Report: Gesture Controlled Robot

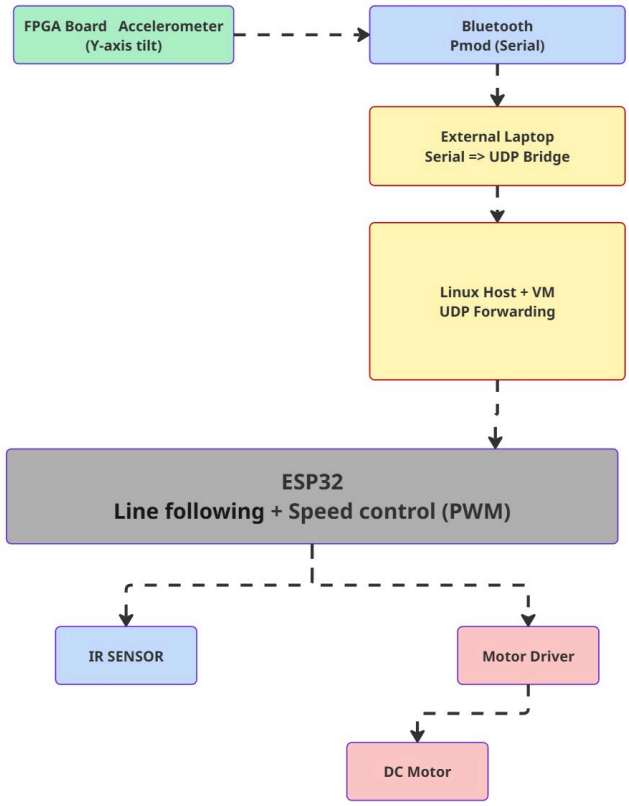
Team B

15.01.2026

1. Team members

Name	Email	Matriculation Number
Yuming Wang	yuming.wang@stud.hshl.de	2219950
Moiz Zaheer Malik	moiz-zaheer.malik@stud.hshl.de	2220074
Rubayet Kamal	rubayet.kamal@stud.hshl.de	2219943

2. Concept description



2.1 Overall system block diagram

This project implements a tilt based speed control system for a small line following vehicle. The main idea is to control the acceleration and deceleration of the vehicle by changing the physical position of an FPGA board.

An accelerometer is connected to the FPGA board. The accelerometer measures the tilt of the board along the Y axis. Based on this tilt value, the FPGA generates a numeric value in the range of 0 to 1000.This value represents the desired speed of the vehicle.

The FPGA sends this speed value through a Bluetooth serial Pmod module. On the vehicle side, an ESP32 microcontroller receives the speed value and uses it to control the motor speed using PWM signals. At the same time, the vehicle follows a black line on the ground using two infrared sensors. The infrared sensors control the steering, while the speed is controlled only by the FPGA tilt.

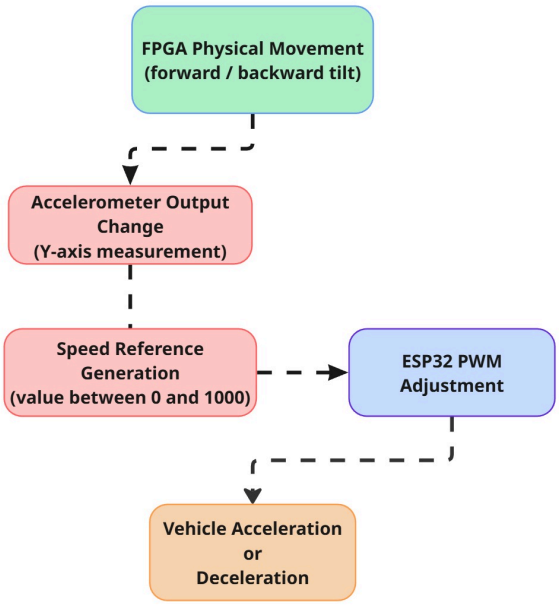
Due to Bluetooth compatibility issues on Linux, a multi stage communication path was used. The Bluetooth module on the FPGA was connected to a friend's laptop. The received data was then forwarded over Wi Fi using UDP packets to a Linux host system. From the host system, the data was forwarded into a QEMU virtual machine. Finally, the virtual machine sent the speed value to the ESP32 over UDP through Wi Fi.

This approach allowed the project to continue even though direct Bluetooth communication with Linux was not reliable.

2.1 Controlling acceleration and deceleration using FPGA tilt

When the FPGA board is tilted forward or backward, the accelerometer output changes. This change is mapped to a speed value between 0 and 1000 inside the FPGA logic. A higher value increases the motor speed, while a lower value reduces it.

The ESP32 receives this value and converts it into a PWM duty cycle for the motor driver. To avoid sudden speed jumps, the ESP32 gradually ramps the PWM value to the new target speed. This results in smooth acceleration and deceleration of the vehicle.



2.2 Acceleration / Deceleration Concept Project/Team management

3. Project method

An incremental development approach was used for this project. Each subsystem was implemented and tested independently before integrating it into the full system.

The work started with the vehicle platform. First, the ESP32 was programmed to control the motors and follow a line using infrared sensors. After this was stable, PWM based speed control was added.

In parallel, the FPGA part was developed to read accelerometer data and generate a speed value. Once both sides worked independently, communication between the FPGA and ESP32 was addressed.

When Bluetooth communication on Linux failed, the team adapted the design and introduced a UDP based forwarding solution using an external laptop and a virtual machine. This decision helped avoid blocking the project and allowed testing of the complete control chain.

3.1 Task breakdown and management

The project tasks were divided into clear blocks:

1. Vehicle hardware setup and motor control
2. Line following using infrared sensors
3. FPGA accelerometer data processing
4. Bluetooth serial data transmission
5. UDP forwarding and communication bridging
6. System integration and tuning

Each task was tested individually before moving to the next one. Problems were documented and solved step by step instead of changing multiple parts at once.

3.2 Team roles

Moiz Zaeer Malik focused on the complete vehicle side of the project. This included the ESP32 based controller, motor driver and DC motors, as well as the infrared sensors used for line following. He also implemented the motor speed control using PWM, integrated the line following logic, and handled software development and testing for the ESP32 using Thonny. This role also included setting up the Linux host environment and virtual machine used for wireless communication and data forwarding between systems.

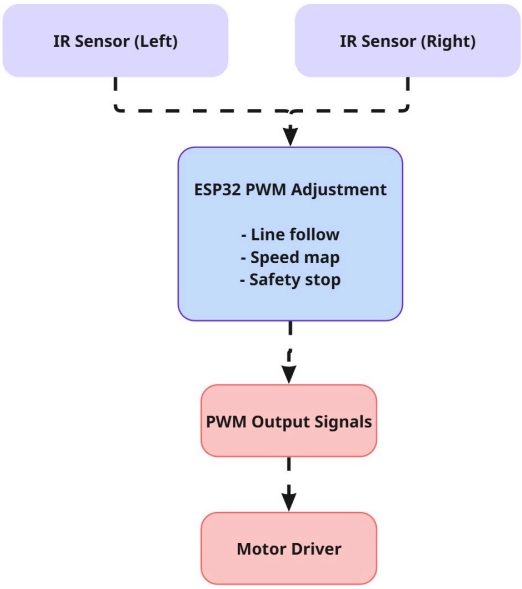
4 Technologies

The project combines embedded hardware, wireless communication, and real time motor control. The main technologies used are an FPGA board with an accelerometer, an ESP32 microcontroller, DC motors with a motor driver, infrared sensors, Bluetooth serial communication, and UDP over Wi Fi.

4.1 VHDL

4.2 FPGA

4.3 ESP32



4.1 ESP32 control structure

The ESP32 is used as the main controller on the vehicle. It is responsible for line following, motor speed control, and wireless communication.

The ESP32 continuously reads two infrared sensors to detect the line position. Based on the sensor values, it adjusts the left and right motor speeds to keep the vehicle on the line.

At the same time, the ESP32 listens for incoming UDP packets that contain the speed value sent from the FPGA. The received value is limited to a valid range and mapped to a PWM duty cycle.

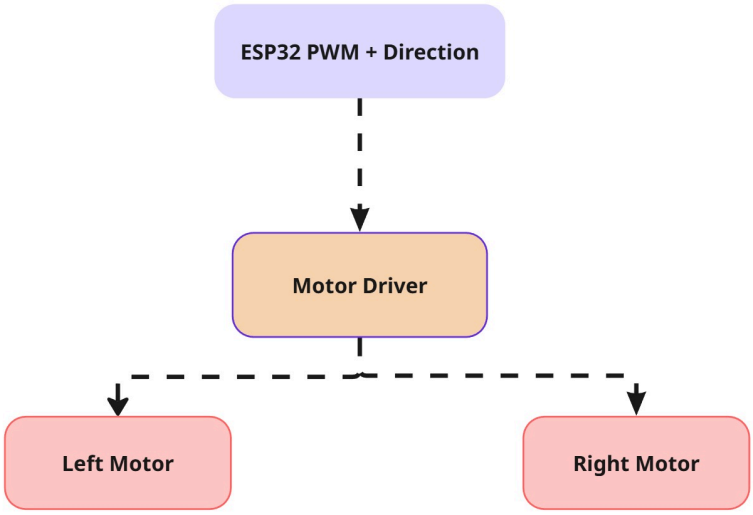
To ensure safe operation, the ESP32 stops the motors if no new speed value is received within a defined timeout period. This prevents uncontrolled movement in case of communication failure.

1. Motors and Motor Driver

The vehicle uses two DC motors driven by a motor driver module. Each motor can be controlled independently using direction pins and a PWM enable signal. The ESP32 sets the motor direction using GPIO pins. The speed is controlled by adjusting the PWM duty cycle applied to the motor driver.

The speed value received from the FPGA defines the base speed for both motors. During line

following, one motor is slowed down relative to the other to allow turning. To avoid mechanical stress and unstable motion, the PWM value is increased or decreased gradually using a ramp function.

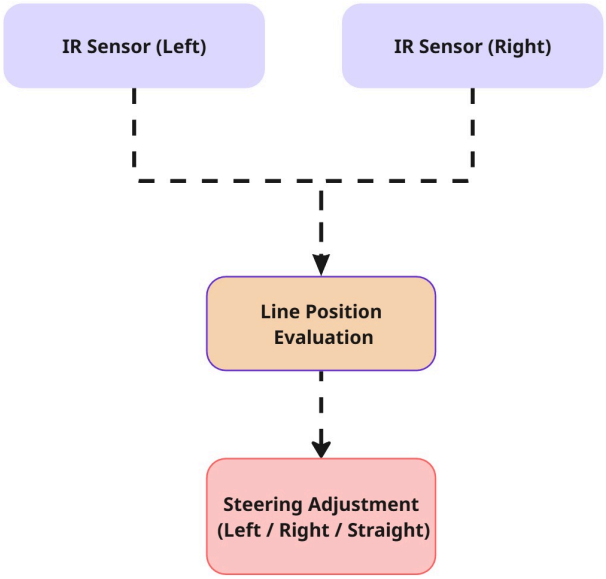


4.4 Motor control concept

2. Infrared Sensors

Two infrared sensors are mounted at the front of the vehicle. One sensor is placed on the left side and the other on the right side. The sensors provide digital output signals that indicate whether the sensor is detecting the line or the background surface. The ESP32 reads these signals through GPIO pins.

If both sensors detect the same surface, the vehicle moves straight. If only one sensor detects the line, the vehicle turns toward that side. If the line is lost, the vehicle slows down or stops depending on the implemented logic. This simple sensor setup provides reliable line following behavior and works independently of the speed control system.

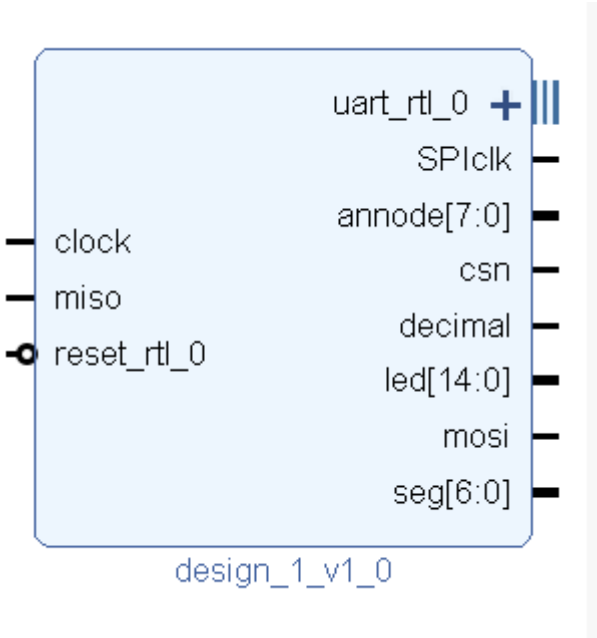


4.5 Line following logic

Implementation and Intergrated Technologies

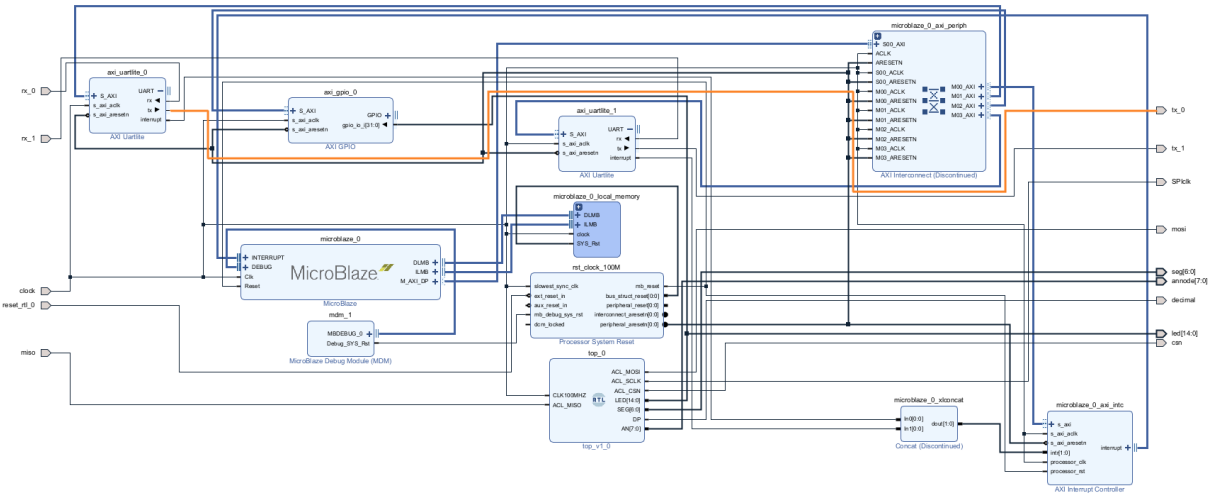
2. *IP integration in VHDL*

Vivado IP refers to the library of reusable, configurable Intellectual Property (IP) cores provided within the Xilinx Vivado Design Suite. These IP cores implement commonly used functions such as communication interfaces, memory controllers, processing elements, and DSP modules. By integrating Vivado IP into a design, developers can significantly reduce development time, improve design reliability, and focus on system-level innovation rather than low-level implementation details. The IP cores are highly parameterizable and seamlessly integrated with Vivado's synthesis, implementation, and verification flows.
In our project, we first create one ip for acceleration meter internal of Nexays A7, The process starts with VHDL code design, then we package it as one ip called ADXL362_Accelerometer . This ip is used furtherly in Microblaze block diagram.



3. *Microblaze configuration*

The MicroBlaze processor is configured as the central control unit of the system and is integrated using an AXI-based architecture. It is connected to local instruction and data memory through the Local Memory Bus (LMB) to provide low-latency access for program execution. An AXI Interconnect is used to interface the MicroBlaze with multiple AXI peripherals, including AXI UARTLite modules for serial communication, an AXI GPIO module for general-purpose I/O control, and an AXI Interrupt Controller for centralized interrupt management. System-wide clocking and reset signals are generated and distributed by the Processor System Reset module to ensure proper synchronization and reliable startup behavior. This configuration enables efficient communication between the processor and peripherals while maintaining a modular and scalable design. The AXI-Uartlite_0 is the one used for serial monitor in PC connected to FPGA, while the AXI-Uartlite_1 is used for Bluetooth Pmod BT2 connected to FPGA. For both of them, the rx and tx are set external.



After setting up Microblaze in Vivado, next step is export the hardware with bitstream into **Vitis**. Then one embedded used case is set up for using the microcontroller we designed. This application is developed in **Vitis** and runs on the MicroBlaze processor to handle GPIO data acquisition and Bluetooth communication via AXI UARTLite. The program continuously reads a 32-bit GPIO input, extracts and processes the lower 16 bits as signed motion-related values, and transmits the computed speed over a Bluetooth UART interface.
Several key functions are used in this implementation.
The **BtUart_Init_ByBaseAddr()** function initializes the AXI UARTLite peripheral using its base address and resets the FIFO to ensure reliable communication.
The **BtUart_DumpRx()** function performs a non-blocking read of the UART receive FIFO and prints any incoming responses from the Bluetooth module, which is mainly used during AT-command configuration.
The **to_binary_16_lower()** function converts the lower 15 bits of a GPIO register value into a binary string representation, simplifying bit-level parsing and debugging.
In the main loop, **Xil_In32()** is used to read raw GPIO data directly from the AXI GPIO register. The extracted signed values are then scaled and accumulated to update a speed variable, which is periodically sent to the Bluetooth module using **XUartLite_Send()**. This design demonstrates a lightweight polling-based approach for peripheral communication and real-time data processing on a MicroBlaze system.

4. Powershell commands

After setting up **BT2** Pmod to be in the transmission mode, we use windows pc to connect to this module. Then powershell commands are used to read the data, the code can be seen below.

```
PS C:\Users\30887> # 电脑A: COM5 -> COM6 (传出到对方COM4) COM5=bluetooth(input), COM6=bluetooth(output)
>> $srcCom  = "COM5"
>> $dstCom  = "COM6"
>> $baudSrc = 9600
>> $baudDst = 9600
>>
>> # 源端口（已有蓝牙数据流）
>> $src = [System.IO.Ports.SerialPort]::new($srcCom, $baudSrc, "None", 8, "One")
>> $src.ReadTimeout = 200
>>
>> # 目标端口（要写过去）
>> $dst = [System.IO.Ports.SerialPort]::new($dstCom, $baudDst, "None", 8, "One")
>> $dst.WriteTimeout = 200
>> $dst.Handshake = [System.IO.Ports.Handshake]::None
>> $dst.DtrEnable = $true
>> $dst.RtsEnable = $true
>>
>> $src.Open()
>> $dst.Open()
>>
>> Write-Host "Forwarding $srcCom -> $dstCom  (Ctrl+C to stop)"
>>
>> try {
>>   while ($true) {
>>     $n = $src.BytesToRead
>>     if ($n -gt 0) {
>>       $data = $src.ReadExisting()
>>
>>       # 建议：去掉 \r，防止终端/下游设备异常
>>       $data = $data -replace "`r",
>>
>>       # 1)本地显示（你能看到实时数据）
>>       Write-Host -NoNewline $data
>>
>>       # 2)转发到 COM6(ESP)
>>       try {
>>         $dst.Write($data)
>>       } catch {
>>         Write-Warning "Write to $dstCom timeout, dropping data"
>>       }
>>     }
>>     Start-Sleep -Milliseconds 20
>>   }
>> }
>> finally {
>>   $src.Close()
>>   $dst.Close()
>> }
```

The **COM5** port is configured as the input interface for the **BT2 Bluetooth module**, receiving wireless data transmitted from the MicroBlaze system. The **COM6** port serves as the output interface connected to the **ESP32**, forwarding the received data in real time. Local data display is implemented using the **Write-Host** command in the PowerShell script. After serial data is read from the source **COM** port via the **ReadExisting**() method, the received content is immediately printed to the PowerShell console using **Write-Host -NoNewline $data**. This allows real-time monitoring of incoming Bluetooth data on the local host without affecting the forwarding process, providing a convenient debugging and verification mechanism during system testing.

2. Results

Describe the implementation of your schematic and PCB design.
Give a summary about your PCB design results (Layout, BOM, Size, Costs? etc.)

3. Conclusion

4.

5. Sources/References

Provide the sources on the technologies and algorithms you used in your project (Github).