**Performance Optimization of EE Architecture for Software Defined Vehicles**

**Submitted**

**By**

**Sunkara Venkatesh**

**Poluru Sai Chaithanya**

**Gurram Chethan Sree Royal**

**Regd. No:**

**BU21EECE0100308**

**BU21EECE0100208**

**BU21EECE0100529**

**Under the Guidance of**

**Dr.S. Karthick, Associate Professor,)**

**(Duration:08 /07/2024 to 11/10/2024)**



**Department of Electrical, Electronics and Communication Engineering**

**GITAM School of Technology**

**GITAM**

**(DEEMED TO BE UNIVERSITY)**

**(Estd. u/s 3 of the UGC act 1956)**

**NH 207, Nagadevanehalli, Doddaballapur taluk, Bengaluru-561203 Karnataka, INDIA.**

**DECLARATION**

**I/We declare that the project work contained in this report is original and it has been done by me under the guidance of my project guide.**

**Name:**

**Date: Signature of the Student**

**Department of Electrical,Electronics and Communication Engineering**

**GITAM School of Technology, Bengaluru-561203**

****

**CERTIFICATE**

**This is to certify that (S Venkatesh & P Sai Chaithanya & G Chethan sree Royal) bearing (BU21EECE0100308 & BU21EECE0100208 & BU21EECE0100529) has satisfactorily completed Mini Project Entitled in partial fulfillment of the requirements as prescribed by University for VIIth semester, Bachelor of Technology in “Electrical, Electronics and Communication Engineering” and submitted this report during the academic year 2024-2025.**

**[Signature of the Guide] [Signature of HOD]**

**Table of contents**

[**Chapter 1: Introduction** 4](#_Toc179551667)

[**1.1 Overview of the problem statement** 4](#_Toc179551668)

[**1.2 Objectives and goals** 4](#_Toc179551669)

[**Chapter 2: Literature Review** 5](#_Toc179551670)

[**Chapter 3: Strategic Analysis and Problem Definition** 6](#_Toc179551671)

[**3.1 SWOT Analysis** 6](#_Toc179551672)

[**3.2 Project Plan - GANTT Chart** 6](#_Toc179551673)

[**3.3 Refinement of problem statement** 6](#_Toc179551674)

[**Chapter 4: Methodology** 7](#_Toc179551675)

[**4.1 Description of the approach** 7](#_Toc179551676)

[**4.2 Tools and techniques utilized** 7](#_Toc179551677)

[**4.3 Design considerations** 7](#_Toc179551678)

[**Chapter 5: Implementation** 12](#_Toc179551679)

[**Chapter 6: Results** 22](#_Toc179551680)

[**6.1 outcomes** 22](#_Toc179551681)

[**6.2 Interpretation of results** 23](#_Toc179551682)

[**Chapter 7: Conclusion** 25](#_Toc179551683)

[ **References** 26](#_Toc179551686)

# **Chapter 1: Introduction**

In embedded systems, communication between devices is crucial for data transfer, synchronization, and control. This project focuses on three widely-used communication protocols: CAN (Controller Area Network), I2C (Inter-Integrated Circuit), and SPI (Serial Peripheral Interface).

CAN is designed for real-time data transfer in noisy environments, widely employed in automotive and industrial systems. I2C, a simple two-wire protocol, facilitates low-speed communication between a master and multiple slave devices, making it ideal for embedded systems like sensors and displays. SPI, a high-speed full-duplex protocol, is used for efficient, low-latency communication between a master and peripherals like memory devices and sensors.

Together, these protocols form the backbone of communication in modern embedded systems, enabling seamless data exchange across various applications.

## **1.1 Overview of the problem statement**

The increasing complexity of embedded systems in fields like automotive, industrial automation, and consumer electronics demands reliable, efficient, and real-time communication between multiple devices. Systems require communication protocols that can handle various data transmission needs, from high-speed, low-latency operations to robust, fault-tolerant interactions in noisy environments.

This project addresses the challenge of designing, simulating, and validating three key communication protocols—CAN, I2C, and SPI—using Verilog. Each protocol has its own strengths and specific use cases, but integrating and understanding their functionality through simulations provides crucial insights into their behavior and performance in real world applications.

## **1.2 Objectives and goals**

* To design and implementation of algorithm for a Software Defined Vehicles using FPGA.
* Performance measurement of FIELD PROGRAMMABLE GATE ARRAY.
* To implement communication protocols I2C, SPI and controller area network (CAN) protocols using FPGA design.
* This project aims to demonstrate the proper functioning of these protocols through waveforms analysis and simulation and ensuring compliance with standard communication specifications.

**Design and Implementation**: To design and implement the CAN, I2C, and SPI communication protocols in Verilog, ensuring each protocol meets industry standards for data transfer, signal integrity, and timing.

**Simulation and Validation**: To simulate the functionality of each protocol and validate their performance through waveform analysis, ensuring reliable data transmission, real-time communication, and error detection.

# **Chapter 2: Literature Review**

* “Guilherme Marcon Zago and Edison Pignaton de Freitas” The number of electronic systems in agriculture machinery grows, becoming more complex and multifunctional. Due to the increasing number of devices connected to the network and the complexity of their functions, the current controller area network (CAN) becomes overloaded and CAN with flexible data rate (CAN FD) emerges as an alternative.
* “Changmin Shin” The purpose of this paper is to provide a framework for the transmission of long data beyond the size of 8 bytes in CAN network. Controller Area Network (CAN) is a multi-master, message-based serial network communication protocol. Since the "Data field" of the CAN Frame supports the size of 8 bytes, the CAN transmitting node cannot transfer data beyond the size of 8 bytes.
* “Peter Hobden and, Saket Srivastav” Serial Peripheral Interface (SPI) is a commonly used communication protocol that allows serial data transfer between a master and a slave device over a short distance.
* “Jun-Cheol Lee, Tae-Oh Kim, Joo-Hyung Chae” Inter-integrated circuit (I2C) is one of the integrated circuit communication protocols, and since its transmission speed is relatively slow, it is mainly used to connect devices operating at low speeds. The fact that there are only two pins, a serial data line (SDL) and a serial clock line (SCL), simplifies the interface, but there is a speed limit because both transmission and reception are done through one data line. If an error occurs during I2C operation, it is important to determine the cause. In this case, various timing parameters on the specification have to be identified.
* “Naveen RS, Sanjay S, Senthil Murugan A , Surya Prakash M , Mukuntharaj C” The I2C involves designing a digital circuit that enables communication between different integrated circuits or peripherals in a synchronous serial manner. I2C is widely used in embedded systems due to its simplicity and versatility, allowing multiple devices to be connected on the same bus.

# **Chapter 3: Strategic Analysis and Problem Definition**

## **3.1 SWOT Analysis**

* **Strengths**

Innovative Problem Solving

Real-world application

Data driven and research based

* **Weaknesses**

Complexity of integration

Security vulnerabilities

Limited flexibility

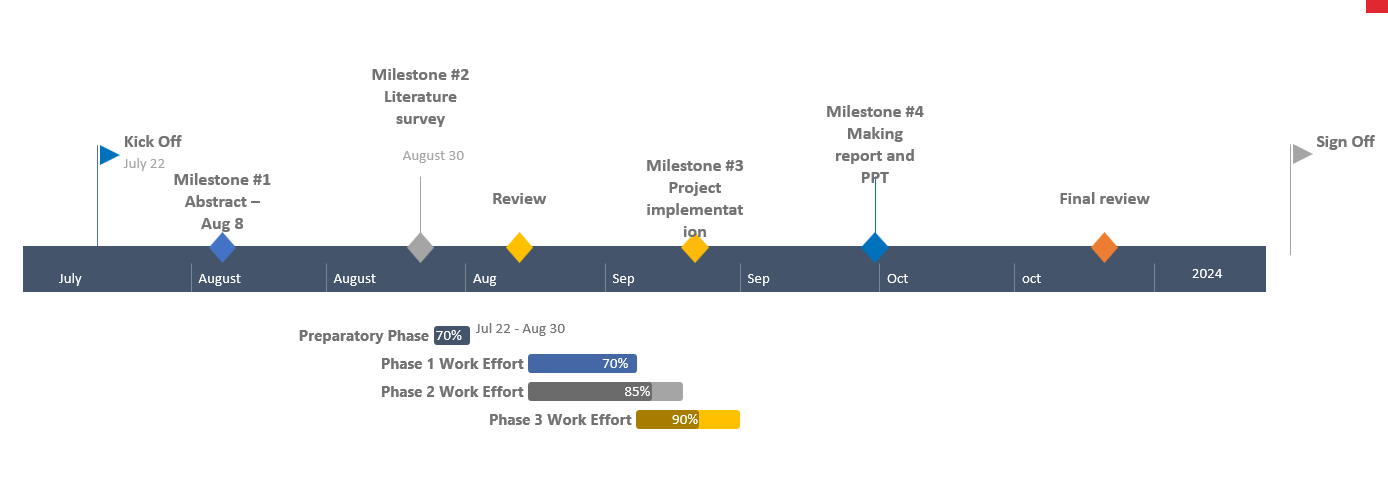
* **Opportunities**

Industry Relevance

Research potential

Collaboration

### **3.2 Project Plan - GANTT Chart**



##### **3.3 Refinement of problem statement**

The project aims to accurately model of the protocols, ensuring they meet industry standards for performance and functionality. Through simulation and waveform analysis, the protocols will be tested for their real-time capabilities and error-handling mechanisms. The ultimate goal is to create a foundation for future hardware implementation on FPGA platforms, where these protocols can be tested and optimized in real-world scenarios.

# **Chapter 4: Methodology**

The methodology involves designing the CAN, I2C, and SPI protocols in Verilog, followed by extensive simulations to validate their functionality and performance through waveform analysis.

## **4.1 Description of the approach**

Initially, we conduct in-depth research on each protocol to understand their specifications, use cases, and operational characteristics. This foundational knowledge informs the design decisions for each protocol.

### **4.2 Tools and techniques utilized**

* Used a Xilinx Vivado 2016.2 version

#### **4.3 Design considerations**

Using Verilog, we develop the architecture for each protocol, focusing on essential components such as message framing for CAN, master-slave communication for I2C, and full-duplex data transfer for SPI. This design phase incorporates best practices to ensure adherence to industry standards.

**PROTOCOL 1: I2C** (Inter Integrated Circuit)

I2C stands for Inter-Integrated Circuit. It is a bus interface connection protocol incorporated into devices for serial communication. It was originally designed by Philips Semiconductor in 1982. Recently, it is a widely used protocol for short-distance communication. It is also known as Two Wired Interface (TWI).

**Working of I2C Communication Protocol**:

* It uses only 2 bi-directional open-drain lines for data communication called SDA and SCL. Both these lines are pulled high.
* Serial Data (SDA) – Transfer of data takes place through this pin.
* Serial Clock (SCL) – It carries the clock signal.
* I2C operates in 2 modes –

Master mode

Slave mode

* Each data bit transferred on SDA line is synchronized by a high to the low pulse of each clock on the SCL line.

A diagram of a computer chip

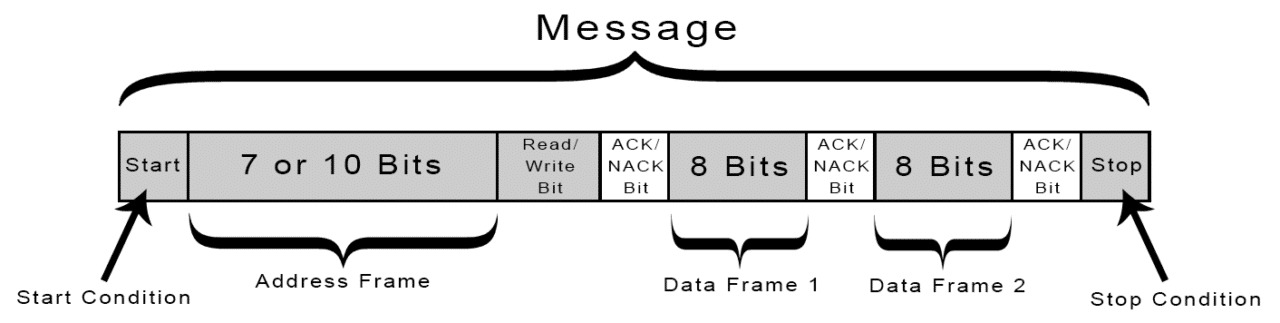
Description automatically generated

According to I2C protocols, the data line cannot change when the clock line is high, it can change only when the clock line is low. The 2 lines are open drain, hence a pull-up resistor is required so that the lines are high since the devices on the I2C bus are active low. The data is transmitted in the form of packets which comprises 9 bits. The sequence of these bits are –

Start Condition – 1 bit

Slave Address – 8 bit

Acknowledge – 1 bit



With I2C, data is transferred in messages. Messages are broken up into frames of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

**Start Condition:** The SDA line switches from a high voltage level to a low voltage level before the SCL line switches from high to low.

**Stop Condition:** The SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high.

**Address Frame:** A 7or10 bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

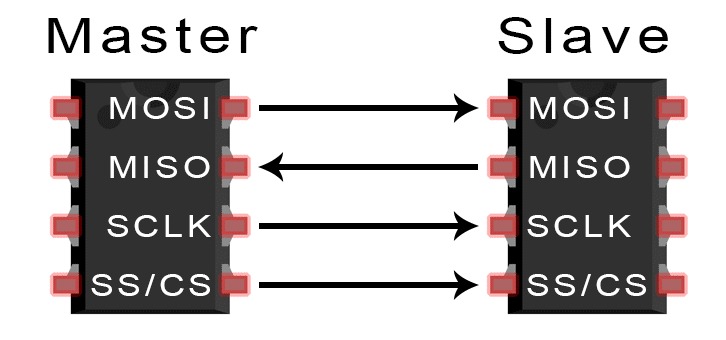
**Read/Write Bit:** A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

**ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

**PROTOCOL 2: SPI (serial peripheral interface)**

SPI is a common communication protocol used by many different devices. For example, SD card reader modules, RFID card reader modules, and 2.4 GHz wireless transmitter/receivers all use SPI to communicate with microcontrollers.

One unique benefit of SPI is the fact that data can be transferred without interruption. Any number of bits can be sent or received in a continuous stream.

Devices communicating via SPI are in a master-slave relationship. The master is the controlling device (usually a microcontroller), while the slave (usually a sensor, display, or memory chip) takes instruction from the master. The simplest configuration of SPI is a single master, single slave system, but one master can control more than one slave (more on this below). 

* MOSI (Master Output/Slave Input) – Line for the master to send data to the slave.
* MISO (Master Input/Slave Output) – Line for the slave to send data to the master.
* SCLK (Clock) – Line for the clock signal.
* SS/CS (Slave Select/Chip Select) – Line for the master to select which slave to send data to.

**Slave Select**

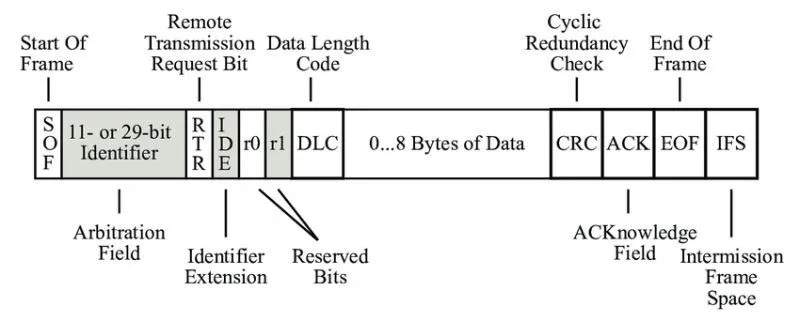
The master can choose which slave it wants to talk to by setting the slave’s CS/SS line to a low voltage level. In the idle, non-transmitting state, the slave select line is kept at a high voltage level. Multiple CS/SS pins may be available on the master, which allows for multiple slaves to be wired in parallel. If only one CS/SS pin is present, multiple slaves can be wired to the master by daisy-chaining.

**MOSI and MISO**

* The master sends data to the slave bit by bit, in serial through the MOSI line. The slave receives the data sent from the master at the MOSI pin. Data sent from the master to the slave is usually sent with the most significant bit first.
* The slave can also send data back to the master through the MISO line in serial. The data sent from the slave back to the master is usually sent with the least significant bit first.

**PROTOCOL 3: CAN** (controller area network)

Controller Area Network, in short CAN protocol is a serial communication protocol which provides efficient support to mainly automotive real time control systems with a very high level of security, error detection and correction. CAN data frame can be defined as in the fig below. It consists of start of frame (SOF), arbitration field, control field, data field, CRC field, ACK field and end of frame (EOF).



* Transmission of a data frame begins with the start bit (Start of Frame — SOF). It is transmitted by the sender as a dominant level which produces a signal edge from the previous recessive (bus idle) level which is used to synchronize the entire network.
* In order for the receivers not to lose synchronism to the sender during transmission of the frame, they compare all recessive-to-dominant signal edges with their preset bit timing. In case of deviation, receivers re-synchronize by the amount of the relevant phase error (re-synchronization).
* Following the SOF is the identifier (ID). This sets the priority of the data frame, and together with the acceptance filtering it provides for sender-receiver relations in the CAN network that are defined in the communication matrix. Next comes the RTR bit (Remote Transmission Request). It is used by the sender to inform receivers of the frame type (data frame). A dominant RTR bit indicates a data frame.
* The IDE bit (Identifier Extension bit) which follows serves to distinguish between standard format and extended format. In standard format the identifier has 11 bits, and in extended format 29 bits. The figure “Data Frame in Standard and Extended Format” is available to study the two formats.
* The DLC (Data Length Code) communicates the number of payload bytes to the receivers. The payload bytes are transported in the data field. A maximum of eight bytes can be transported in one data frame.
* The payload is protected by a checksum using a cyclic redundancy check (CRC) which is ended by a delimiter bit. Based on the results of the CRC, the receivers acknowledge positively or negatively in the ACK slot (acknowledgement) which also is followed by a delimiter bit.
* After this the transmission of a data frame is terminated by seven recessive bits (End of Frame — EOF).

# 

# **Chapter 5: Implementation**

* Implemented all these protocols in verilog codes and testbenches.

**5.1 Verilog Code**

* **Protocol 1(I2C):**

**Master Code:**

module master(

output reg sda,

input [7:0] data,

input [7:0] data\_wr,

input clk,

input rw,

output reg scl,

input [6:0] address,

input [7:0] register

);

reg [8:0] temp;

reg [7:0] register2;

reg [7:0] data\_wr\_dup;

reg pstate;

reg scl2x;

reg ack;

integer i;

integer n;

// Initial block to set initial values

initial begin

i = 0;

n = 0;

scl2x = 0;

ack = 1'b1;

sda = 1;

scl = 1;

#5 sda = 0; // START BIT condition starts here

end

// Detect start condition

always @(negedge sda) begin

if (scl == 1)

n = 1;

end

// Main clock control block

always @(posedge clk) begin

ack = 0;

temp = {address, rw, ack};

register2 = register;

data\_wr\_dup = data\_wr;

if (n == 1 && rw == 1) begin

repeat(50) begin

#2 scl <= !scl;

n = 0;

#1 scl2x <= !scl2x;

end

end else if (n == 1 && rw == 0) begin

repeat(64) begin

#2 scl <= !scl;

#1 scl2x <= !scl2x;

n = 0;

end

end

end

// Control signal for read/write operations

always @(posedge clk) begin

if (i == 25 && rw == 1)

repeat(2) #1 scl2x = !scl2x;

else if (i == 32 && rw == 0)

repeat(2) #1 scl2x = !scl2x;

end

// I2C bit transmission control

always @(posedge scl2x) begin

if (i <= 9) begin

sda = temp[8];

temp = temp << 1;

end else if (i == 12 || i == 13) begin

sda = 1'b0; // Acknowledge bit

end else if (i >= 14) begin

sda = register2[7];

register2 = register2 << 1;

end

if (rw == 0 && i >= 23) begin

sda = data\_wr\_dup[7];

data\_wr\_dup = data\_wr\_dup << 1;

end

i = i + 1;

// Stop condition

if (i > 32 && rw == 0)

sda = 1;

else if (i > 25 && rw == 1)

sda = 1;

end

// Instantiate the slave module

slave slv(data, sda, scl);

endmodule

**Slave Code:**

module slave(

output reg [7:0] out,

input sda,

input scl

);

integer j = 0;

reg [6:0] temp;

reg [7:0] add;

reg rw;

reg [7:0] register\_address;

reg bitin;

reg [7:0] storage[0:38];

initial begin

// Initialize storage with a value at index 37

storage[37] = 16;

end

parameter address = 7'b1101001; // Corrected parameter declaration

always @(posedge scl) begin

bitin = sda;

if (j < 8)

temp = {temp, bitin}; // Shift in the bit

if (j == 8) begin

// Determine read/write based on the ACK bit

if (bitin == 0)

rw = 0; // Write operation

else

rw = 1; // Read operation

end

j = j + 1;

// Check if the address matches for read operations

if (temp == address && (j > 15 && j < 24) && rw == 1) begin

add = {add, bitin};

end

// Check if the address matches for write operations

if (temp == address && rw == 0 && j > 15 && j != 24 && j < 33) begin

add = {add, bitin};

end

// Store the register address at the appropriate time

if (j == 24)

register\_address = add;

// Store the data in storage if it's a write operation

if (j == 33 && rw == 0)

storage[register\_address] = add;

// Output the value from the storage

out = storage[add];

end

endmodule

**Testbench Code:**

module tbmast;

// Inputs

reg [6:0] address;

reg [7:0] register;

reg [7:0] data;

reg [7:0] data\_wr;

reg clk;

reg rw;

// Outputs

wire sda;

wire scl;

// Instantiate the Unit Under Test (UUT)

master uut (

.address(address),

.register(register),

.clk(clk),

.rw(rw),

.sda(sda),

.scl(scl),

.data(data),

.data\_wr(data\_wr)

);

initial begin

// Initialize Inputs

address = 7'b1101001; // Corrected binary declaration

register = 7'b0100101;

data = 7'b1011010; // Corrected binary declaration

clk = 0;

rw = 0;

data\_wr = 20;

// Wait 100 ns for global reset to finish

#100;

// Add stimulus here

end

// Generate clock signal

always

#1 clk = ~clk; // Toggle clock every 1 time unit

Endmodule

* **Protocol 2(SPI):**

**Master Code:**

module spi\_master (

input wire clk, // System clock

input wire rst, // Reset

input wire [7:0] data\_in, // Data to be sent to the slave

input wire miso, // Master In Slave Out

output reg mosi, // Master Out Slave In

output reg sclk, // Serial Clock

output reg ss, // Slave Select (Active Low)

output reg [7:0] data\_out // Data received from the slave

);

reg [2:0] count; // 3-bit counter to count 8 bits (0-7)

reg [7:0] mem; // Register to store received data

initial begin

sclk = 0;

ss = 1; // Initially, deselect the slave

end

// Generate SPI Clock (SCLK) and select slave (SS)

always @(posedge clk or posedge rst) begin

if (rst) begin

sclk <= 0;

ss <= 1;

count <= 0;

end else begin

if (count == 0) ss <= 0; // Activate SS when transmission starts

sclk <= ~sclk; // Toggle SPI clock

end

end

// Transmit data on the positive edge of SCLK

always @(posedge sclk or posedge rst) begin

if (rst) begin

count <= 0;

mosi <= 0;

end else if (count < 8) begin

mosi <= data\_in[7 - count]; // Transmit MSB first

count <= count + 1;

end else begin

count <= 0;

ss <= 1; // Deactivate SS when transmission ends

end

end

// Receive data on the negative edge of SCLK

always @(negedge sclk or posedge rst) begin

if (rst) begin

data\_out <= 0;

mem <= 0;

end else if (count > 0 && count <= 8) begin

mem[7 - count] <= miso; // Store received bit from slave

end else if (count == 8) begin

data\_out <= mem; // Transfer received data to output

end

end

endmodule

**Slave Code:**

module spi\_slave (

input wire clk, // System clock

input wire rst, // Reset

input wire mosi, // Master Out Slave In

input wire sclk, // Serial Clock from Master

input wire ss, // Slave Select (Active Low)

output reg miso, // Master In Slave Out

input wire [7:0] data\_in, // Data to be sent to the master

output reg [7:0] data\_out // Data received from the master

);

reg [2:0] count; // 3-bit counter to count 8 bits (0-7)

reg [7:0] mem; // Register to store received data

initial begin

miso = 0;

end

// Receive data from the master on the positive edge of SCLK

always @(posedge sclk or posedge rst) begin

if (rst) begin

count <= 0;

data\_out <= 0;

end else if (!ss) begin // Only receive when SS is low (slave selected)

if (count < 8) begin

mem[7 - count] <= mosi; // Receive bit from master

count <= count + 1;

end else begin

data\_out <= mem; // Transfer received data to output

count <= 0; // Reset counter for next transmission

end

end

end

// Transmit data to the master on the negative edge of SCLK

always @(negedge sclk or posedge rst) begin

if (rst) begin

miso <= 0;

end else if (!ss && count < 8) begin

miso <= data\_in[7 - count]; // Send bit to master

end

end

endmodule

**Testbench:**

module spi\_master\_slave\_tb;

// Signals between Master and Slave

reg clk; // System clock

reg rst; // Reset signal

wire sclk; // SPI clock generated by master

wire ss; // Slave select (active low)

wire mosi; // Master Out Slave In

wire miso; // Master In Slave Out

reg [7:0] master\_data\_in; // Data sent by the master

reg [7:0] slave\_data\_in; // Data sent by the slave

wire [7:0] master\_data\_out; // Data received by the master

wire [7:0] slave\_data\_out; // Data received by the slave

// Clock generation (for system clock)

initial begin

clk = 0;

forever #5 clk = ~clk; // Generate a clock with a period of 10 time units

end

// Testbench process

initial begin

rst = 1;

master\_data\_in = 8'b10101010; // Example data to be sent from master

slave\_data\_in = 8'b11001100; // Example data to be sent from slave

#20 rst = 0; // Deassert reset after 20 time units

// Simulation running for some time to test data transfer

#500 $finish; // End simulation after 500 time units

end

// Instantiate SPI Master

spi\_master master (

.clk(clk),

.rst(rst),

.data\_in(master\_data\_in), // Data to be sent to slave

.miso(miso), // Data coming from slave

.mosi(mosi), // Data sent to slave

.sclk(sclk), // SPI clock

.ss(ss), // Slave select

.data\_out(master\_data\_out) // Data received from slave

);

// Instantiate SPI Slave

spi\_slave slave (

.clk(clk),

.rst(rst),

.mosi(mosi), // Data coming from master

.sclk(sclk), // SPI clock from master

.ss(ss), // Slave select from master

.miso(miso), // Data sent to master

.data\_in(slave\_data\_in), // Data to be sent to master

.data\_out(slave\_data\_out) // Data received from master

);

endmodule

* **Protocol 3(CAN):**

**Code:**

module CAN\_Frame(

input wire clk, // Clock signal

input wire rst\_n, // Reset signal (active low)

input wire start\_transmission, // Signal to start the transmission

input wire [10:0] id, // 11-bit identifier

input wire [7:0] data, // 8-bit data (1 byte for simplicity)

output reg tx // Transmit data to CAN bus

);

// CAN Frame fields

parameter SOF = 1'b0; // Start of Frame bit

parameter RTR = 1'b0; // Remote Transmission Request bit (dominant for data frame)

parameter IDE = 1'b0; // Identifier Extension bit (0 for 11-bit identifier)

parameter DLC = 4'd8; // Data Length Code (8 bytes)

parameter CRC = 15'b110110011001101; // Example 15-bit CRC for the frame

parameter ACK = 2'b11; // Acknowledge bits (recessive for no error)

parameter EOF = 7'b1111111; // End of Frame (7 recessive bits)

// State Machine states

localparam IDLE = 4'd0,

SOF\_STATE = 4'd1,

ID\_STATE = 4'd2,

RTR\_IDE\_STATE= 4'd3,

DLC\_STATE = 4'd4,

DATA\_STATE = 4'd5,

CRC\_STATE = 4'd6,

ACK\_STATE = 4'd7,

EOF\_STATE = 4'd8;

reg [3:0] state, next\_state;

reg [3:0] bit\_cnt; // Bit counter for frame fields

reg [3:0] byte\_cnt; // Byte counter for data transmission

// FSM to manage CAN frame transmission

always @(posedge clk or negedge rst\_n) begin

if (!rst\_n) begin

state <= IDLE;

bit\_cnt <= 4'd0;

byte\_cnt <= 4'd0;

tx <= 1'b1; // Default recessive state

end else begin

state <= next\_state;

// Transmit CAN frame bits

case (state)

IDLE: begin

tx <= 1'b1; // Recessive state

end

SOF\_STATE: begin

tx <= SOF;

end

ID\_STATE: begin

tx <= id[10 - bit\_cnt]; // Transmit 11-bit identifier

end

RTR\_IDE\_STATE: begin

if (bit\_cnt == 0)

tx <= RTR;

else

tx <= IDE; // Transmit IDE bit

end

DLC\_STATE: begin

tx <= DLC[3 - bit\_cnt]; // Transmit Data Length Code (4 bits)

end

DATA\_STATE: begin

tx <= data[7 - bit\_cnt]; // Transmit data byte by byte

end

CRC\_STATE: begin

tx <= CRC[14 - bit\_cnt]; // Transmit 15-bit CRC

end

ACK\_STATE: begin

tx <= ACK[1 - bit\_cnt]; // Transmit 2-bit ACK

end

EOF\_STATE: begin

tx <= EOF[6 - bit\_cnt]; // Transmit 7-bit End of Frame

end

endcase

// Bit counter logic

if (state != IDLE)

bit\_cnt <= bit\_cnt + 1;

else

bit\_cnt <= 4'd0;

end

end

// FSM Next State Logic

always @(\*) begin

next\_state = state;

case (state)

IDLE: begin

if (start\_transmission) // External signal to start frame transmission

next\_state = SOF\_STATE;

end

SOF\_STATE: begin

if (bit\_cnt == 0)

next\_state = ID\_STATE;

end

ID\_STATE: begin

if (bit\_cnt == 10)

next\_state = RTR\_IDE\_STATE;

end

RTR\_IDE\_STATE: begin

if (bit\_cnt == 1)

next\_state = DLC\_STATE;

end

DLC\_STATE: begin

if (bit\_cnt == 3)

next\_state = DATA\_STATE;

end

DATA\_STATE: begin

if (bit\_cnt == 7) begin

if (byte\_cnt == (DLC - 1))

next\_state = CRC\_STATE;

else begin

byte\_cnt = byte\_cnt + 1;

bit\_cnt = 0;

end

end

end

CRC\_STATE: begin

if (bit\_cnt == 14)

next\_state = ACK\_STATE;

end

ACK\_STATE: begin

if (bit\_cnt == 1)

next\_state = EOF\_STATE;

end

EOF\_STATE: begin

if (bit\_cnt == 6)

next\_state = IDLE;

end

endcase

end

endmodule

**Testbench:**

module CAN\_Frame\_tb;

// Inputs

reg clk;

reg rst\_n;

reg start\_transmission;

reg [10:0] id;

reg [7:0] data;

// Outputs

wire tx;

// Instantiate the CAN\_Frame module

CAN\_Frame uut (

.clk(clk),

.rst\_n(rst\_n),

.start\_transmission(start\_transmission),

.id(id),

.data(data),

.tx(tx)

);

// Clock generation: 10 ns period

initial clk = 0;

always #5 clk = ~clk; // Clock with 10ns period (50 MHz frequency)

// Test vector generation

initial begin

// Initialize Inputs

rst\_n = 0;

start\_transmission = 0;

id = 11'd0;

data = 8'd0;

// Apply reset for 20 ns

#20;

rst\_n = 1;

#20;

// Set ID and Data for transmission

id = 11'h7CF; // Example CAN ID

data = 8'hB5; // Example data byte

// Trigger transmission

start\_transmission = 1;

#10;

start\_transmission = 0;

// Wait for transmission to complete

#500; // Adjust the wait time according to FSM timing

// Finish simulation

$finish; // Stop the simulation after the frame is transmitted

end

// Monitor signals for debugging

initial begin

$monitor("Time=%0t | rst\_n=%b | start\_transmission=%b | id=0x%h | data=0x%h | tx=%b",

$time, rst\_n, start\_transmission, id, data, tx);

end

endmodule

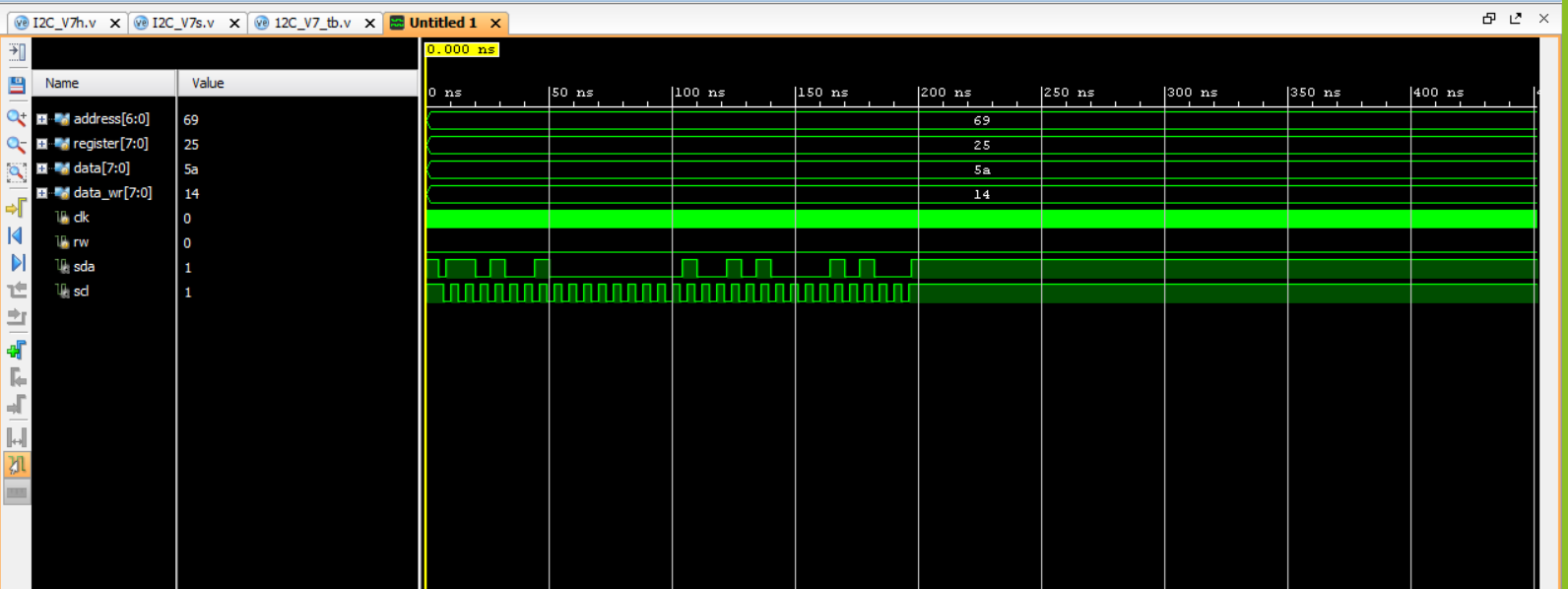
**5.2 Challenges faced and solutions implemented**

* Complexity of Protocol Design
* Simulation Errors and Debugging
* Version Control and Collaboration
* Time Management and Project Coordination

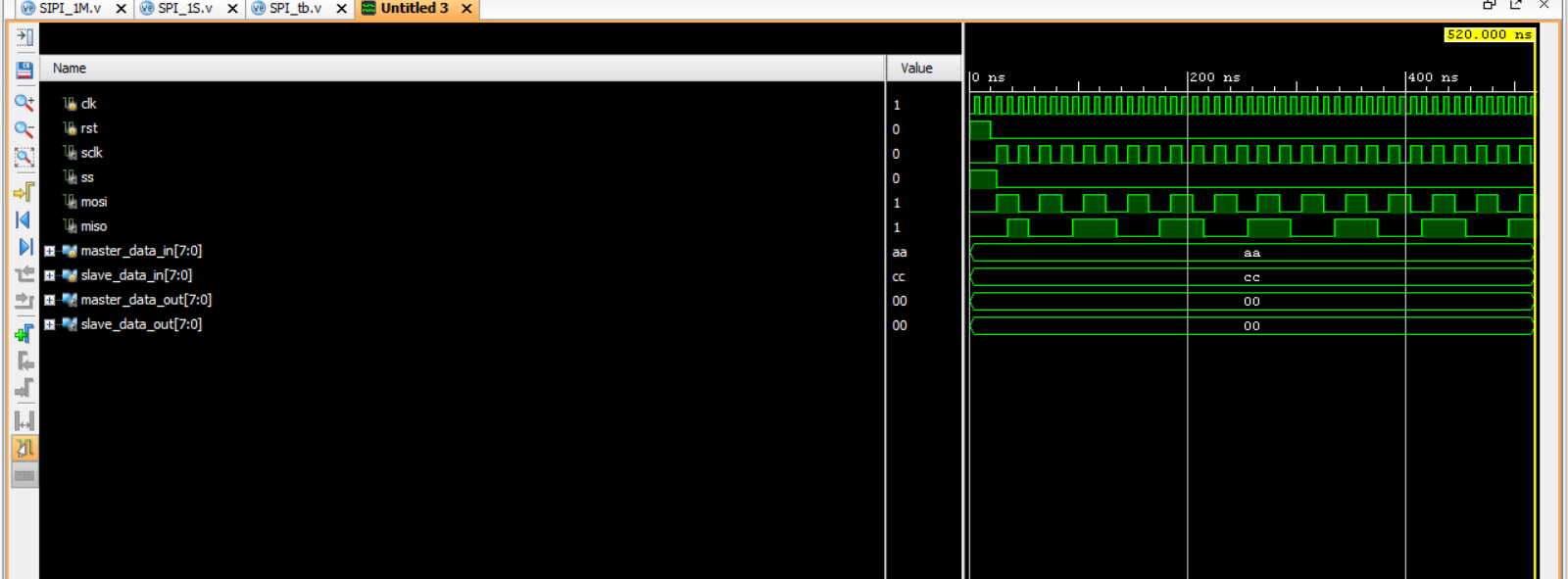
# **Chapter 6: Results**

## **6.1 outcomes**

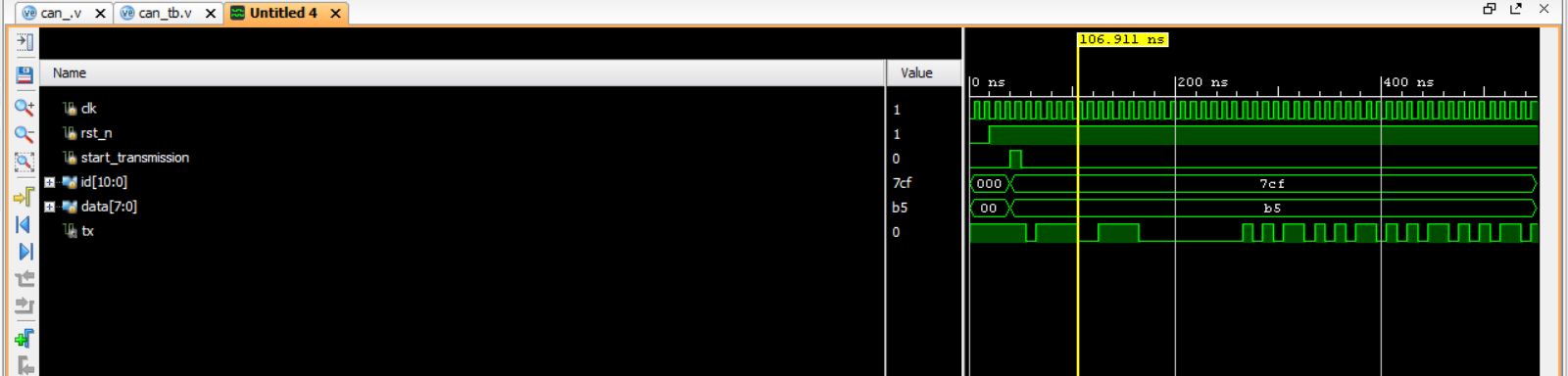
**Protocol 1 – I2C Graph Wave Form**



**Protocol 2 – SPI Graph Wave Form**



**Protocol 3 – CAN Graph Wave Form**



### **6.2 Interpretation of results**

The attached waveforms in the report provide a comprehensive view of the performance and functionality of the CAN, I2C, and SPI communication protocols as implemented in Verilog. Below is the interpretation of the key results obtained from the simulation:

* **I2C Protocol Waveforms**:

**Data Transmission:** The waveforms reveal successful clock (SCL) and data (SDA) signal synchronization, confirming that the master-slave communication operates as intended. The clear transitions in the SDA line during data transfer indicate that the protocol can handle multiple devices on the same bus effectively.

**Start and Stop Conditions:** The presence of distinct start and stop conditions in the waveforms validates the protocol's capability to initiate and terminate communication sessions accurately.

**Address Recognition:** The waveform analysis shows successful address recognition by the slave devices, confirming that the protocol can efficiently manage multiple slaves on the bus.

* **SPI Protocol Waveforms:**

**Clock Synchronization**: The waveforms demonstrate correct synchronization between the master and slave devices, with clear clock signals leading data transfers. This indicates that SPI can achieve high-speed communication reliably.

**Data Integrity:** The received data waveform matches the transmitted data, confirming that the SPI protocol maintains high data integrity during transfer, essential for applications requiring low latency and fast communication.

* **CAN Protocol Waveforms:**

**Message Framing:** The waveforms show clear delineation of message start and end bits, confirming proper message framing. This indicates that the protocol can effectively delineate messages in a multi-node environment.

**Error Detection:** The error detection waveforms illustrate that any corrupted messages were flagged correctly, showcasing the robustness of the CAN protocol in maintaining data integrity.

# **Chapter 7: Conclusion**

* The project successfully implemented and simulated the CAN, I2C, and SPI communication protocols in Verilog, showcasing a understanding of embedded systems.
* Each protocol was designed and rigorously tested to meet industry standards, focusing on signal integrity, timing accuracy, and reliable data transfer.
* Extensive simulations and waveform analysis were conducted to verify the real-time performance and practical applicability of the protocols.
* The project demonstrated the significance of communication protocols in key sectors like automotive, industrial, and consumer electronics.
* Strong teamwork and individual contributions ensured the project’s success, laying a solid foundation for future work in hardware design and communication technologies.

**Chapter 8: Future Work**

#### **Hardware Implementation**: The next phase of this project involves implementing the CAN, I2C, and SPI protocols on a Xilinx FPGA development kit. This will allow real-world testing and validation beyond simulations.

#### **Real-Time Testing:** By deploying the protocols on the Xilinx kit, we can evaluate their real-time performance in handling data transfers between connected devices such as sensors, memory units, and controllers.

#### **References**

1. Shaila C K , G Manoj , Divya PS and Vijila M “Functional Verification of SPI Protocol using UVM based on AMBA Architecture for Flash Memory Applications” -2023 4th International Conference on Signal Processing and Communication (ICSPC) | 23-24 March 2023.
2. [Naveen RS](https://ieeexplore.ieee.org/author/146888140003147); [Sanjay S](https://ieeexplore.ieee.org/author/37089741291); [Senthil Murugan A](https://ieeexplore.ieee.org/author/773306600785226); [Surya Prakash M](https://ieeexplore.ieee.org/author/37087756433); [Mukuntharaj C](https://ieeexplore.ieee.org/author/473412329042561) “Implementation Of I2C Protocol With Adaptive Baud Rate Using Verilog” -[2024 7th International Conference on Devices, Circuits and Systems (ICDCS)](https://ieeexplore.ieee.org/xpl/conhome/10560478/proceeding) **-**Print on Demand(PoD) ISBN:979-8-3503-5048-7.
3. [Saket Srivastava](https://ieeexplore.ieee.org/author/37085471661); [Peter Hobden](https://ieeexplore.ieee.org/author/37991166800) “Low Cost FPGA Implementation of a SPI over High Speed Optical SerDes”[2018 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS)](https://ieeexplore.ieee.org/xpl/conhome/8716489/proceeding)
4. [Jun-Cheol Lee](https://ieeexplore.ieee.org/author/37089752400); [Tae-Oh Kim](https://ieeexplore.ieee.org/author/37089753644); [Joo-Hyung Chae](https://ieeexplore.ieee.org/author/37085644699) “Module Implementation and Simulation of Timing Constraint Check Function of I2C Protocol Using Verilog “ Date Added to IEEE *Xplore*: 10 March 2023.
5. [Guilherme Marcon Zago](https://ieeexplore.ieee.org/author/37086265042); [Edison Pignaton de Freitas](https://ieeexplore.ieee.org/author/37085970699) “ A Quantitative Performance Study on CAN and CAN FD Vehicular Networks “ I[EEE Transactions on Industrial Electronics](https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=41) ( Volume: 65, [Issue: 5](https://ieeexplore.ieee.org/xpl/tocresult.jsp?isnumber=8260593&punumber=41), May 2018) Page(s): 4413 – 4422.
6. Kwonneung Cho , Jeongeun Kim , DoYound Choi , Yung Hyun Yoon , Jung Hwan Oh and Seung Eun Lee “An FPGA-Based ECU for Remote Reconfiguration in Automotive Systems” *Micromachines* 2021, *12*(11), 1309; <https://doi.org/10.3390/mi12111309>.
7. Shreejith Shanker and Suhaib A Fahmy “Extensible FlexRay Communication Controller for FPGA-Based Automotive Systems” IEEE transactions on vehicular technology, vol. 64, no. 2, february 2015.