Project: Implementation of Communication Protocol on FPGA

Implementation - Continuous progress review

I. Introduction

The main goal of my project is to design and implement a communication system that connects a computer, FPGA, and Arduino using two different protocols: **UART (Universal Asynchronous Receiver and Transmitter)** and **SPI (Serial Peripheral Interface)**. The project shows how data can move from one device to another, be processed, and then sent back, with the FPGA acting as the main controller.

In this implementation, the following flow takes place:

1. A terminal on the PC sends data to the FPGA through a USB-to-UART adaptor.
2. The FPGA receives the data and sends it further to the Arduino using SPI, where the FPGA is the master and Arduino is the slave.
3. The Arduino increments the data and sends it back to the FPGA through SPI.
4. The FPGA then sends this updated data back to the terminal through UART.
5. At the same time, the Arduino also shows the data on another terminal window connected through its serial port in both decimal and hexadecimal format.

For development, I used **Quartus** for coding and programming the FPGA, **ModelSim** for simulation of the protocols (UART, SPI, and I2C), and the **Arduino IDE** for uploading code to the Arduino. I also started building a **web dashboard** to show communication waveforms, where the port detection is working but the waveform part is not yet finished.

II. Code Structure and Organization

The code is written in a modular and organized way so that each protocol and device has its own file or section. Below is the structure:

FPGA (Verilog in Quartus)

- **uart_rx.v** → Module for receiving UART data.
- **uart_tx.v** → Module for transmitting UART data.
- **spi_master.v** → Module for controlling SPI communication from FPGA as master.
- **fpga_top.v** → Top-level file that connects UART modules with SPI master and coordinates data flow.

- **baud_gen.v** → Baud rate generator for UART timing.
- **testbenches** → Separate testbench files for simulating UART and SPI in ModelSim.

Arduino (C code in Arduino IDE)

- **spi_slave.ino** → Code to configure Arduino as SPI slave. It receives data, increments it, and sends it back. Also sends data to another terminal through Serial.print() in decimal and hexadecimal format.

Web Dashboard (HTML + JavaScript + Python backend – under development)

- **dashboard.html** → Basic webpage with port detection.
- **script.js** → Handles port scanning and will later be used for waveform plotting.
- **server.py** (planned) → For connecting FPGA/Arduino outputs to the dashboard.

Version Control

All code is managed in a **GitHub repository**. Each commit clearly mentions changes, such as "added SPI slave code," "tested UART in ModelSim," or "connected top-level FPGA module." This helps track the progress step by step.

III. Implementation Details

UART Communication (FPGA ↔ PC)

- Implemented using Verilog modules for **transmitter** and **receiver**.
- The **baud rate generator** is used to match the terminal speed (9600 bps for testing, can be changed).
- Data from PC terminal enters FPGA through UART RX and is stored in a register.
- After processing (through SPI and Arduino), the data is sent back through UART TX.

SPI Communication (FPGA ↔ Arduino)

- FPGA works as **SPI master**. It controls **SCLK (clock), MOSI (Master Out Slave In), and CS (chip select)** lines.
- Arduino is programmed as **SPI slave**. It receives data from FPGA and increments it.
- The incremented data is sent back to FPGA through the **MISO (Master In Slave Out)** line.
- This ensures a synchronous transfer where FPGA decides the clock.

Arduino Processing

- When Arduino receives data (example: 0x05), it increments it (to 0x06).
- This incremented value is sent back to FPGA via SPI.

- Additionally, Arduino prints the received and incremented values in **decimal** and **hexadecimal** format on another serial port terminal.

FPGA Controller (Top Module)

- This is the central part where UART and SPI are connected.
- Steps:
    1. Get data from UART RX.
    2. Send to SPI Master.
    3. Collect response from SPI Slave (Arduino).
    4. Send back to UART TX.
- This completes the communication cycle.

Web Dashboard

- Currently detects connected ports.
- Plan is to take the communication logs and plot waveforms of UART and SPI in real time.
- Future upgrade: show both digital waveforms (timing diagrams) and data logs on the dashboard.

IV. Testing Procedures and Results

Testing was done in different stages:

1. Simulation in ModelSim

- UART transmitter and receiver were tested with different baud rates.
- SPI master and slave modules were simulated to check clock synchronization and data transfer.
- I2C was also simulated as an additional protocol to explore adding more devices in future.

2. Hardware Testing with FPGA and Arduino

- Connected USB-UART adaptor to FPGA and opened PC terminal with Putty.
- Sent data like 0x05 from terminal.
- Observed the incremented response 0x06 returned to terminal after going through FPGA and Arduino.
- Opened a second terminal connected to Arduino serial port to view received and incremented values in both decimal and hexadecimal.

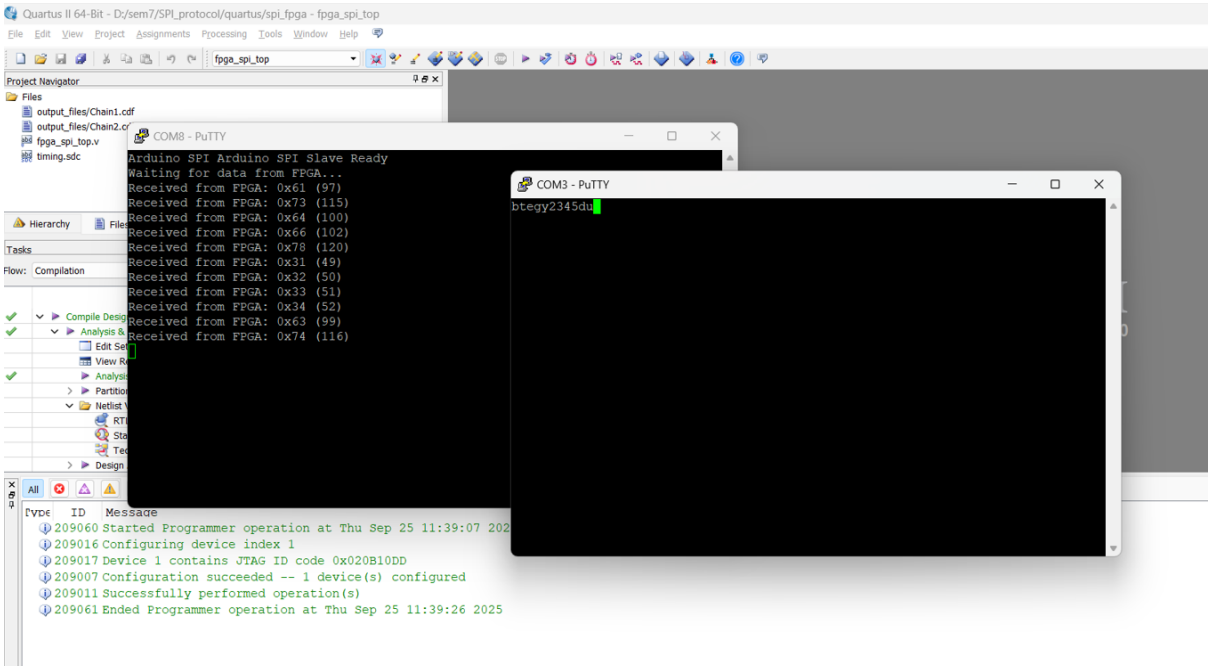
- Verified multiple test cases with different values (e.g., sending 0x10 returned 0x11).

3. Integration Testing

- Confirmed that both UART and SPI work together in real hardware.
- FPGA successfully forwards data between UART and SPI modules.
- Arduino increments data and both FPGA and Arduino terminals show correct outputs.
- Integration is seamless because each protocol module was first tested separately before combining.

Evidence of Testing

- Screenshots from Putty terminals showing sent and received values.
- Logs from Arduino serial monitor displaying decimal and hex outputs.
- ModelSim waveforms showing UART and SPI transactions.



Link of testing video:
https://drive.google.com/drive/folders/1en1WtyRCgajVP2L9R_DOUQ6kZULVtpDS?usp=drive_link

V. Instructions for Running the System

To run the system, follow these steps:

1. **FPGA Setup**

    - Open Quartus project.
    - Compile Verilog code.
    - Load bitstream to FPGA board using USB Blaster.
2. **Arduino Setup**
    - Open spi_slave.ino in Arduino IDE.
    - Upload to Arduino board.
3. **Connections**
    - Connect USB-UART adaptor to FPGA UART pins.
    - Connect SPI lines: MOSI, MISO, SCLK, CS between FPGA and Arduino.
    - Connect Arduino USB cable to PC for second terminal monitoring.
4. **Terminal Setup**
    - Open Putty (or any terminal) for USB-UART with settings: 9600 baud, 8 data bits, no parity, 1 stop bit.
    - Open another terminal for Arduino COM port.
5. **Testing**
    - Send a value from the first terminal.
    - Observe incremented result back in the same terminal.
    - Check Arduino terminal for logs in decimal and hex.

Integration Across Components

This project integrates multiple parts:

- **UART module in FPGA** connects PC terminal with FPGA.
- **SPI module in FPGA** connects FPGA with Arduino.
- **Arduino processing code** provides increment functionality.
- **FPGA top module** ensures smooth flow of data between UART and SPI.
- **Web dashboard** (in progress) adds a modern interface for visualizing communication.

The integration was successful because each part was first tested independently, then combined step by step. The final system shows correct data flow, increments at Arduino, and consistent results on both terminals.

Conclusion

The implementation shows a **fully functional prototype** where UART and SPI protocols work together on FPGA and Arduino to send, process, and return data. The code is modular, clean, and tested in both simulation and hardware. Integration of UART, SPI, and Arduino is seamless, with results observed in terminals.

Even though the web dashboard is still under development, the system already demonstrates the core objectives of reliable communication and protocol integration. The future work will focus on completing the dashboard with real-time waveforms and extending the project with I2C and more devices.

This project proves the capability of FPGA to act as a central controller for multi-protocol communication and shows how modular coding, systematic testing, and careful integration lead to a robust and scalable system.