trajectory<=  count_acc_part)

      v(i)= A_max*Timer_trajectory*TTC;

    elseif (Timer_trajectory> count_acc_part)&& (Timer_trajectory <= ↙
count_const_part)

     v(i) =V_max;

    elseif (Timer_trajectory> count_const_part) &&(Timer_trajectory  <= ↙
count_dec_part)

      v(i)= (ctf*A_max*TTC)-A_max*Timer_trajectory*TTC;

    end

 Timer_trajectory=Timer_trajectory+1;

end

plot(y,v)
```

Figure 10: Matlab script

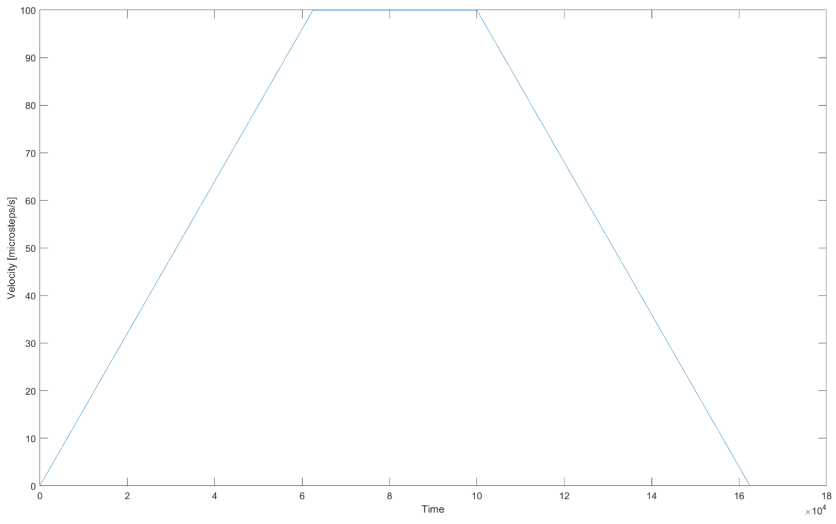

Figure 11: velocity profile in Matlab

4.2.1 Stepper Trajectory

The trapezoidal trajectory module calculates the ARR value based on the phase of the trajectory it is in. The value of ARR determines the frequency of the pulses sent to the stepper motor. The lower the ARR value, the higher the pulse frequency, and therefore the higher the motor speed. Each $ARR(t)$ value will be sent to a frequency modulator which will determine the period of each single pulse sent to the motor driver. Since the stepper's maximum velocity v_{max} and maximum acceleration a_{max}, expressed in RPM, must be fixed in order to calculate the final time of the trajectory, it is necessary to fix maximum values which have been obtained experimentally:

$$v_{max} = 1,875[RPM] \implies f_{microstep,max} = \frac{1,875[revolution] \cdot 3200[\frac{microstep}{revolution}]}{60[s]} = 100 \quad [microstep/s] \tag{8}$$

$$a_{max} = 10 \quad [\frac{microstep}{s^2}] \tag{9}$$

Libraries and Entity

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all; -- test
use ieee.std_logic_unsigned.all;

ENTITY Stepper_Trajectory is

    GENERIC(
        Prescaler_FSM :INTEGER := 8334; -- fFMSM=6 KHz:  for the PFM state machine
        TTC:INTEGER := 16; -- 16*10^-5 *10^5  conversion rate used for pass from time to counter value

        PRESCALER_ARR    : INTEGER := 50; -- 50MHZ/1 MHz ->1MHz < 6MHz (fmax stepper driver)
        CLOCK_FREQ_const : INTEGER := 50000000;

        V_max : INTEGER := 100; -- microstep/s V_max = 1,785 rpm
        A_max : INTEGER :=  10; -- microstep/s^2

        tc    : INTEGER := 10   -- V_max/A_max
    );

  PORT ( clk        : IN  STD_LOGIC;
         h_out      : IN    INTEGER;
         Step_count : IN    INTEGER;
         ARR_out    : OUT   INTEGER
       );
  END Stepper_Trajectory;
```

Figure 12: Libraries and Entity of Stepper_Trajectory

The entity has four ports:

- **clk:** the input clock signal of the board
- **h_out:** the distance to reach
- **Step_count:** the number of steps done during the time
- **ARR_out:** the value calculated in this module used for modulation

In addition to the ports, there are constants within the Generic, which are necessary for final time calculations and the ARR calculation.

```
ARCHITECTURE BEHAVE OF Stepper_Trajectory IS

SIGNAL Clock_PFM : STD_LOGIC:= '0'; -- Clock for the frequency modulation process
SIGNAL Clock_FSM : STD_LOGIC:= '0'; -- Clock for the counter of the Finite state machine

SIGNAL Count_PFM : INTEGER := 0;
SIGNAL Count_FSM : INTEGER := 0;

SIGNAL ARR : INTEGER   :=  0 ;
SIGNAL DIR : STD_LOGIC := '0';

SIGNAL v: INTEGER := 0;
SIGNAL a: INTEGER := 0;

-- COUNTERS:

SIGNAL ctf: INTEGER := 0; -- counter timer final: it reprents the final time with counts

SIGNAL Timer_trajectory     : INTEGER := 0;   -- counters for trajectory that increases with each TTC
SIGNAL Timer_trajectory_old : INTEGER := 0;

SIGNAL count_acc_part:   INTEGER   :=0;   -- counts required for the acceleration part
SIGNAL count_const_part: INTEGER   :=0;   -- counts required for the constant velocity part
SIGNAL count_dec_part:   INTEGER   :=0;   -- counts required for the deceleration part

--FSM Signal Declaration

SIGNAL data_in: STD_LOGIC:='0';

TYPE   state_values IS (HOLD, ACCELLERATION, CONSTANT_VELOCITY, DECELLERATION);

SIGNAL pres_state, next_state : state_values;

SIGNAL Sel_State   : STD_LOGIC_VECTOR( 1 downto 0) := "00" ;
```

Figure 13: Architecture of Stepper_Trajectory

Different processes have been implemented in stepper trajectory:

Process Clock FSM: the TMC2209 motor driver can accept a maximum step frequency ($f_{step,max}$) of 6 MHz, which determines the stepper's movement speed. To ensure that the FSM is synchronised with this maximum $f_{step,max}$ a clock divider is required to reduce the FPGA clock frequency from 50 MHz to a maximum of 6 MHz.

A prescaler of 9 reduces f_{clk} to 5.55 MHz, which is less than 6 MHz. In the project has been chosen a frequency of 6 kHz, so a prescaler of 8333 was needed.

$$\frac{50MHz}{6kHz} = 8333[PSC] \implies f_{FSM} = 6kHz \implies TTC = \frac{1}{f_{FSM}} = 16 \cdot 10^{-5}[s] \quad (10)$$

```
ClockDivider_FSM: PROCESS (clk)

BEGIN

        IF (RISING_EDGE(clk)) THEN

          Count_FSM <= Count_FSM+1;

                IF ( Count_FSM = ( Prescaler_FSM -1 )) THEN

                        Clock_FSM <= NOT Clock_FSM;
                        Count_FSM <= 0;

                END IF;

        END IF;

END PROCESS;
```

Figure 14: ClockDivider_FSM Process

TTC is the time it takes to complete each step, i.e. the time it takes to update each ARR value or to switch from one state to another.

Process DIR Whenever a new position value is given as input, the process will be activated. If the position value is positive, the motor will rotate clockwise; if it is negative, it will rotate counterclockwise. The 'DIR' pin connects the control board to the motor driver and can be either 1 or 0, determining the direction of rotation:

- DIR = 1: clockwise motor
- DIR = 0: counterclockwise motor

The DIR signal is digital and indicates the direction of the motor to the driver, changing the sequence of voltages applied to the coils without altering the shape of the PFM signal.

```
Sel_Dir: PROCESS(h_out)
  BEGIN
          IF(h_out > 0) THEN

              DIR <= '1';

          ELSE

              DIR <= '0';

          END IF;

  END PROCESS;
```

Figure 15: Sel_Dir Proces

Process TimeCalculation: the TimeCalculation process calculates the final time t_f of the trajectory and the acceleration time t_c. It compares these times to determine whether the trajectory is trapezoidal ($t_c \leq \frac{t_f}{2}$) or triangolar ($t_c > \frac{t_f}{2}$). Based on the result, it calculates the required clock cycles for the acceleration, constant speed and deceleration phases in the respective cases. This ensures accurate control of the stepper motor, synchronising the movement with the FPGA clock and respecting the calculated acceleration, constant speed and deceleration times.
In particular, it was calculated:

```
TimeCalculation: PROCESS (clk)

 BEGIN

 IF (rising_edge(clk)) THEN
   -- Time in counts to complete the trajectory:
    ctf <= (((h_out*100000)/V_max) + tc*100000)/TTC;

    IF (tc*10000/TTC) > (ctf/2) THEN  -- tc*1000 to avoid comma numbers

    -- TRIANGULAR VELOCITY TRAJECTOR

      v <= (ctf/2)*A_max; -- triangular wave: v=t*a_max -> t=tf/2

      count_acc_part   <= (ctf/2);
      count_const_part <= count_acc_part+1 ;
      count_dec_part   <= 2* count_acc_part-1;

   ELSE
   -- TRAPEIZODAL VELOCITY TRAJECTOR

       count_acc_part   <= tc*10000/TTC;
       count_const_part <= ctf - count_acc_part;
       count_dec_part   <= ctf;

       v <= V_max;

   END IF;

 END IF;

END PROCESS;
```

Figure 16: ClockDivider_FSM Process

- **tc:** time needed to complete the acceleration phase.
- **count_acc_part:** number of clock strokes needed to complete the acceleration phase.
- **count_const_part:** number of clock strokes needed to complete the constant speed phase
- **count_dec_part:** number of clock strokes needed to complete the deceleration phase and reach the final time

Process Counter FSM: in order to implement the trapezoidal trajectory, it is necessary to use a finite state machine (FSM), which is nothing more than a model that

manages the behaviour of the stepper through a series of transitions between distinct states, corresponding to each phase of the trajectory that depend on certain temporal conditions.

The following figure shows the implemented FSM:

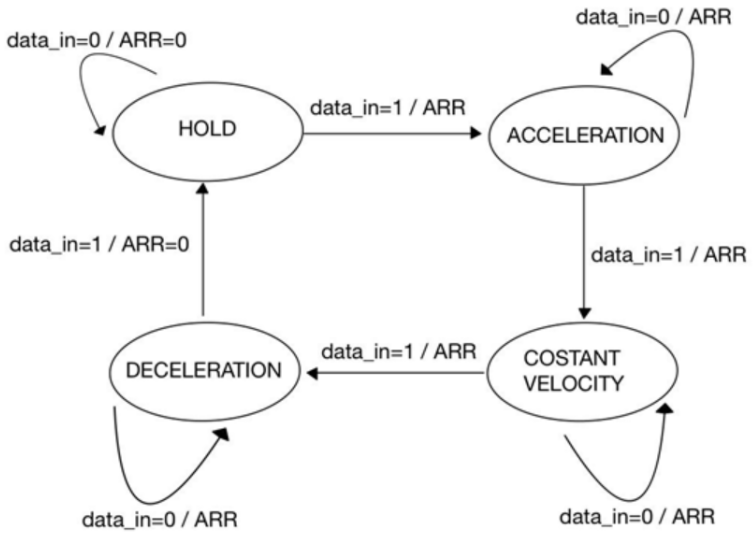

Figure 17: FSM

The CounterFSM process is responsible for managing Timer_trajectory, i.e. a counter to track time and determine which phase of the trapezoidal or triangular trajectory the stepper motor is in, in order to determine the value of the data_in signal, which is used by the FSM process to switch to the next state or remain in the current one.

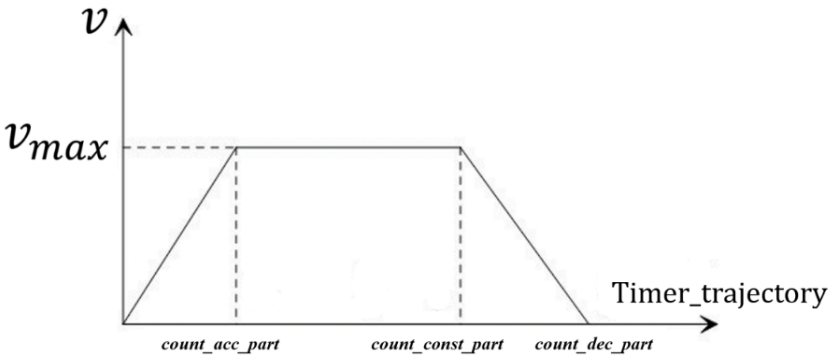

Figure 18: Trajectory

Each state of the FSM identifies a portion of the trajectory to be implemented. The FSM must remain in the current state, according to the calculated time, which guarantees that the portion of the trajectory is correctly completed. To do this it is necessary to verify the previous condition and ensure that the FSM state transition occurs only once per portion of the trajectory. To avoid errors during this crucial phase, state control conditions were introduced. Specifically, the state used is a vector that identifies the state transitions, the Sel_State vector which uniquely identifies the next change of state and

prevents backward changes of state from being implemented.

We identify changes of state as:

- *00:* HOLD;
- *01:* ACCELERATION to CONSTANT VELOCITY;
- *11:* CONSTANT VELOCITY to DECELERATION;

At the start of the FSM the Sel_State signal is initialized to 00 (which identifies the hold part), as soon as the first time update occurs the first time condition will be verified. Once the first condition on time has been verified, we verify a second condition, that is, if Sel_State=”00” , once this condition has been verified; we impose that the Sel_State vector points to the next portion of the trajectory and carry out the change of state by setting data_in equal to 1, this will bring the FSM to the next state. The same operation is carried out for the other two cases. First the time interval is verified and then if the change of state is possible, if this second condition is verified the vector of changes of state is updated and set data_in equal to 1. When the trajectory is finished, reset the FSM by setting Sel_State to the HOLD condition, this will guarantee the possibility of generating a new trajectory correctly.

```
CounterFSM: PROCESS(Clock_FSM)

  BEGIN

  IF (rising_edge(Clock_FSM))THEN

      IF (Step_count*ctf/h_out= 0)THEN

          Timer_trajectory     <= Timer_trajectory + 1;
          Timer_trajectory_old <= Timer_trajectory;

      ELSIF (Step_count*ctf/(h_out+1)> 0) THEN
          Timer_trajectory <= Timer_trajectory_old +((Step_count*ctf)/(h_out+1));
          Timer_trajectory_old <= 0;

      END IF;

     -- Saturation Timer_trajectory
      IF ( Timer_trajectory > ctf) AND (Step_count=(h_out-1)) THEN
          Timer_trajectory <= 0;

      ELSIF ( Timer_trajectory > ctf) AND (Step_count=h_out) THEN
          Timer_trajectory <= 0;

      ELSIF ( Timer_trajectory > ctf) THEN
          Timer_trajectory <= ctf;
      END IF;
```

Figure 19: Start of the CounterFSM Process and assignation of Timer_trajectory

```
data_in<='0';

   IF ( ((Timer_trajectory > 0) AND (Timer_trajectory <=  count_acc_part)) ) THEN

       IF (Sel_State= "00") THEN

           data_in <= '1';
           Sel_State <= "01";

     END IF;


     ELSIF ( ((Timer_trajectory> count_acc_part) AND (Timer_trajectory <= count_const_part)) ) THEN

       IF ((Sel_State = "01") ) THEN

          data_in <= '1';
          Sel_State <= "11";

     END IF;

   ELSIF (  ((Timer_trajectory> (count_const_part+1)) AND (Timer_trajectory  <= count_dec_part-1 )))  THEN

       IF  (Sel_State = "11") THEN

           data_in   <= '1' ;
           Sel_State <= "00";

       END IF;


   ELSIF ((Timer_trajectory > (count_dec_part-1)) ) THEN

           data_in   <= '1' ;
           Sel_State <= "00";
           Timer_trajectory <= 0;

    END IF;

END IF;

END PROCESS;
```

Figure 20: CountFSM Process VHDL implementation

process statereg: this process is responsible for updating the current state (pres_state) to the value of the next state (next_state) at each rising edge of Clock_FSM.

```
statereg:PROCESS(Clock_FSM)
    BEGIN

    IF RISING_EDGE(Clock_FSM)    THEN
            pres_state <= next_state;
    END IF;
END PROCESS statereg;
```

Figure 21: process statereg

In particular, it is necessary to create an enumerative data type called 'state value' which is used to define a set of states that the FSM can assume during stepper control. After that, two signals next_state and pres_state of type state values were declared which respectively indicate the present and current state that the FSM will have to assume. I our case HOLD,ACCELERATION,CONSTANT VELOCITY E DECELERATION.

process FSM: this process determines the next state (next_state) based on the current state (pres_state) and the input signal (data_in), using a CASE structure to handle transitions between states. When data_in is equal to 1 the FSM transitions to the next state otherwise it remains in the current state

```
FSM:PROCESS(pres_state,data_in)
      BEGIN
        CASE pres_state IS

            WHEN HOLD =>
              CASE data_in IS
                  WHEN '0' => next_state <= HOLD;
                  WHEN '1' => next_state <= ACCELLERATION;
                  WHEN OTHERS => next_state <= HOLD;
               END CASE;
            WHEN ACCELLERATION =>
               CASE data_in IS
                  WHEN '0' => next_state <= ACCELLERATION;
                  WHEN '1' => next_state <= CONSTANT_VELOCITY;
                  WHEN OTHERS => next_state <= ACCELLERATION;
              END CASE;

            WHEN CONSTANT_VELOCITY =>
               CASE data_in IS
                  WHEN '0' => next_state <= CONSTANT_VELOCITY;
                  WHEN '1' => next_state <= DECELLERATION;
                  WHEN OTHERS => next_state <= CONSTANT_VELOCITY;
                END CASE;

           WHEN DECELLERATION =>
              CASE data_in IS
                  WHEN '0' => next_state <= DECELLERATION;
                  WHEN '1' => next_state <= HOLD;
                  WHEN OTHERS => next_state <= DECELLERATION;
                END CASE;

            WHEN OTHERS => next_state <= HOLD;
       END CASE;
 END PROCESS FSM;
```

Figure 22: process FSM

process outputs: the outputs process has the task of determining the value of the ARR representing the output of the FSM, based on the current state of the FSM (pres_state), the input signal (data_in) and the time counter (Timer_trajectory). This is necessary because the ARR must be recalculated every time the FSM is clocked.

Variation of ARR during the various phases of the trajectory:

- **Acceleration:** During the acceleration phase, the motor speed increases linearly. This implies that pulses must be generated faster and faster as the frequency increases, so the ARR which determines the period of the triangular carrier, must decrease. $f_{microstep}$ increases from 0 to 533 Hz, so the ARR decreases as timer_trajectory increases, starting from a very high initial value

$$ARR(t) = \frac{f_{clk,FPGA}}{PSC \cdot f_{microstep}(t)} \implies \frac{f_{clk,FPGA}}{PSC \cdot a_{max} \cdot t} \tag{11}$$

- **Constant velocity:** During this phase, the motor maintains a constant speed. As a result, the ARR remains unchanged and corresponds to the desired maximum speed. $f_{microstep}$ remains constant at 533 Hz, so the ARR remains fixed

$$ARR(t) = \frac{f_{clk,FPGA}}{PSC \cdot f_{microstep}(t)} \implies \frac{f_{clk,FPGA}}{PSC \cdot v_{max}} \tag{12}$$

- **Deceleration:** During deceleration, the motor speed decreases linearly. The pulses must be generated more and more slowly, so the ARR must increase. $f_{microstep}$ decreases from 533 Hz to 0, so the ARR increases as Timer_trajectory increases

$$ARR(t) = \frac{f_{clk,FPGA}}{PSC \cdot f_{microstep}(t)} \implies \frac{f_{clk,FPGA}}{PSC \cdot (v_{max} - a_{max} \cdot t)} \tag{13}$$

Each ARR(t) value will be sent to the Frequency Modulator module and will determine the period of the carrier at each instant, and it is in this way that the frequency of the PFM signal can be varied. Can be noticed that in the implementation we use TTC term because the time is discretized and not continuous, the TTC term divides the calculation in constant intervals.

```
outputs: PROCESS(pres_state, data_in, Timer_trajectory)
    BEGIN
    CASE pres_state IS

        WHEN HOLD =>
            IF data_in ='0' THEN

                ARR <= 0;

            END IF;


        WHEN ACCELLERATION =>
            IF data_in = '0' THEN

                IF (A_max*Timer_trajectory = 0) THEN

                    ARR <= 0;

                ELSE

                    ARR <= (CLOCK_FREQ_const/PRESCALER_ARR)*1000/((A_max/10)*Timer_trajectory*TTC);

                END IF;

            END IF;


        WHEN CONSTANT_VELOCITY =>
            IF data_in = '0' THEN

                ARR <= (CLOCK_FREQ_const/PRESCALER_ARR)/v;

            END IF;

        WHEN DECELLERATION =>
            IF data_in = '0' THEN

                IF ( Timer_trajectory > (ctf-1)) THEN
                ARR <= 60000;

                ELSE

                ARR <= (CLOCK_FREQ_const/PRESCALER_ARR)*1000/(ctf*(A_max/10)*TTC-(A_max/10)*TTC*Timer_trajectory);

                END IF;

            END IF;

        WHEN OTHERS =>

             ARR <= 0; -- If we don't Know the status in the indecision DO NOTHING !

        END CASE;


END PROCESS outputs;
```

Figure 23: process outputs

process ARR saturation: during the simulation phase, excessively high ARR values were noted that the counter was unable to achieve in the time available. These ARR values, corresponded to speeds that were too low to be implemented by the motor. Consequently, an upper limit for the ARR was empirically calculated to which corresponded a minimum speed manageable by the motor.

```
PROCESS (clk)
  BEGIN

      IF ( ARR > 60000) THEN -- Saturation of the ARR

        ARR_out <= 60000;

      ELSE

        ARR_out <= ARR;

      END IF;

 END PROCESS ;
```

Figure 24: ARR saturation

4.2.2 Frequency Modulator

The file Frequency_Modultor.vhd modulates the frequency of a PFM signal based on the value of ARR given as input by the component Stepper_Trajectory.vhd . In each clock cycle, it updates a counter (counter_val) and compares this value with CCR (a quarter of ARR). Based on this comparison, it generates a PFM signal at the output. The logic ensures that the PFM signal has a variable frequency, determined by the value of ARR, which is dynamically updated.

Libraries and Entity

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

ENTITY Frequency_Modulator is
    PORT( clk : IN STD_LOGIC;
          ARR : IN INTEGER;
          PWM_out: out STD_LOGIC

    );
END     Frequency_Modulator;
```

Figure 25: Libraries and Entity

The entity has three ports:

- **clk** the input clock signal of the board
- **ARR** the value used for modulation
- **PFM_out** the PFM output signal

Architecture

```
ARCHITECTURE PFM_GEN OF Frequency_Modulator IS

 SIGNAL counter_val   : INTEGER   :=  0 ;
 SIGNAL COUNT_t       : INTEGER   :=  0 ;
 SIGNAL PRESCALER_ARR : INTEGER   := 50 ;
 SIGNAL Clock_PFM     : STD_LOGIC := '0';
 SIGNAL Count_PFM     : INTEGER   :=  0 ;

 SIGNAL CCR     : INTEGER := ARR/4;
 SIGNAL ARR_old : INTEGER := 0;

 SIGNAL First_time : STD_LOGIC := '0';

 BEGIN
```

Figure 26: Architecture and Signal declaration

This section declares the internal signals used in the modulation:

- **counter_val:** counter which updates at each clock stroke for comparison with the CCR
- **COUNT_t:** counter which updates at each clock stroke for comparison with the ARR
- **CCR:** counter which updates at each clock stroke for comparison with the ARR. In our case it is initialized to a value equal to ARR/4 or to a duty cycle of 25%
- **ARR_old:** stores the previous value of ARR

Start of Process and assignment of CCR

```
CMP: PROCESS(Clock_PFM)
BEGIN

  IF (RISING_EDGE(Clock_PFM)) THEN

    IF (First_time = '0') THEN

        CCR <= ARR/4;
        First_time <= '1';
        ARR_old <= ARR;

    ELSE

        CCR<= ARR_old/4;

    END IF;
```

Figure 27: control signal for the first cycle

The 'CMP' (comparison) process is activated at each clock. At each rising edge, the assignment is made. If it is the first time, CCR will depend on the first ARR value passed to the trajectory generation module. from this value. If it is not the first time and the counter (counter_value) is still less than the old ARR value, CCR will depend on ARR_OLD. This ensures that the comparison is always made with the correct ARR, by waiting for the counter to finish counting before comparing it with the new ARR.

```
COUNT_t <= COUNT_t+1;

IF (2*COUNT_t < ARR_old) then -- ARR/2

   counter_val <= counter_val + 1;

END IF;

IF (2*COUNT_t > ARR_old) then -- ARR/2

    counter_val <= counter_val-1;

END IF;

IF (COUNT_t = ARR_old) THEN

    counter_val <= 0;
    COUNT_t <= 0;

    ARR_old <= ARR;

END IF;
```

Figure 28: counter_val change and ARR assignment

The COUNT_t time counter is incremented from time to time and compared with ARR_old/2. If less, counter_val will increment by one unit, vice versa if it is greater. Once the same value is reached, both counters will reset and the reference ARR will be assigned as the old of ARR value, this is so that the same operation can be performed for another value received as input.

The use of ARR_OLD is necessary to avoid that during the comparison, a new ARR value is updated before the counter has reached the reference value.

The logic of these comparisons, arises from the idea of wanting to have a counter_val trend that is triangular in shape as shown in figure:

```
        IF (counter_val < CCR OR counter_val = 0 ) then

            PFM_out <= '0';

        ELSE

            PFM_out <= '1';

        END IF;

    END IF;

END PROCESS;


END PFM_GEN;
```

Figure 29: PFM high to low and end of Process and Architecture

In this portion of the code is implemented the generation of square wave, centered in the triangle wave, with constant duty cycle, which will be used to control the motor. As the value of CCR changes (which in turn depends on ARR), this wave will have a larger or smaller period to which it will subsequently correspond to a speed that changes inversely proportional to the period.

As explained in the past paragraphs, the signal value will change to a high state once counter_val exceeds the threshold value, otherwise it will change to low or remain at low if it is in the initial state.

For clarity, again compare figure 5.

4.2.3 Stepper Motion (Top-level Entity)

The top level entity plays a fundamental role. It represents the main entry point of the project, defining the external interfaces through which the system interacts with the surrounding environment. The top level entity acts as the main container, within which the various components that realise the desired functionality of the system are instantiated and connected.

In particular, the top level entity StepperMotion coordinates the internal components Frequency_Modulator and Stepper_Trajectory, managing the input and output signals to control the stepper motor to execute the desired trajectory.

Libraries and Entity

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

ENTITY StepperMotion IS
    PORT(
        clk             : IN  STD_LOGIC;
        THIS_PFM_OUT    : OUT STD_LOGIC;
        FORCING_DIR_SW  : IN  STD_LOGIC;
        DIR_SW          : IN  STD_LOGIC;
        DIR             : OUT STD_LOGIC

    );
END StepperMotion;
```

Figure 30: Libraries and Entity

The entity has three ports:

- **clk:** Clock signal of the input board.
- **THIS_PFM_OUT:** PFM output signal for controlling the motor.
- **FORCING_DIR_SW:** input signal which if is high, activates a rotation mode in which the motor rotates back and forth after completing rotation. In case it is low, activate the DIR_SW switch.
- **DIR_SW:** Input signal for controlling the direction of the motor via switch.
- **DIR:** Output signal indicating the direction of the motor that is sent to the motor driver.

Architecture

```
ARCHITECTURE BEHAVE OF StepperMotion IS

    COMPONENT Frequency_Modulator IS
        PORT(
            clk : IN STD_LOGIC;
            ARR : IN INTEGER;
            PFM_out : OUT STD_LOGIC
             );
    END COMPONENT;

    COMPONENT Stepper_Trajectory IS

        GENERIC(
            Prescaler_FSM      : INTEGER := 8334;
            TTC                : INTEGER := 16;
            PRESCALER_ARR      : INTEGER := 50;
            CLOCK_FREQ_const   : INTEGER := 50000000;
            V_max              : INTEGER := 100;
            A_max              : INTEGER := 10;
            tc                 : INTEGER := 10
               );
        PORT (
            clk              : IN  STD_LOGIC;
            h_out            : IN  INTEGER; -- distance
            Step_count       : IN  INTEGER;
            ARR_out          : OUT INTEGER
             );
    END COMPONENT;
```

Figure 31: Architecture and components

There are two components:

- **Stepper_Trajectory:** calculates the stepper motor trajectory based on configuration parameters and received inputs
- **Frequency_Modulator:** modulates the frequency of the PWM signal based on the ARR value received by the other component.

Signal used

```
SIGNAL h    : INTEGER := 800; -- 800=90° 1600=180°
SIGNAL ARR  : INTEGER :=   0;

SIGNAL ARR_internal : INTEGER   := 0 ;
SIGNAL PFM_internal : STD_LOGIC :='0';
SIGNAL Count_step   : INTEGER   := 0 ;

SIGNAL Count_step_internal: INTEGER:= 0;

SIGNAL Stop_PFM     : STD_LOGIC := '0';
SIGNAL First_time   : STD_LOGIC := '1';

-- Signals needed to choose direction
SIGNAL DIR_internal: STD_LOGIC := '1';
SIGNAL FORCED       : STD_logic := '0';
SIGNAL CLOCKWISE    : STD_LOGIC := '0';
```

Figure 32: Signals Declaration

- **h:** number of microsteps to apply to the motor so that it performs the desired angle.
- **ARR:** is the value of ARR generated by Stepper_Trajectory
- **ARR_internal** and **PFM_internal:** these signals connect the output of the Stepper_Trajectory component with the input of the Frequency_Modulator component, allowing the frequency of the PFM signal to be modulated according to the calculated trajectory.
- **PFM_internal:** PFM signal which is passed to frequency modulator and assigned to THIS_PFM_OUT.
- **Count_step:** is the counter used to count the steps done by the motor, i.e. is the number of the rising edge of the PFM signal. This value is used by the Stepper_Trajectory component
- **Count_step_internal:** this also counts the steps taken by the motor
- **Stop_PFM:** signal to stop the PFM signal of the motor
- **DIR_internal:** manages the motor direction according to the command received from the DIR_SW signal.
- **FORCED:** signal that if it is 1 it activates the FORCING_DIR_SW switch, otherwise it deactivates it.

- **CLOCLWISE:** signal that if it is 1 it rotates the motor clockwise. Vice versa if it is 0.

Instantiation of Components

```
BEGIN

  c1: Stepper_Trajectory  PORT MAP( clk, h, Count_step, ARR );
  c2: Frequency_Modulator PORT MAP( clk, ARR_internal,PFM_internal );
```

Figure 33: Instantiation of Components

Signals labelled **internal** are those signals, which are used to manage communication between internal components and to implement the logic required for system operation.

```
PROCESS (clk,PFM_internal)
    BEGIN

       IF (count_step_internal >= 0) THEN

        ARR_internal <= ARR;

       ELSE

        ARR_internal <= 0;

    END IF;

END process;
```

Figure 34: Process to update ARR_internal

The process for updating ARR_internal has two main cases:

- **Motor Running:** If the motor is running and the stop condition is not verified, ARR_internal is updated with the latest ARR value, which is required for the frequency modulator
- **Motor Stopped:** If the motor is stopped, ARR_internal is set to zero

```
PROCESS (PFM_internal)
   BEGIN
    IF rising_edge(PFM_internal) THEN
      IF (Count_step_internal> abs(h-1)) THEN
          Count_step_internal <= 0;
          IF ( FORCED = '1') THEN
              DIR_internal <= not DIR_internal;
              ElSIF (FORCED ='0')THEN
                  IF (CLOCKWISE= '1') THEN
                      DIR_internal <= '1';
                  ELSE
                      DIR_internal <= '0';
                  END IF;
          END IF;
      ELSE
      Count_step_internal <= Count_step_internal + 1;
      END IF;
     END IF;
  END Process;
```

Figure 35: Process to update Count_step_internal

The step count process (Count_step_internal) is essential to monitor and control the number of steps performed by the stepper motor. This process uses the PFM_internal signal to determine when to increase the step count. The process for updating Count_step_internal has two main cases:

- **Counter Reset:** when the motor has completed all steps, the counter is reset for reuse.
- **Counter increment:** when the motor has yet to complete the entire trajectory, the counter is incremented with each step taken

```
Sel_Dir: PROCESS(clk)

  BEGIN
    IF( FORCING_DIR_SW = '1' ) THEN

        FORCED <= '1';

       ELSE

         FORCED <= '0' ;

            IF ( DIR_SW = '1') THEN

                CLOCKWISE <= '1';

            ELSE

                CLOCKWISE <= '0';

            END IF;

       END IF;
  END PROCESS;


END BEHAVE;
```

Figure 36: Process for selecting direction

Furthermore, as you can see from the last two images, the following functionality has been implemented: - If the FORCING_DIR_SW Switch is pressed, the stepper is reversed direction at the end of the trajectory. If it is not pressed, it is possible to control a further switch DIR_SW which determines the direction of rotation of the motor depending on whether it is pressed or not.

```
Count_step   <= Count_step_internal;

-- Output of the Entity
THIS_PFM_OUT <= PFM_internal;
DIR <= DIR_internal;
```

Figure 37: Assignment of signals to the outputs of the top level entity

Once all operations have been performed, the updated internal signals are connected to the output ports of the top level entity.

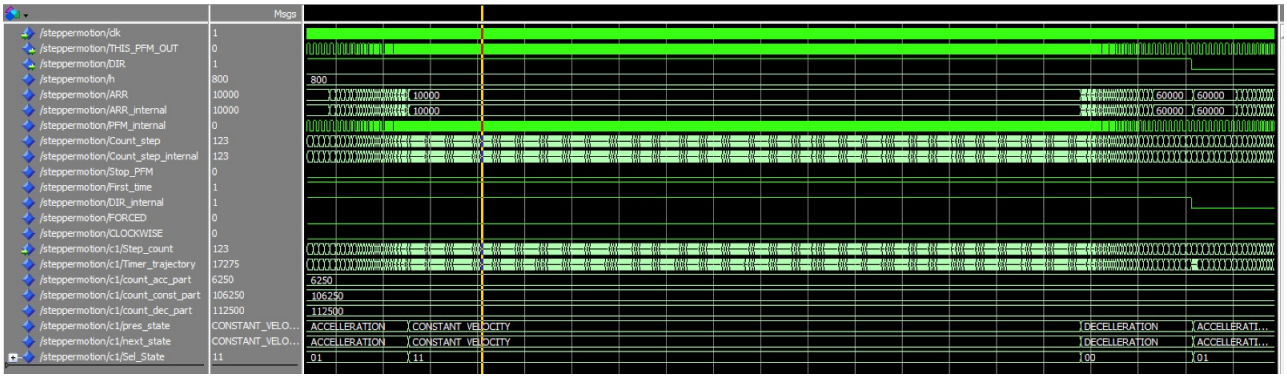

Figure 38: Multisim simulation of our project. In this image can be seen the most important signals used to generate the trajectory

Conclusion and Future Developments

This project was really challenging and has forced us to push our knowledge more and more.The first major challenge encountered was the analytical definition of the trajectory which required a lot of effort to be determined in a scalable manner. Another great challenge was that of the correct timing of the signals having specifications to be respected by the driver it was necessary to determine the most suitable frequency ranges in which to work and how to "orchestrate" all the components in such a way as to generate a repeatable and scalable trajectory.

As can be seen from the image it is possible to notice how the signal we give to the driver is exactly as we expected, in the corresponding phase of the FSM we can identify the PFM signal behavior:

- **ACCELERATION :** the period of the signal is gradually getting shorter so the speed is increasing,
- **CONSTANT VELOCITY:** the period remains constant compared to the last period of the acceleration phase, the speed does not vary,
- **DECELERATION:** starting from the constant period it gradually widens until it reaches zero, the speed decreases until it reaches zero.

With our project we wanted to demonstrate how it was possible to generate real time trajectories using an FPGA, a crucial task in modern robotics systems, which combined with control algorithms and kinematic inversion allows the control of a manipulator. Although very far from a marketable solution, it appears to be a good starting point for future developments, which could include:

- *Increasing the accuracy:* more precise control can be achieved introducing of an encoder on the rotor, which feedback the real position of the rotor shaft.
- *Compensate the load:* introduction of a control loop which could correct errors due to the load torque.
- *Comunication:* introduction of a communication system for the exchange of data and set-points

References

[1] Wang Bangji et al. "Velocity profile algorithm realization on FPGA for stepper motor controller". In: *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*. 2011, pp. 6072–6075. DOI: 10.1109/AIMSEC.2011.6009864.

[2] *DE10-Lite FPGA Datasheet*. https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/Boards/DE10-Lite/DE10_Lite_User_Manual.pdf.

[3] *DRV8825 driver TI official documentation*. https://tinyurl.com/ti-documentation-drv8825.

[4] Chiu-Keng Lai, Wei-Nan Chien, and Yaw-Ting Tsao. "An FPGA-Based Multiple-Axis Velocity Controller and Stepping Motors Drives Design". In: *MATEC Web of Conferences* 71 (2016). Ed. by C.-H. Liu and C.-C. Wang, p. 05002. ISSN: 2261-236X. DOI: 10.1051/matecconf/20167105002. URL: http://dx.doi.org/10.1051/matecconf/20167105002.

[5] Chiu-Keng Lai et al. "Development of an FPGA-Based Motion Control IC for Caving Machine". In: *Advances in Mechanical Engineering* 6 (Jan. 2014), p. 813204. ISSN: 1687-8140. DOI: 10.1155/2014/813204. URL: http://dx.doi.org/10.1155/2014/813204.

[6] *MATLAB*. https://www.mathworks.com/products/matlab.html?s_tid=hp_ff_p_matlab.

[7] Bruno Siciliano et al. *Robotics: Modelling, Planning and Control*. Springer Publishing Company, Incorporated, 2010. ISBN: 1849966346.

[8] *TMC2209 Datasheet*. https://www.analog.com/media/en/technical-documentation/data-sheets/TMC2209_datasheet_rev1.08.pdf.