POLITECNICO DI BARI

DEI DEPARTMENT

MASTER'S DEGREE IN AUTOMATION ENGINEERING

Digital Programmable Systems

Prof. Ing. De Carlo Martino

Project:

Remote Controlled Car by using a FPGA

Students:
Delio Montanaro
Francesco Stasi
Pierfrancesco Timpone
Davide Tonti

Academic Year 2023-2024

Contents

1 Introduction **1**

2 Logic Design **2**

3 Hardware **4**
3.1 Transmission module HC-12 4
3.2 Servomotor Operating Principle 5
3.3 NMOS IRFZ44N 6
3.4 BTS7960 7
3.5 FPGA Max10Lite 8
3.6 Transmitter 9
3.7 Schematic 10

4 VHDL **12**
4.1 UART RX 12
4.2 Prescaler 14
4.3 Preliminary control 15
4.4 Byte divider 16
4.5 PWM DCMotor 17
4.6 DCmot_PWM_cntrl 18
4.7 Servomotor VHDL 20
4.8 Led Control 21

5 3d model and assembly **23**

6 Conclusions **28**

References **29**

1 Introduction

FPGAs are known for their flexibility and ability to be reconfigured to perform any digital function, making them ideal for a wide range of applications in engineering.
The use of a Field Programmable Gate Array (FPGA) such as the MAX10Lite in our design offers a number of significant advantages.
Historically, FPGAs were introduced in the 1980s as a solution to reduce the time and cost associated with designing custom integrated circuits. Since then, their popularity has grown tremendously, especially in signal processing, telecommunications, and, more recently, robotics engineering applications.

The project shown here aims to realize a radio-controlled vehicle driven through the DE10-Lite board.
The project wants to emphasize the board's ability to execute processes suitable for control in a parallel manner. Specifically, it is intended to design a vehicle with two steered wheels on the front axle and two driven wheels on the rear axle.

The expected difficulties are:

- Selection of all physical components
- Implementation of the radio module and its communication protocol
- Processing of received and transmitted data
- Realization of control signals
- Practical realization of the prototype.

2 Logic Design

The code was developed starting from a general logical scheme, first a written draft and then gradually expanded until we reached the goals we set for the project.
This Diagram can be summarized downstream of corrections such as the figure 1 below:

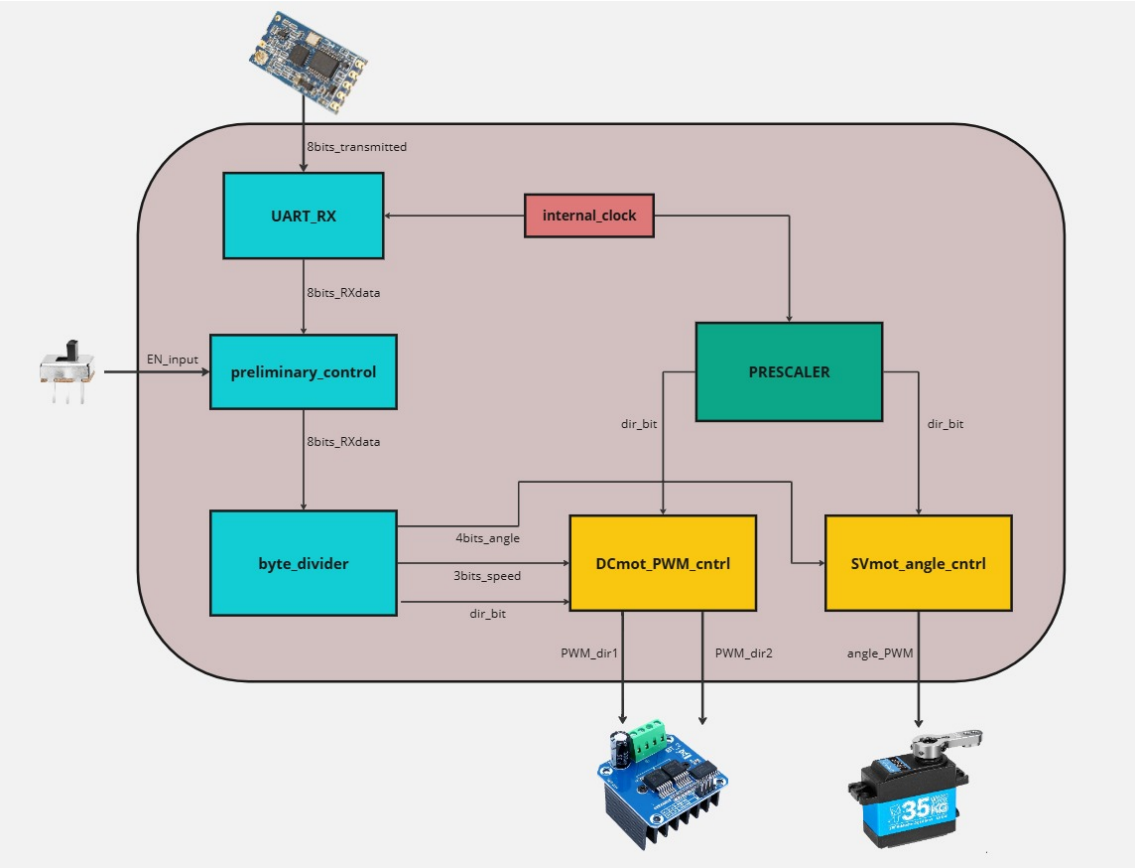

Figure 1: Logic Concept Schematic

This diagram represents in a fairly concise way the operation of the whole:

- A UART protocol that receives a data stream as input
- A preliminary control that will be used to perform a specific function at the setup
- Finally a separation of the data that will go to the motor and servo

Almost all these steps are synchronized with different clocks starting from the internal one. The implementation of this logical schema has been facilitated by the possibility of creating a schematic file directly within the IDE, which will be roughly the same.
In fact, here is the schematic file completed at the end of the work in the figure 2:

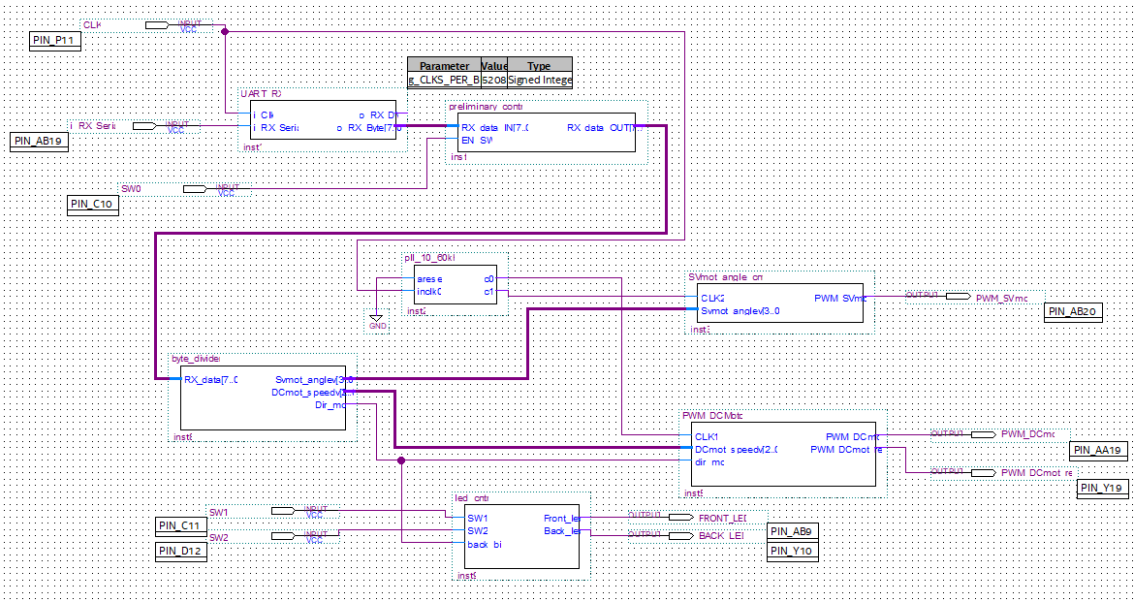

Figure 2: Final Logic File Schematic

3 Hardware

This section briefly discusses the hardware used to carry out the project with some explanations on how it works to examine the points on which the most focus has been given.

3.1 Transmission module HC-12

The transmission module used for our purposes is the HC-12 (fig.3), a wireless communication device that operates in the ISM (Industrial, Scientific, and Medical) frequency band at 433 MHz. The HC-12 module offers various operational modes, including FU1, FU2, FU3, and FU4, which allow for balancing transmission speed, range, and power consumption. For our specific project, the FU1 mode at 9600 bps was chosen primarily because a speed of 9600 bps is considered acceptable for a remote control and sufficient to transmit commands and data in real-time without noticeable delays to the user.

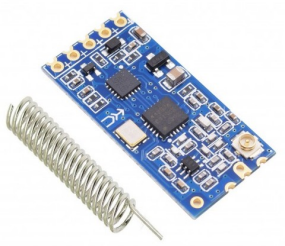

Figure 3: HC-12 module

Furthermore, a higher transmission speed would require greater power consumption, both in transmission and reception. Additionally, the FU1 mode allows for a good transmission range without significantly degrading the signal. Increasing the transmission speed would reduce the effective range of the radio signal, which is undesirable for a remote control that must operate over reasonable distances. All this ensures, finally, that the commands sent by the remote control are received accurately and without data loss.
A configuration code was set up with Arduino to communicate with the HC-12 module via a software serial port, allowing the device to receive and send data between the module and the computer. In this code, it was first necessary to include the SoftwareSerial library, which enables the creation of a software serial port on Arduino's digital pins. This is useful when using pins other than those dedicated to Arduino's hardware serial port. Next, a new software serial port called 'HC12' was defined using pin 10 as TX (transmission) and pin 11 as RX (reception). This establishes a connection between Arduino and the HC-12 module for serial communication initialized at a baud rate of 9600, for the reasons specified above. In the main loop, two 'while' loops check for available data from the HC-12 module or the computer's serial monitor, and if successful, they send it to the receiving terminal.

In terms of hardware, the pins of interest of HC-12 module are:

1. **VCC**: Power supply at 3.2V - 5.5V. Provides the necessary power for the module to function correctly, in our case connected to the **5V** pin of the Arduino Uno Nano.
2. **GND**: Ground reference, necessary to complete the circuit, connected to the **GND** pin of the Arduino Uno Nano.
3. **TXD**: Data transmission pin, connected to the **D10** pin of the Arduino Uno Nano.
4. **RXD**: Data reception pin, connected to the **D11** pin of the Arduino Uno Nano.
5. **SET**: Pin for configuring the module, left disconnected in our case.

3.2 Servomotor Operating Principle

For this project we used initially a Servomotor MG996R, so we developed the VHDL code based on this, once we calculated the right value for the counter and the clock for our purposes, also considering the specific case of initial position at 0°, then we moved for performance needs to another engine of the same type that would allow a wider range of motion.

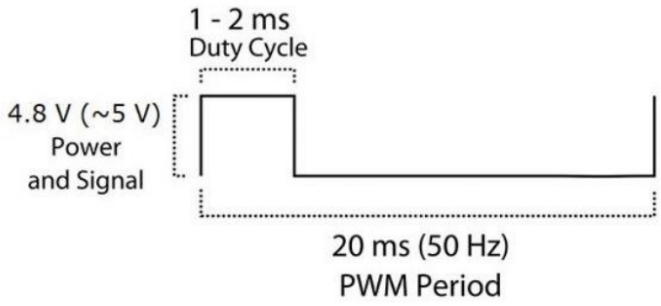

Figure 4: Servomotor PWM

The figure 4 describes how Servo PWM works. Basically the working principle is divided into 3 parts:

- Hold the output high for un amount of time
- A PWM that can be of a variable range, in our case just a combination of 4 bit minus 0°
- Finally low output for the rest of the period

For our first Servo the PWM range was between 1-2 ms and can work at most +-45°, this has been replaced with a "MS61" Miuzei Servo 35 kg (in fig. 5) which can allow a +-135° of motions, an higher torque, it can be powered at higher voltages but with a wider range of pwm but with the same operation; also in this case there mechanical problems and we limited the range to +-51° by software.

Figure 5: Miuzei Servo

3.3 NMOS IRFZ44N

We tried to create a first circuit for motor management
in the four quadrants. Therefore, the use of a circuit such
as the h-bridge with MOSFETs capable of being opened and closed with a certain frequency is necessary. For this purpose, we plan to use an initial circuit made on a stripboard board as shown in the figure 6, with NMOS IRFZ44n and optoisolators for board protection. Although these MOSFETs can withstand current under load (we get to values above 10A under full load) they tend to overheat easily, and not having an adequate dissipation system they are unusable.

Although the circuit devised therefore appears to work, in practice we opt for a solution that integrates a ready-made system for our purpose.

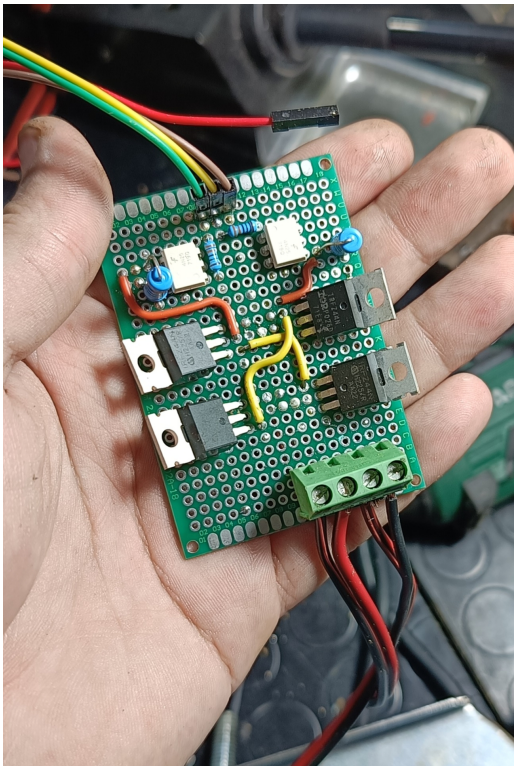

Figure 6: Old DIY H-bridge

To limit the waste of material, these MOSFETs are used to power ancillary devices who are less current demanding and who use 16v power supply with the use of FPGA GPIOs.

3.4 BTS7960

To control the DC motor, we decided to purchase the BTS7960 module (refer to fig. 7). The BTS7960 module is a motor driver that stands out for its ability to handle heavy loads and its flexibility of use. It is designed to control DC motors and can handle a very high current, up to 43A (in our case we also reach peaks of more than 10A), making it ideal for applications that require a large driving force. It offers seamless compatibility with microcontrollers, thanks to the integrated driver IC. The versatile IC comes with various features, including logic-level inputs, current sensing diagnostics, slew rate adjustment, dead time control, and robust protection mechanisms against overtemperature, overvoltage, undervoltage, overcurrent, and short circuits.

Using this module allowed us to expedite the project implementation as we no longer needed to design and create the cooling system for the h-bridge ourselves. This module was selected also because it enables motor control using two signals: Forward PWM for forward motion and Reverse PWM for reverse motion.

Figure 7: bts7960 integrated circuit

3.5 FPGA Max10Lite

In our project, we utilized the DE10-Lite board based on the Altera MAX 10 produced by Terasic (refer to fig.8). The most appreciated feature concerns the integrated FPGA MAX 10 10M50DAF484C7G, of which some details are reported below. DE10-lite board also includes an accelerometer and a 4-bit resistor for specific applications. The board guarantees 64 MB of SDRAM with a 16-bit data bus and features an on-board USB Blaster (type B USB) for programming and debugging purposes. The board includes specific status LEDs to indicate its state.

As known, an FPGA (Field-Programmable Gate Array) is a programmable electronic device composed of an array of configurable logic blocks and programmable interconnections that allow the creation of customized digital circuits. Unlike microprocessors, which execute software instructions, FPGAs are designed to directly implement user-specified hardware behavior.

Specifically, the integrated FPGA MAX 10 (10M50DAF484C7G) used in our project has approximately 50,000 logic elements (LE) and an integrated analog-to-digital converter (ADC). It supports on-chip memory blocks and offers high performance due to its programmable logic cells. It provides configurable digital signal processing (DSP) blocks.

It features an SRAM Memory Architecture with configurable SRAM memory blocks that can be used to implement on-chip RAM for temporary data storage. The FPGA MAX 10 has an internal oscillator that can generate clock signals, useful for synchronizing components within the FPGA.

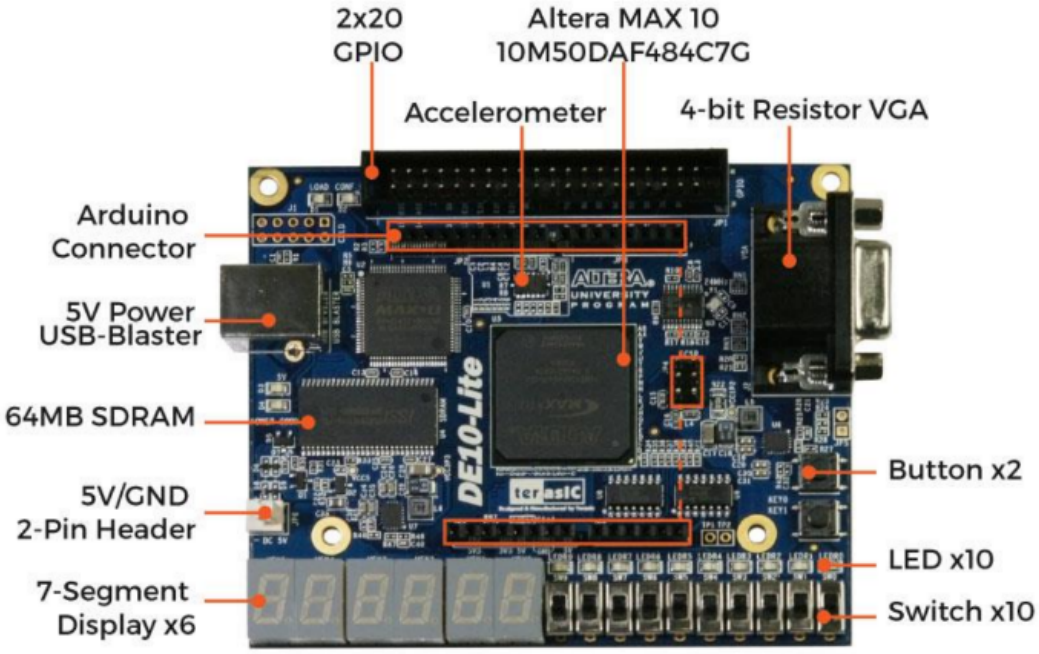

Figure 8: Fpga Max 10 lite

3.6 Transmitter

For the transmitter we use a simple solution with the use of an Arduino nano (fig. 9), which allows a small space footprint and programming simplified by the presence of the different libraries. We use two analog inputs to read the two different directions of the analog (used to control the radio-controlled machine). One of the two HC12 radio modules is then connected for signal transmission.

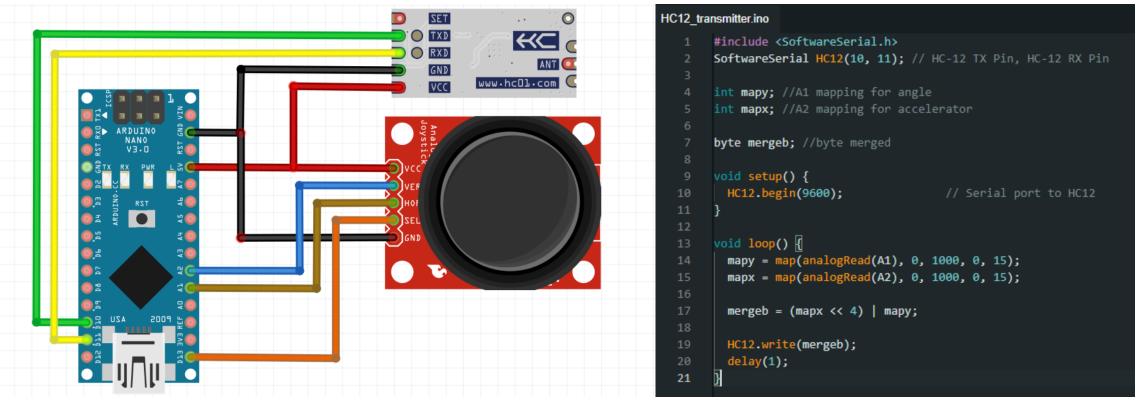

Figure 9: Arduino Transmitter

In the code we include a single library "softwareSerial [1]" that can simulate UART communication on the two GPIOs where the radio module is connected. The radio module and the serial are initialized at 9600 baud: we do not need an excessive baud rate, so reducing it will allow us to extend the transmission range. In the remaining part of the code we make a mapping between the two analog values in the range it covers (0 to 1024), keeping a margin on the extremes that may take an unstable value. Thus we get two values represented by 4 bits, one indicating the direction and speed of the car, the other indicating its steering angle. To create the 8-bit composition to send through the UART, we make the union of these two values. To the FPGA that receives the data on the UART awaits the task of decomposing the 8 bits and reinterpreting them to implement the control.
The final show-case of the transmitter is the following figure 10 with a proper 3d model case:

Figure 10: Final Result of the Transmitter

3.7 Schematic

The various attempts and studies finally led us to the schematic shown in the figure 12 using the Software Kikad [2] . The whole is powered by a battery with nominal 16V that will power the DC motor for vehicle traction through the BTS7960 module.

The remaining devices, including the DE 10lite board, the servo motor for vehicle steering, and the radio module need to be powered at a lower voltage. Therefore, a DC-DC converter is inserted that stabilizes the input voltage to 5V. A device is chosen that can provide up to 5A output, which is more than sufficient for this purpose.

The BMS (battery management system) is able to charge the 4 18650 batteries placed in series only if the voltage does not fall below a threshold value (Approximately 14.6V). Since there is no installation of a device that automatically blocks the power supply near the threshold value, it is necessary to monitor the voltage with a built-in voltmeter. A master switch allows the system to be turned on and off.

The FPGA must handle the PWM signals for motor control, as well as the radio module and MOSFETs for front and rear headlight switching. The signal management is done on-site and in a modular manner with the creation of a shield (fig.11) to ensure that the connections are firm during movement.

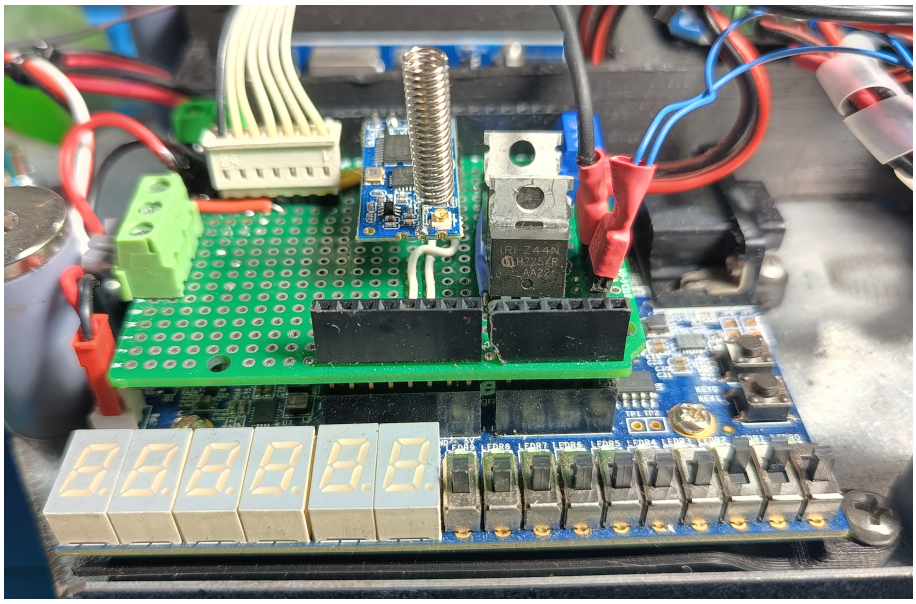

Figure 11: custom shield for De10-Lite Board

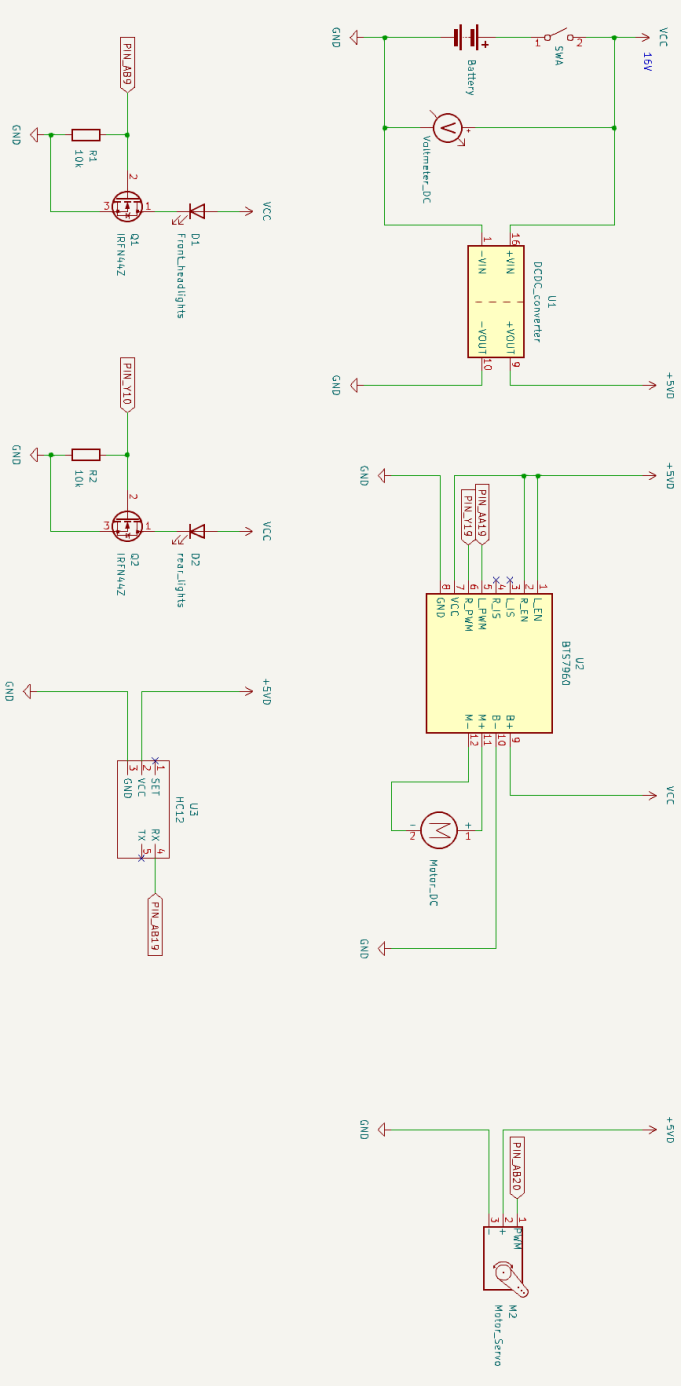

Figure 12: Schematic

4 VHDL

The VHDL is the programming language that we used in combination with the Quartus Pime Ide [3]. For a better understanding at a functional level the code has been developed into several "components" and link to each other through a schematic file (see fig. 2). This is usefull because you don't need to mapping each input to the corresponding output, simulate each component separately (using the University Program VWF in the Functional way) and have a overall view of the program. For clarity, the code was uploaded directly to the device's flash memory through the .pof file [4]. The VHDL components used are explained below.

4.1 UART RX

The chosen radio module uses the UART communication protocol. Therefore, the first step is to implement such control on the board. Fortunately, some open source libraries are available online. Among those most compatible with our needs we find the one published by nandland [5].

In our case the FPGA will only have to receive a signal, and not transmit it, this allows us to use only the part for receiving the message, remembering that only the transmission is delegated to the transmitter driven by the Arduino nano. This suggests that ours is an open-loop control, but nothing prohibits a future implementation that allows closed-loop control with bidirectional communication.

Data transmission occurs starting with a start bit that signals to receiver the start of a new communication. Immediately following the start bit follows, bit by bit, the transmission of the data (usually consisting of 8 bits as in our case). An optional parity bit may indicate the integrity of the data sent. The transmission ends with the stop bit, which allows the receiver to understand the conclusion of the transmission.

Receipt takes place in a similar way following the reading of the start bit that signals the beginning of the communication which is followed by the reading of the 8 bits that make up the data. The distop bit indicates the conclusion of the transmission.

The baud rate of both UART devices must be set to be about the same to ensure reliable communication. A discrepancy of more than 10% between baud rates can lead to errors in bit timing.

We next analyze the downloaded UART library that enables reception.

```
1  library ieee;
2  use ieee.std_logic_1164.ALL;
3  use ieee.numeric_std.all;
4
5  entity UART_RX is
6    generic (
7      g_CLKS_PER_BIT : integer := 5208     -- clock DE 10 lite/baundrate : ...
           50000000/9600
8      );
9    port (
10     i_Clk       : in  std_logic;
11     i_RX_Serial : in  std_logic;
12     o_RX_DV     : out std_logic;
13     o_RX_Byte   : out std_logic_vector(7 downto 0)
14     );
15 end UART_RX;
```

The UART_RX entity defines the UART receiver interface.
The generic parameter g_CLKS_PER_BIT indicates the number of clock cycles per bit, which is calculated from the desired clock frequency and baud rate. The ports are the input and output signals of the component: i_Clk is the clock signal, i_RX_Serial is the incoming serial signal, o_RX_DV (Data Valid) indicates when the received data is valid, and o_RX_Byte is the byte of data received.

```
architecture rtl of UART_RX is

  type t_SM_Main is (s_Idle, s_RX_Start_Bit, s_RX_Data_Bits,
                     s_RX_Stop_Bit, s_Cleanup);
  signal r_SM_Main : t_SM_Main := s_Idle;
```

The rtl architecture of the UART_RX entity is initialized, where the behavior of the UART receiver will be described. Here an enumerated type t_SM_Main is defined to represent the various states of the finite state machine (FSM) of the UART receiver. The FSM has five states: s_Idle, s_RX_Start_Bit, s_RX_Data_Bits, s_RX_Stop_Bit, and s_Cleanup. The r_SM_Main signal keeps track of the current state of the FSM.

```
  signal r_RX_Data_R : std_logic := '0';
  signal r_RX_Data   : std_logic := '0';

  signal r_Clk_Count : integer range 0 to g_CLKS_PER_BIT-1 := 0;
  signal r_Bit_Index : integer range 0 to 7 := 0;  -- 8 Bits Total
  signal r_RX_Byte   : std_logic_vector(7 downto 0) := (others => '0');
  signal r_RX_DV     : std_logic := '0';
```

r_RX_Data_R and r_RX_Data are used to record the incoming serial signal to remove problems caused by metastability. r_Clk_Count is a counter used to keep track of the number of clock cycles for each bit. r_Bit_Index keeps track of the index of the current bit while receiving data bits. r_RX_Byte is a vector that stores the byte of data received. r_RX_DV is a signal that indicates when data is valid.
After configuring the signals and defining the state machine, the code proceeds with two main processes: p_SAMPLE and p_UART_RX.

After configuring the signals and defining the state machine, the code proceeds with two main processes: p_SAMPLE and p_UART_RX.

```
 p_SAMPLE : process (i_Clk)
  begin
    if rising_edge(i_Clk) then
      r_RX_Data_R <= i_RX_Serial;
      r_RX_Data   <= r_RX_Data_R;
    end if;
  end process p_SAMPLE;
```

The p_SAMPLE process is responsible for sampling the incoming serial signal. It uses a double register to reduce metastability, which can occur when a signal changes state near the receiver clock edge.

```
p_UART_RX : process (i_Clk)
begin
  if rising_edge(i_Clk) then
    case r_SM_Main is
      when s_Idle =>
        -- Resets the signals and waits for the start bit
```

```
7           --
8         when s_RX_Start_Bit =>
9           -- Check the center of the start bit
10          --
11        when s_RX_Data_Bits =>
12          -- Samples the data bits and stores them
13          --
14        when s_RX_Stop_Bit =>
15          -- Attende il bit di stop
16          --
17        when s_Cleanup =>
18          -- Cleaning and preparation for the next byte
19          --
20        when others =>
21          r_SM_Main <= s_Idle;
22      end case;
23    end if;
24  end process p_UART_RX;
```

The p_UART_RX process is the heart of the state machine. It manages the transition between states based on the clock signal and input data. Here is what it does in each state:

- `s_Idle`: This is the wait state. The system remains in this state until it detects a start bit (a low level in the serial signal).
- `s_RX_Start_Bit`: Once the start bit is detected, the system waits until it reaches the middle of the bit to confirm that it is indeed a start bit.
- `s_RX_Data_Bits`: In this state, the system samples the data bits. It uses the r_Clk_Count counter to wait for an integer bit before sampling the data. The sampled data is stored in r_RX_Bytes.
- `s_RX_Stop_Bit`: After receiving all the data bits, the system waits for the stop bit. This bit should be high, indicating the end of the byte transmission.
- `s_Cleanup`: This is a cleanup state that prepares the system to receive the next byte. Set r_RX_DV to '1' to indicate that the byte has been received correctly and can be read.

At the end of the p_UART_RX process, the signals o_RX_DV and o_RX_Bytes are assigned to the respective signals registered within the architecture, making the received data available outside the UART_RX component.

4.2 Prescaler

We needed a prescaler which divided the clock of 50Mhz into subclocks of 10kHz for the DC motor and 60kHz for the Servomotor in our case, so we used the Altera Phase-Locked Loop (Altera PLL) IP Core [6] without the need to add a clock divider ourselves because it's already present in the Quartus Prime IDE.

4.3 Preliminary control

This code is useful to avoid the car from starting to run and rotate immediately after being turned on. This situation occurs because the UART module has a initial setup of "11111111" which in our case correspond to max velocity and +51°. So to avoid this, initially the switch is off, then we turn on the car that has now the correct initial setup finally the enable switch is turned on (in this moment or after linking the transmitter).

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_unsigned.ALL;
4
5
6  ENTITY preliminary_control IS
7      PORT (
8
9          RX_data_IN : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
10         RX_data_OUT : OUT STD_LOGIC_VECTOR(7 DOWNTO 0); --to the bytedivider
11         EN_SW : IN STD_LOGIC --enable switch
12
13     );
14 END preliminary_control;
```

The first three lines of code specify the use of library IEEE. Entity is composed by:

- `RX_data_IN`: input vector coming from transmitter;
- `RX_data_OUT`: output vector going to the bytedivider;
- `EN_SW`: security switch;

```
1  ARCHITECTURE preliminary_control_arch OF preliminary_control IS
2
3  BEGIN
4
5      assign : PROCESS (EN_SW)
6      BEGIN
7
8          IF EN_SW = '0'  THEN
9              RX_data_OUT ≤ "01110111"; --FORCE TO 0 THE RX OUTPUT WHEN DISABLED
10         ELSE
11             RX_data_OUT ≤ RX_data_IN;
12         END IF;
13     END PROCESS assign;
14
15
16 END preliminary_control_arch;
```

In the process, the status of `EN_SW` is checked. If is "0" (switch turned off) the stop command is sent to the bytedivider otherwise, `RX_data_OUT` is sent.

4.4 Byte divider

We encoded the information of motion using 4 bits for the steering angle, 3 bits for the velocity of the car and 1 for the direction. The following code allows the FPGA to extract these information.

```
1      LIBRARY ieee;
2      USE ieee.std_logic_1164.all;
3
4      ENTITY byte_divider IS
5          PORT(RX_data: IN STD_LOGIC_VECTOR(7 DOWNTO 0);
6               DCmot_speedv: OUT STD_LOGIC_VECTOR(2 DOWNTO 0);
7               Svmot_anglev: OUT STD_LOGIC_VCTOR(3 DOWNTO 0);
8               Dir_mot: OUT STD_LOGIC);
9      END byte_divider;
```

The first two lines of code specify the use of library IEEE to handle logic data types. Entity is composed by:

- `RX_data`: 8-bit input vector received by the transmitter. It contains all information about steering angle, motor speed and direction;
- `DCmot_speedv`: 3-bit output vector containing information about steering angle;
- `Svmot_anglev`: 4-bit output vector containing information about motor speed;
- `Dir_mot`: output bit coding the direction (1 means forward, 0 means backward);

```
1  ARCHITECTURE byte_divider OF byte_divider IS
2  BEGIN
3      PROCESS(RX_data)
4      BEGIN
5      Dir_mot ≤ RX_data (7);
6          DCmot_speedv ≤ RX_data(6 DOWNTO 4);
7          Svmot_anglev ≤ RX_data(3 DOWNTO 0);
8      END PROCESS;
9  END byte_divider;
```

The `byte_divider` architecture consists of a single process. This process assigns the most significant bit to the output `Dir_mot`, the second, third, and fourth bits to the output vector `DCmot_speedv`, and the fifth, sixth, seventh, and eighth bits to the output vector `Svmot_anglev`, based on the chosen encoding.

4.5 PWM DCMotor

The following VHDL code defines a PWM controller for a direct current (DC) motor with speed and direction control, using the 4 bits received from the byte divider component. As usual, it uses the IEEE libraries to handle logic data types and arithmetic operations on logic vectors. The definition of the entity `PWM_DCMotor` includes inputs for the clock `CLK1`, the motor speed `DCmot_speedv`, which is a 3-bit vector, and the motor direction, which is the bit `dir_mot`. The outputs defined in the entity are two PWM signals: `PWM_DCmot` for normal control and `PWM_DCmot_rev` for reverse control.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3  USE ieee.std_logic_unsigned.ALL;
4
5  ENTITY PWM_DCMotor IS
6      PORT (
7          CLK1 : IN STD_LOGIC;
8          DCmot_speedv : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
9          dir_mot : IN STD_LOGIC;
10         PWM_DCmot : OUT STD_LOGIC;
11         PWM_DCmot_rev : OUT STD_LOGIC
12     );
13 END ENTITY PWM_DCMotor;
```

The architecture `driverdc_behave` includes an internal component `DCmot_PWM_cntrl` that generates the PWM signal based on the speed bits. There are two instances of this component:

- `U_rev` which generates an inverted PWM signal
- `U0` which generates a normal PWM signal

```
1  ARCHITECTURE driverdc_behave OF PWM_DCMotor IS
2      COMPONENT DCmot_PWM_cntrl IS
3          PORT (
4              CLK1 : IN STD_LOGIC;
5              DCmot_speedv : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
6              PWM_DCmot : OUT STD_LOGIC
7          );
8      END COMPONENT;
9
10     SIGNAL PWM_signal_U0 : STD_LOGIC;
11     SIGNAL PWM_signal_U_rev : STD_LOGIC;
12
13 BEGIN
14     U_rev: DCmot_PWM_cntrl
15         PORT MAP (
16             CLK1 => CLK1,
17             DCmot_speedv => not DCmot_speedv,
18             PWM_DCmot => PWM_signal_U_rev
19         );
20
21     U0: DCmot_PWM_cntrl
22         PORT MAP (
23             CLK1 => CLK1,
24             DCmot_speedv => DCmot_speedv,
25             PWM_DCmot => PWM_signal_U0
26         );
```

Within the process, the control of which PWM signal to use is defined based on the motor direction (`dir_mot`):

- If `dir_mot` is `0`, it enables the inverted PWM signal (`PWM_DCmot_rev`) and disables the normal PWM signal (`PWM_DCmot`).
- If `dir_mot` is `1`, it enables the normal PWM signal (`PWM_DCmot`) and disables the inverted PWM signal (`PWM_DCmot_rev`).

```
    PROCESS (CLK1)
    BEGIN
        IF rising_edge(CLK1) THEN
            IF dir_mot = '0' THEN
                PWM_DCmot ≤ '0';
                PWM_DCmot_rev ≤ PWM_signal_U_rev;
            ELSE
                PWM_DCmot_rev ≤ '0';
                PWM_DCmot ≤ PWM_signal_U0;
            END IF;
        END IF;
    END PROCESS;
END driverdc_behave;
```

To summarize the purpose of this VHDL code, it controls a DC motor by generating a PWM signal to regulate the speed and switching the PWM signal to reverse the motor's direction.

4.6 DCmot_PWM_cntrl

The PWM (pulse width modulation) is used to control the DC motor using the 3 bit received by the byte divider component. The PWM is a technique used to control the power applied to a motor by varying the width of voltage pulses instead of continuously applying a fixed voltage, PWM rapidly switches the voltage on and off. The wider the pulse (higher duty cycle), the more average voltage reaches the motor terminals therefore the motor rotates faster. By adjusting the duty cycle, we effectively control the motor's speed.

```
IF rising_edge(CLK1) THEN
            counter ≤ counter + '1';
            IF counter = "110" THEN
                counter ≤ "000";
            END IF;
            IF counter < DCmot_speedv THEN
                PWM_DCmot ≤ '1';
            ELSE
                PWM_DCmot ≤ '0';
            END IF;

        END IF;
        }
```

The above VHDL code is then used for the creation of the PWM. A counter has been used that increases with each clock cycle and compares its current value with the speed we want the motor to assume(DCmot_speedv) using the 3 bits for a total of 7 speeds excluding the first one with the engine off. This type of PWM allows you to have a speed ranging from engine off to maximum engine speed. If you want to limit the speed range to make better use of the 7 levels only by eliminating those levels for which the motor will not start for less than resistant starting torque or limit the maximum speed in case it is too high, you can change the code to limit the pwm in a certain range of the total period. This has been done in the following way:

```
1
2  ARCHITECTURE driverdc_behave OF DCmot_PWM_cntrl IS
3
4
5      SIGNAL counter : STD_LOGIC_VECTOR(14 DOWNTO 0) := (OTHERS => '0');
6      SIGNAL innercounter : STD_LOGIC_VECTOR(2 DOWNTO 0) := "000";
7      constant A_position: integer:= 9100;
8      constant B_position: integer:=18200;
9      signal flag: integer:=1;
10
11 BEGIN
12     PROCESS (CLK1)
13     BEGIN
14
15         IF rising_edge(CLK1) THEN
16
17             counter ≤ counter + '1';
18             IF counter = "100111000011111" THEN
19                 counter ≤ (OTHERS => '0');
20             END IF;
21
22
23                 IF DCmot_speedv="000" then
24                 PWM_DCmot≤'0';
25                 ELSE
26
27                     IF counter < A_position THEN
28                          PWM_DCmot ≤ '1';
29
30                     ELSIF counter ≥ B_position THEN
31                          PWM_DCmot ≤ '0';
32
33                     ELSE
34                          IF flag=1 THEN
35                              innercounter ≤ innercounter + '1';
36                              flag≤(B_position-A_position)/7;
37                              IF innercounter = "110" THEN
38                                   innercounter ≤ "000";
39                              END IF;
40                              IF innercounter < DCmot_speedv THEN
41                                   PWM_DCmot ≤ '1';
42                              ELSE
43                                   PWM_DCmot ≤ '0';
44                              END IF;
45
46                          ELSE
47                              flag≤flag-1;
48                          END IF;
49                     END IF;
50
51
52                 END IF;
53          END IF;
54
55
56     END PROCESS;
57 END driverdc_behave;
```

In this case the speed has been limited to 45-91%, the period is divided into 20k sections and 2 setpoint were used to set the range. Until the A_position the pmw output is set to 1, then the pwm has been performed and after all the output is set to 0. The flag has the scope to hold the previous bit output for several range. Note that it has been used a 20kHz but it has to be modified for the specific application and of course for the levels of velocity and setpoints.

4.7 Servomotor VHDL

As mentioned the operating principle in sec. 3.2 the VHDL is so developed:

```
1     ARCHITECTURE servomotor_behave OF SVmot_angle_cntrl IS
2
3     SIGNAL counter : STD_LOGIC_VECTOR(10 DOWNTO 0) := (OTHERS => '0'); --create a ...
          counter of 11 bits for divide the 20ms into 1200sections
4     SIGNAL innercounter : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";  --second ...
          counter for the inner pwm generation
5      constant A_position: integer:= 74; --75 sections (from 0)
6      constant B_position: integer:=90; --setpoint for the angle 0
7      constant C_position: integer:=105;--105-75=30/2=15 levels (from 0)
8      signal flag: std_logic:='1';
9
10 BEGIN
11    PROCESS (CLK2, Svmot_anglev)
12    BEGIN
13
14        IF rising_edge(CLK2) THEN
15
16            counter ≤ counter + '1';
17            IF counter = "10010101111" THEN
18                counter ≤ (OTHERS => '0');
19            END IF;
20
21                IF Svmot_anglev="1000" or Svmot_anglev="0111" then
22
23                    IF counter <("000000000" + B_position) then
24                        PWM_SVmot ≤ '1';
25                    IF counter≥("000000000"+B_position) then
26                        PWM_SVmot ≤ '0';
27                    end if;
28
29                ELSIF counter ≤ ("000000000" + A_position) THEN
30                        PWM_SVmot ≤ '1';
31
32
33                ELSIF counter ≥ ("000000000" +C_position) THEN
34
35                        PWM_SVmot ≤ '0';
36
37                ELSE
38
39                        IF flag='0' THEN
40                        flag≤'1';
41
42                        ELSE
43
44                            innercounter ≤ innercounter + 1;
45                            flag≤'0';
46                            IF innercounter = "1110" THEN
47                                innercounter ≤ "0000";
48
49                            END IF;
50
51                            IF innercounter < Svmot_anglev THEN
52                                 PWM_SVmot ≤ '1';
53                            ELSE
54                                 PWM_SVmot ≤ '0';
55                            END IF;
56                        end if;
57                END IF;
58
```

```
59
60          END IF;
61      END PROCESS;
62  END servomotor_behave;
```

The code has A, B and C position to understand the points in which act the 3 flows of data to control the output. This has been done in order to modulate the range with other servo and control bits; of course is important to change the counter too for the max available rotation. For this code in particular, the output is set to high for 1.25 ms, is set to low after the 1.75 ms and a variable PWM between them which has the same behavior as PWM on the DCmot.

4.8 Led Control

The following code is used to turn on or turn off the front and back LEDs.

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL; --for the std_logic struct
3
4  ENTITY led_cntrl IS
5      PORT (
6          SW1 : IN STD_LOGIC;
7          SW2 : IN STD_LOGIC;
8          back_bit: in STD_LOGIC;
9          Front_led : OUT STD_LOGIC := '0';
10         Back_led: OUT STD_LOGIC := '0'
11
12     );
13 END ENTITY led_cntrl;
```

The first two lines of code specify the use of library IEEE to handle logic data types. Entity is composed by:

- `SW1`: input bit controlling the back led;
- `SW2`: input bit controlling the front led;
- `back_bit`: input bit coding motion direction (0 means forward, 1 means backward);
- `Front_led`: output bit controlling the front led;
- `Back_led`: output bit controlling the back led;

```
1  ARCHITECTURE led_cntrl_b OF led_cntrl IS
2  SIGNAL STATE: STD_LOGIC_VECTOR(1 DOWNTO 0);
3  BEGIN
4      PROCESS (SW1,SW2,back_bit)
5      BEGIN
6              STATE<=SW1&SW2;
7
8              case STATE is
9              when "01"=>
10                 Front_led<='1';
11                 Back_led<='0';
12             when "10" =>
13                 Front_led<='1';
14                 if (back_bit='1') then
15                     Back_led<='1';
16                 else
17                     Back_led<='0';
18                 end if;
```

```
19              when "11" =>
20                  Front_led<='1';
21                  Back_led<='1';
22              when OTHERS =>
23                  Front_led<='0';
24                  Back_led<='0';
25              end case;
26
27      END PROCESS;
28  END led_cntrl_b;
```

In the architecture, a signal `STATE` is defined as a two-bit vector. It is evaluated in the process by means of a case-when structure. If `STATE` assumes the value "01", then only the front LED is lit up. If it is "10", the front LED will be on, and the back LED will be turned on only if the car goes backward. If `STATE` is "11", then both the front LED and back LED will be turned on. If `STATE` is "00", then both the front LED and back LED will be turned off.

5 3d model and assembly

The construction of the vehicle is done from salvaged materials: for example, the wheels are salvaged from a skateboard and the iron frame is cut out from the shell of an old stove. These parts are important on Solidworks software [7] and will be the basis for modeling the remaining mechanical parts, eventually 3d printing with fdm or resin printers.

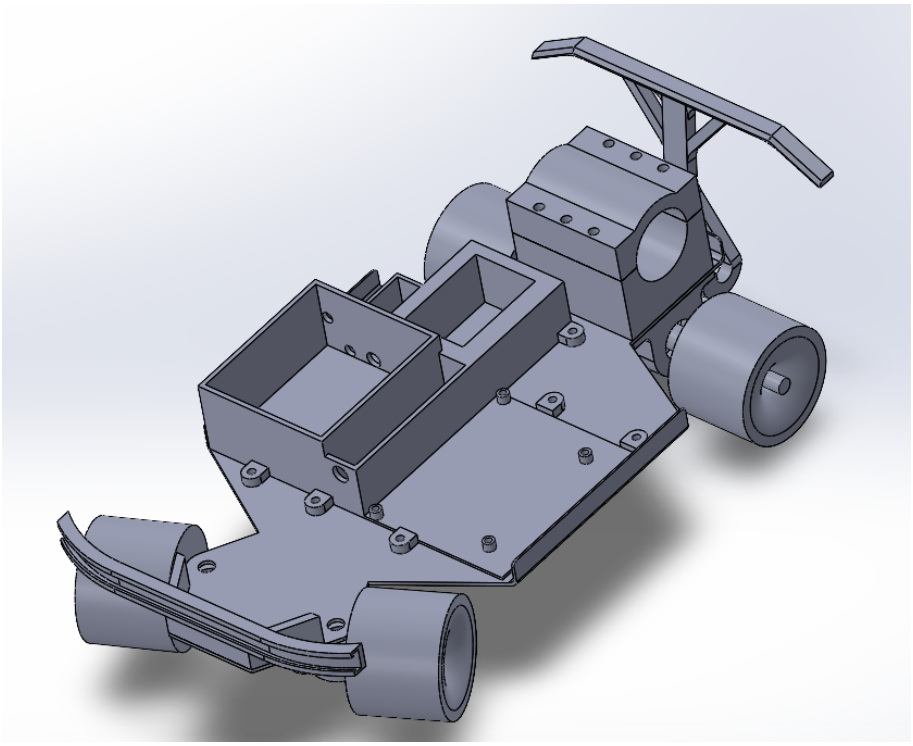

Figure 13: 3d model assembled

A delicate part is the design of the servo-actuated electro steering. For this to work well it must follow specific design laws (an example is the Ackermann steering). The steering also uses recycled parts from a skateboard such as ball bearings. The motor for traction is anchored to the frame with a molded mount; traction is implemented by a strong belt recovered from a CNC that transfers the torque from the motor to the rear axle. The drive axle is also anchored to the chassis with a specially designed part.

Finally, a support is created to house the various electrical devices, ensure the isolation and protection of the shorts (important because the chassis is conductive), and the connection to the common ground via chassis. Each component must be tightly welded to ensure stability of connections during movement. To aid dissipation of the BTS7960 module, a direct connection is made between the cooling fin and the chassis.

The 3d model (fig.13) is concluded with the general assembly and the creation of the front headlight and the rear headlight embedded in the spoiler. For protection of the parts, a body with magnetic attachment is considered for easy access to the electronics (fig.14).
The major difficulties in making the 3D model are due to the readjustment of some components and the study of the usable space. In the practical realization, electronic wiring must be taken into

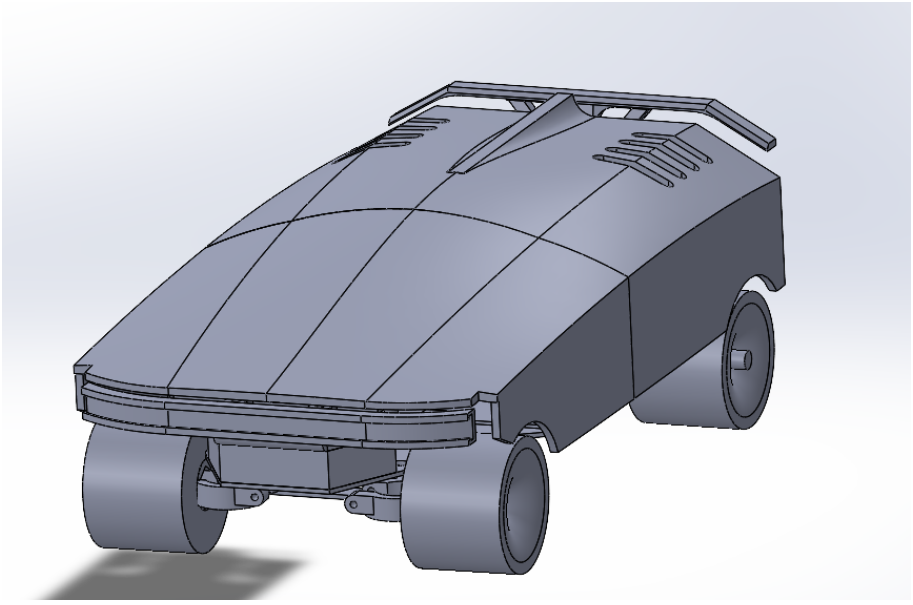

Figure 14: Final result of 3d model

account, which requires a certain harmony in space arrangement. Materials that are not salvaged, then specially 3d printed must be adequate for the purpose. Aesthetic components but requiring more precision are printed in resin, a more delicate and inflexible material. For the anchor parts of electronic devices, and for the magnetic shell, it is decided to use the fdm printer with pla+ material. Finally, parts subject to greater stress, involved in steering and traction, are molded with a special stronger polymer-carbon compound.

The final result of the prototype can be seen in the figures below (from fig.15 to 20).

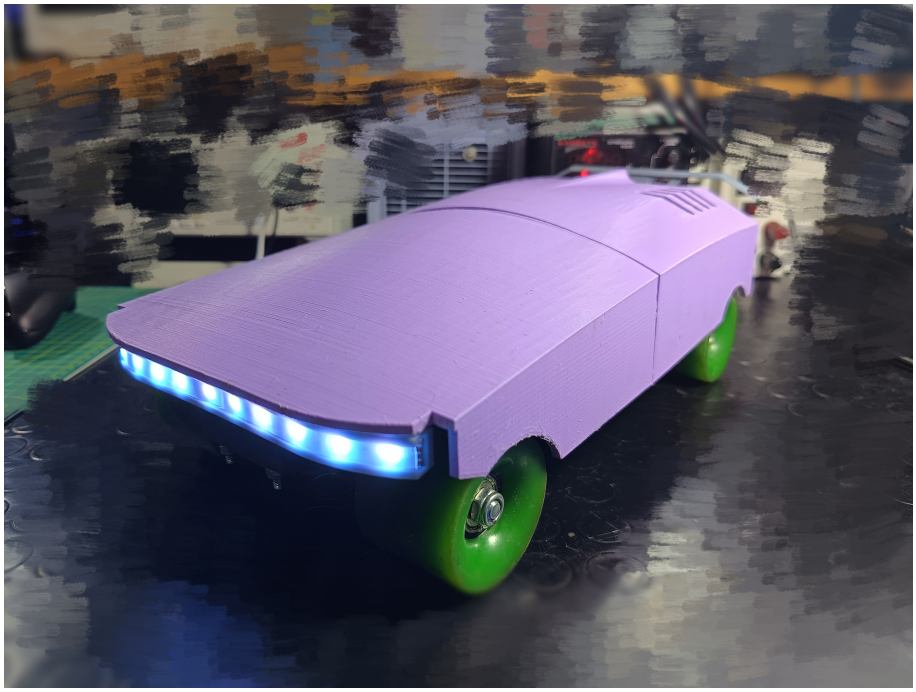

Figure 15: Final result

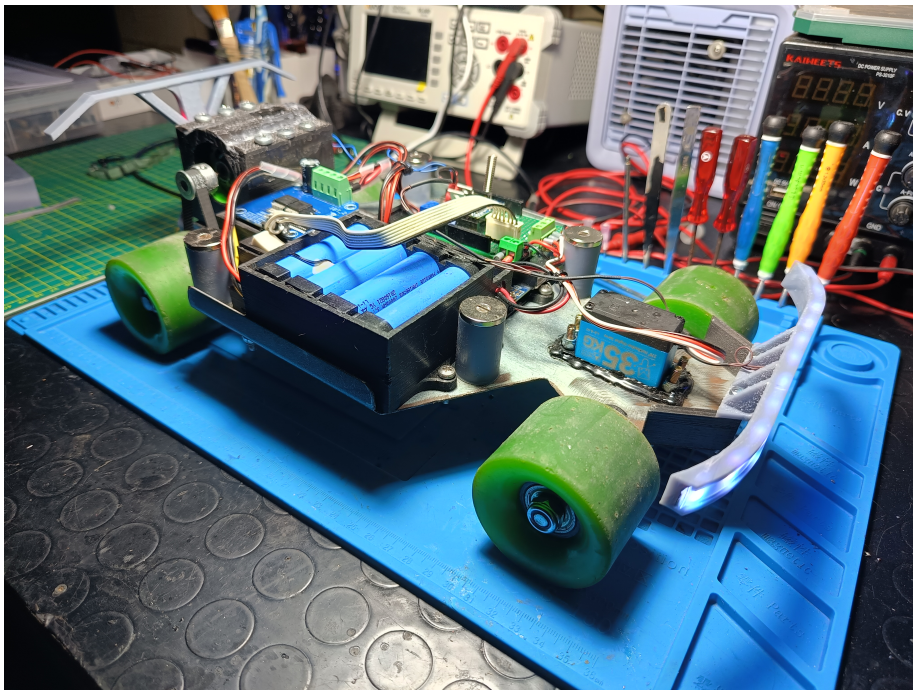

Figure 16: Generic view prototype

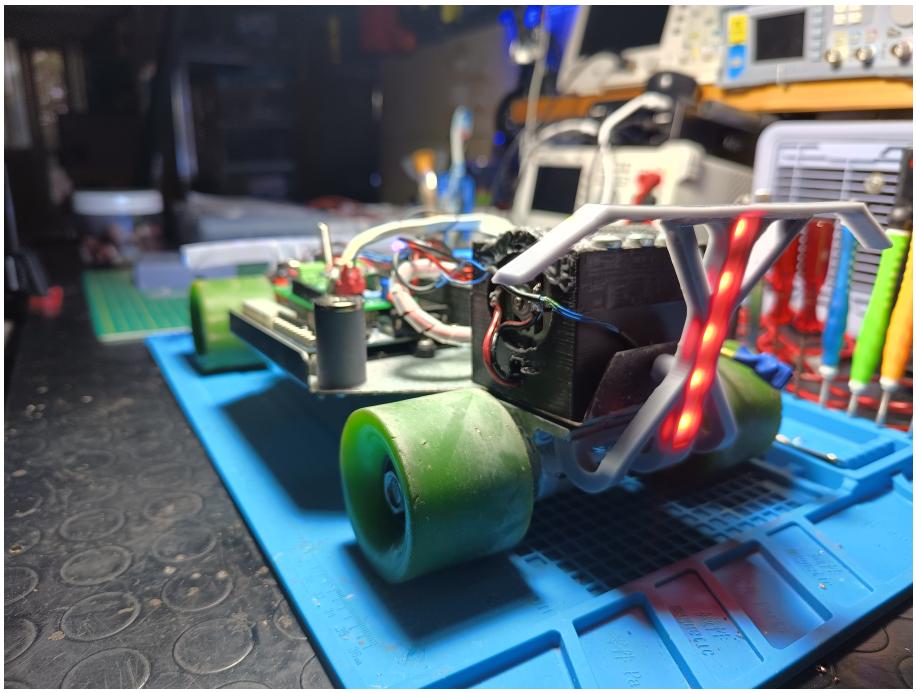

Figure 17: ISO rear prototype

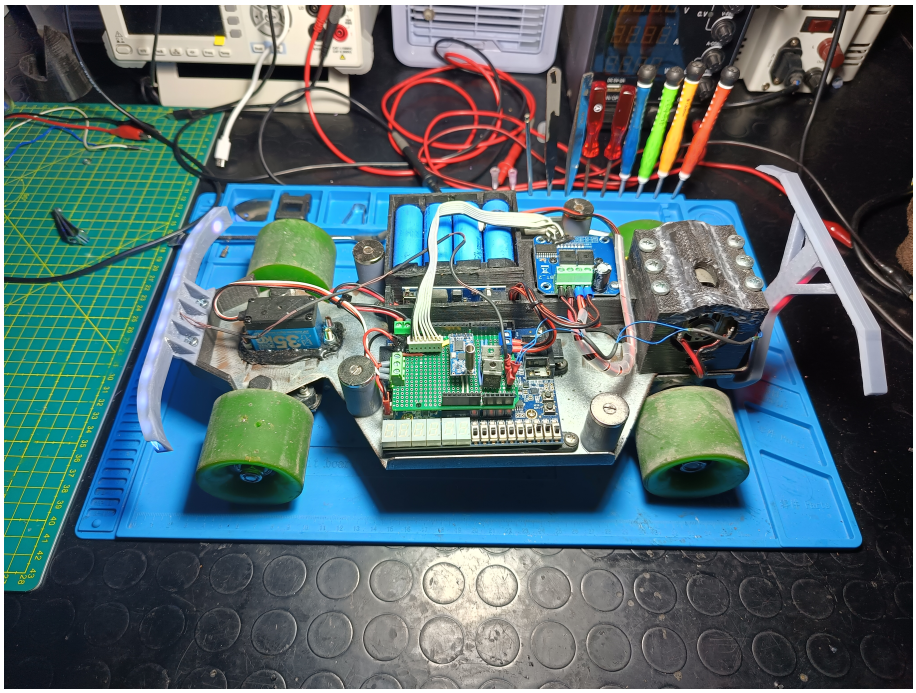

Figure 18: Overview prototype

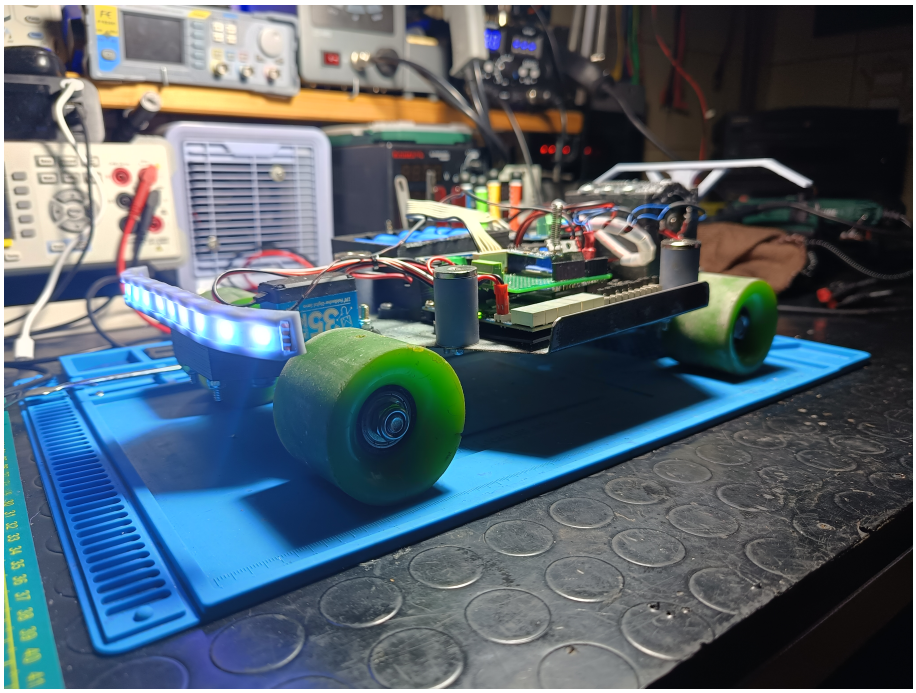

Figure 19: ISO front prototype

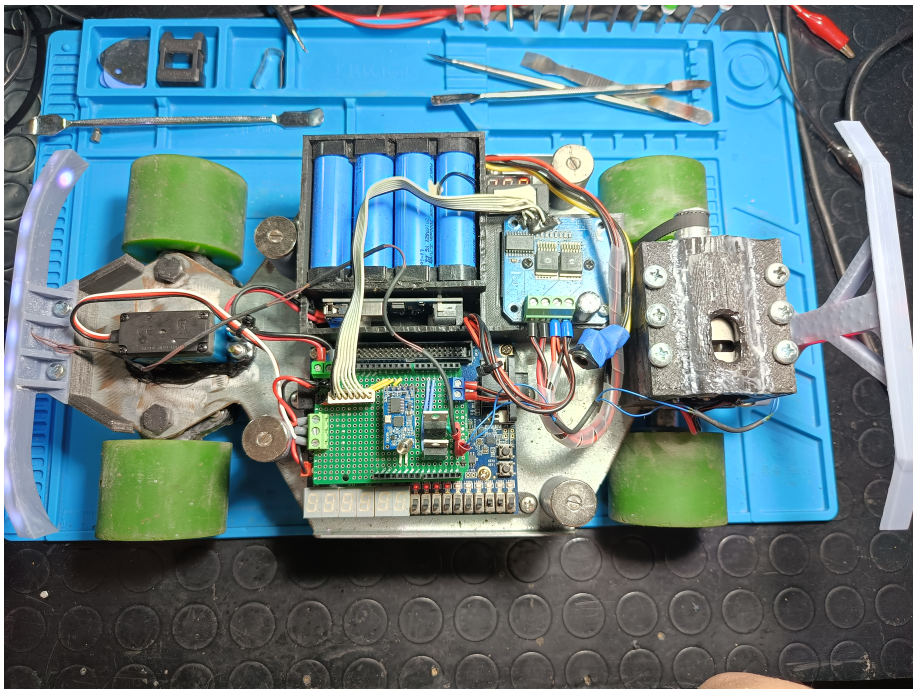

Figure 20: Top view prototype

6 Conclusions

The tests performed point out the responsiveness of the vehicle in responding to commands. The result can be partly attributed to the parallel progress management that the FPGA is able to offer compared to a common MCU. The implementation of the different components, interconnected then only in the main, made the program modular and flexible, making it easy to modify and add new features.

There are several possible improvements to the design, starting with the introduction of a closed loop on board the vehicle. To do this, it would first be necessary to add the library that would allow transmission via UART protocol; this would make two-way communication possible. The advantage of this would be to be able to introduce a set of sensors on board the vehicle to monitor certain parameters of interest (e.g., the speed of the vehicle, rather than the residual charge or the temperature reached by the power components). The transmitter would then act accordingly by closing this loop.

However, this improvement would lead us to face another problem due to the saturation of the best available 8 bits. Unfortunately, the UART allows the transmission of only a few bits at a time. Other radio modules allow more substantial data transmissions with the use of other protocols such as SPI or I2C.

Modifications on the hardware possible, on the other hand, are many and include work on the mechanics. Similar radio-controlled vehicles, for example, use brushless motors rather than DC motors; this would allow greater efficiency but would force us to rewrite the block for PWM control of the DC motor, having to adapt different logic.

References

[1] *Arduino SoftwareSerial library:* URL: https://docs.arduino.cc/learn/built-in-libraries/software-serial/ (visited on 05/22/2024).

[2] *Kikad Software:* URL: https://www.kicad.org/ (visited on 05/22/2024).

[3] *Quartus Software:* URL: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime/resource.html (visited on 05/22/2024).

[4] *Programming the Flash:* URL: https://www.intel.com/content/www/us/en/docs/programmable/683526/current/programming-the-flash-using-quartus.html (visited on 05/22/2024).

[5] *File Downloaded from.* URL: http://www.nandland.com (visited on 05/22/2024).

[6] *More Info About AlteraPll:* URL: https://www.intel.com/content/www/us/en/docs/programmable/683359/17-0/altera-phase-locked-loop-ip-core-user-guide.html (visited on 05/22/2024).

[7] *Solidworks Software:* URL: https://www.solidworks.com/ (visited on 05/22/2024).