Ultra-Low-Power Inter-Integrated Circuit Implementation for Fabric-Based Self-Powered Systems

A

Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In partial fulfillment

of the requirements for the degree

Master of Science

by

Zhenghong Chen

April 2023

Abstract

With the advance of various energy harvesting and storage devices, self-powered wearable systems have gained significant interest in the past few years. In addition, the growing focus on fibers with digital devices is driving the potential for fabrics with digital systems for health monitoring and human-machine interfaces. While such digital-enable fabric systems consist of various distributed digital devices, a low-power serial communication component is in high demand. The standard Serial Peripheral Interface (SPI) commonly used in device connection is unsuitable for communication in fiber because the bit width of the slave select signal (SS) increases with the number of networked devices.

To address this research gap, we propose using Inter-Integrated Circuit (I^2C) for communication in fiber and presenting the SPI-I^2C Protocol Converter chip for data transfer from the SPI to the I^2C bus. Our research involves designing and implementing a low-power I^2C module with tunable frequency, data width, and output voltage. The I^2C module on the chip can work at 0.4 V supply voltage with only 2.6 nW static power. In addition, we designed the I^2C controller on the chip that can adaptively adjust the drive capability to balance power consumption and reliability when more devices are connected. Finally, we integrate the I^2C module into the System-on-Chip (SoC) using the ARM Advanced Peripheral Bus (APB) to complete the digital-enabled fabric system.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Benton H. Calhoun, for his invaluable advice, unwavering support, and endless patience throughout my master's degree.

I also want to extend my heartfelt appreciation to my friends and collaborators, Xinjian Liu and Suprio Bhattacharya, who provided critical assistance in digital circuit design, tape-out processes, and chip testing. Their expertise, dedication, and hard work were essential in bringing this project to fruition.

I want to thank Yimin Gao from the University of Virginia High-Performance Low-Power Lab for his advice throughout designing the digital circuit and optimizing the SoC. Thank you, Yimin, for your support and contribution to this work.

I wish to thank my committee members, Prof. Todd A. DeLong and Prof. Steven M. Bowers, for their help and support throughout my master's degree.

Finally, I thank my parents for their selfless love and care. With their help, I have a better living and learning environment, as well as more opportunities for growth.

Contents

Chapter 1

Introduction

In recent years, there has been a significant surge in interest in developing self-powered systems for various applications, including wireless sensors [1] [2] and Internet of Things (IoT) devices [3] [4]. These systems utilize energy harvesting techniques, such as solar, thermal, or kinetic energy harvesting, to generate energy from the environment and sustain their operation without needing batteries or external power sources. Self-powered systems [5] offer numerous advantages, including reduced maintenance costs, improved reliability, and increased flexibility in terms of where and how they can be used. To achieve self-powered, wearable electronics have integrated energy harvesters such as photovoltaic cells, piezoelectric generators [6] [7], and enzymatic or microbial biofuel cells [8] to balance energy consumption. These wearable systems [9] can eliminate the need for frequent recharges and wired power transmission, allowing for uninterrupted use and an improved user experience.

Current wearable devices [10] [11] are limited in their ability to collect physiological data due to their relatively rigid structure, unacceptable size, and limited contact area with the body. In contrast, traditional clothing fabrics offer a promising opportunity in power harvesting and physiological variables analysis since they already come into contact with large surface areas of the human body. Fibers, the basic building blocks of fabrics, offer a natural candidate for enabling sensory, memory, and other digital functions. Existing technologies have enabled digital components into flexible polymer fiber bundles, allowing access to the collection of devices inside the fiber through connection ports at the end of the fiber [12] [13]. This presents an opportunity to realize digital systems integration into clothing, providing a solution for the comfortable and senseless self-powered wearable system.

The Fabric-Based Distributed System aims to address the challenges posed by in-fiber digital devices size limited, which dimensions can not exceed 0.6mm. In this distributed system, the traditional structure is divided into smaller subsystems that can be distributed throughout a garment. Each subsystem can work independently, allowing for greater flexibility and customization. Besides the need to design each chip, the challenge is finding ways to interconnect these subsystems and chips in a reliable and efficient manner. We hope to find a suitable solution by exploring Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C) communication protocols to achieve interconnection while minimizing power consumption.

1.1 Related Work

1.1.1 I^2C Protocol Based Multi-Device System

This custom home automation system [14] with eight devices interconnected via the I^2C protocol is programmable and modular. The I^2C protocol has the characteristics of simple structure and high compatibility in building a multi-device system. This user provides a technical basis for customizing devices in the system. The novelty of the subject lies in the combination of communication protocols and software tools to obtain reconfigurable, open-source, custom-made applications.

1.1.2 SPI to I^2C Protocol Conversion

SPI to I^2C Protocol Conversion Unit [15] proposes a novel scheme to convert SPI to I^2C by connecting the output of the SPI slave to the input of the I^2C master and using FIFOs to build the protocol conversion unit. This solution solves the problem of SPI and I^2C being out of sync and operating at different frequencies, enabling the interconnection of these two protocol systems in low-power embedded systems. Although the proposed design has been successfully implemented on Field Programmable Gate Array (FPGA), it only supports one-way conversion from SPI to I^2C. Nonetheless, the approach presented in this paper provides a promising basis for future work on bidirectional protocol translation.

1.2 Motivation

1.2.1 Fabric-Based Self-Powered System

The Self-Powered Fabric-Based System is a densely integrated digital system comprising interconnected functional components distributed throughout various positions on clothing. To enhance clothing comfort, integrated circuits must be divided into multiple ultra-low-power microchips that form distributed subsystems, enabling each subsystem to function independently. Moreover, the system's power consumption must be maintained within the nanowatts (nW) range, as limited by the size of the battery and energy harvester.

In the previous design, the SPI protocol was utilized as a communication tool for components due to its low power consumption and high efficiency, making it possible to achieve internal communication within the subsystem. However, regarding global communication, SPI is not ideal for several reasons. Each subsystem may act as a master, and SPI only supports single-master systems when interconnected. This limitation makes SPI unsuitable for global communication, where multiple subsystems need to communicate with each other. In addition, as the number of networked devices increases, the bit width of the SPI slave select signal (SS) also increases. This limitation makes SPI unsuitable for communication in fiber because the fiber only allows for limited transmission lines, which limits the number of devices that can be connected.

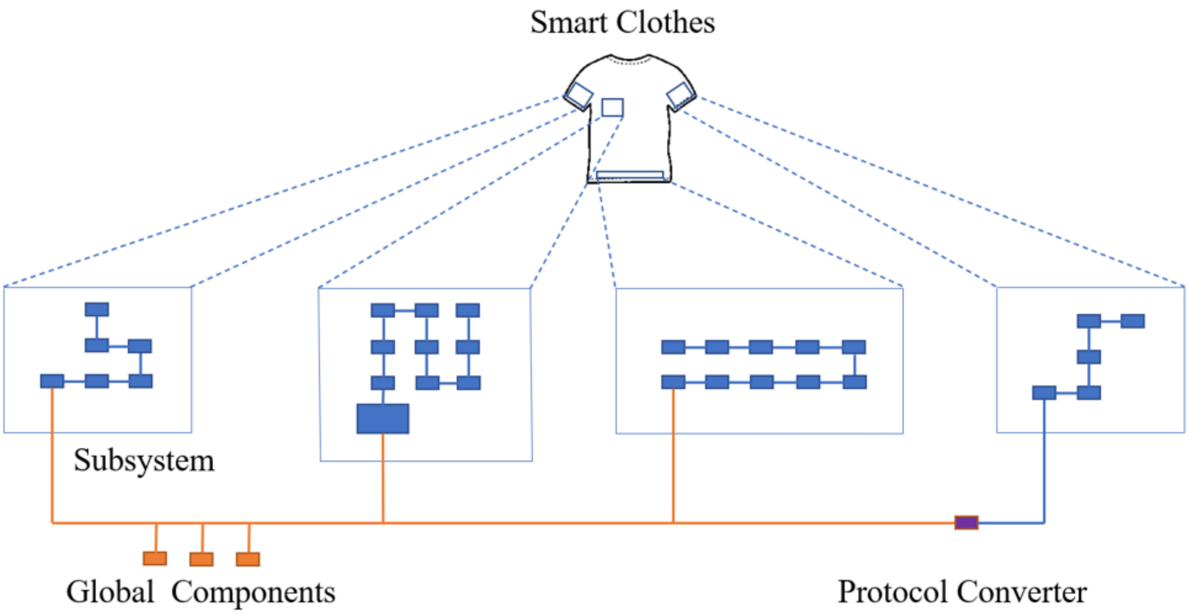


Figure 1.1: Fabric-Based Self-Powered System

1.2.2 Communication for Ultra-Low-Power System

Sensors play a critical role in human-computer interaction and are indispensable to such systems. Achieving low-power or self-powered systems requires using sensors with the lowest supply voltage available. While several sensors are available on the Texas Instruments website, only a few options meet the controlled supply voltage requirements of the envisioned system, which is between 0.6-1.8V. While these sensors may not be suitable for the Self-Powered Fabric-Based System due to more integrated design requirements for the system, they are highly relevant when building a low-power embedded system where voltage and power consumption are top priorities. In general, sensors with I^2C protocol have lower supply voltage requirements than other sensors.

Our current System on Chip (SoC) only supports SPI components, which means that a conversion structure from SPI to I^2C needs to be designed or an I^2C module needs to be incorporated into the SoC to achieve the desired functionality.

Table 1.1: Sensors

Classification	**Product**	**Interface**	**Supply Voltage (min) (V)**
Temperature Sensors	TMP114 [16]	I^2C, SMBus	1.08
	TMP126 [17]	SPI	1.26
	TMP144 [18]	UART	1.4
Humidity Sensors	HDC3022 [19]	I^2C	1.62
Ambient light sensors	OPT3005 [20]	I^2C	1.6

1.3 Thesis statement

1. Design the customizable I^2C module with tunable frequency, data width, and output voltage. Power consumption measurement and analysis assess the viability of embedding the module in a ultra-low-power system.

2. Extend the function and efficiency of the distributed system through implementation of SPI-I^2C Protocol Converter chip.

3. Optimize the System-on-Chip (SoC) performance with integrate an I^2C module.

Chapter 2

Background Theory

This chapter will provide an overview of communication techniques used in integrated circuits, including using buses for communication, the underlying theory of interconnects, and introducing bus protocols such as I^{2}C and SPI.

2.1 Communication in Integrated Circuits

Due to the modular structure of distributed systems, integrated circuit (IC) modules with distinct functions are packaged separately in the chip, requiring communication as an essential aspect. Communication within ICs can be achieved through serial or parallel transfers, which can be synchronous or asynchronous.

Serial communication involves transmitting data bits sequentially, one after the other. In contrast, parallel communication uses multiple transmission lines to transmit each bit simultaneously, resulting in high throughput but increased power consumption and area costs due to the need for more wires. Synchronous communication utilizes a dedicated clock signal to synchronize transfers between components, with the clock signal indicating when the receiver should sample the data on the line. On the other hand, asynchronous communication uses a handshake protocol to synchronize transfers without relying on a separate clock signal line. However, this approach can result in higher overhead, as many transmitted bits are utilized only for control purposes.

A bus is a set of wires that permits signals to be transmitted between one or more integrated circuits. In such communication, the device that initiates communication is referred to as the Master, and the device that responds to the communication request is known as the Slave. Moreover, the bridge connects different buses while a decoder selects the appropriate slave for receiving data. Later in the thesis, we will discuss interfaces like ARM Advanced Microcontroller Bus Architecture (AMBA), Advanced eXtensible Interface 4 (AXI4), and Advanced Peripheral Bus (APB) that utilize these concepts in signal transmission.

2.2 Serial Communication Protocols

Serial Communication Protocols suit low-power embedded systems with simple structures and high transmission efficiency. In most cases, SPI is selected because it has the advantages of high transmission frequency, simple module structure, and low power consumption compared with I2C. In contrast, the advantages of I2C are high reliability, fewer transmission lines, and support for multiple masters. This thesis presents the design and implementation of the SPI-I^{2}C protocol converter chip. The following section provides an introduction to the SPI and I^{2}C communication protocols and compares their respective characteristics.

2.2.1 Serial Peripheral Interface (SPI)

Serial Peripheral Interface (SPI) [21] [22] [23] is a synchronous serial full-duplex data transmission communication method introduced by Motorola. It is generally considered that the SPI protocol is a three-wire synchronous bus, but to connect multiple slave devices, each slave device needs to be equipped with a Chip Select signal (CS/SS). As the number of components connected to the system increases, the SPI master has to be designed with more chip-select signal lines to maintain

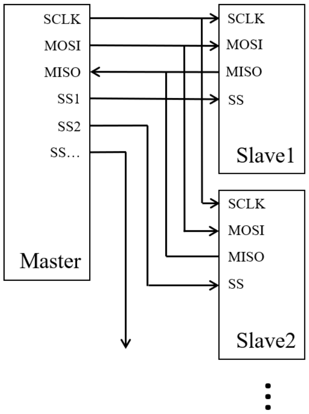


Figure 2.1: SPI Structure

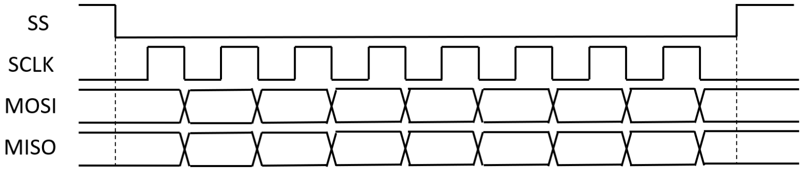


Figure 2.2: SPI Timing Diagram

control over all slaves in the system.

Table 2.1: I^{2}C Lines List

Classification	**Signal Provider**	**Definition**
SCLK	Master	Serial Clock
MOSI	Master	Master Out Slave In
MISO	Slave	Master In Slave Out
CS/SS	Master	Chip Select, Slave Select

To initiate a read/write operation in SPI, the master device first lowers the active SS line corresponding to the target slave and then sends clock pulses on the Serial Clock (SCLK). Depending on the mode selected, the output signal may be toggled by either the rising or falling edge of the clock pulse, while the input signal is sampled on the opposite edge. To perform a write operation, the master device transmits data on Master-Out-Slave-In (MOSI), while a read operation involves sampling Master-In-Slave-Out (MISO). Notably, SPI offers four different modes, each defining the operation at which specific edge and stable level value of the clock signal.

The SPI protocol offers a simple communication process that separates input and output data transmission for the master and slave. This separation eliminates the need for additional slave address checks during communication since each slave has a separate chip select line. However, as the number of devices increases, more chip-select signal lines are required, which can lead to increase area and power consumption. In addition, the SPI protocol only supports one Master in the system, which impedes establishing crossover between multiple chips.

2.2.2 Inter-Integrated Circuit (I^{2}C)

I^{2}C or I2C [24] [25] is a simple bidirectional two-wire bus protocol standard developed by Philips, now NXP Semiconductors. I^{2}C uses two signal lines, Serial Data (SDA) and Serial Clock (SCL), and does not rely on a physical chip selection signal or arbitration logic circuit. Instead, each I^{2}C device has a unique seven-bit address for protocol handshake. I^{2}C supports multiple master devices, and the protocol specification designates the initiating device as the master, while the all other devices on the bus are designated slaves.

Table 2.2: I^{2}C Lines List

Name	**Signal Provider**	**Definition**
SDA	Master, Slave	Serial Data
SCL	Master	Serial Clock

The I^{2}C communication process consists of several steps. First, the master device sends a START signal, followed by a 7-bit device address and a bit for describing read or write operation. Upon receiving the data, the slave devices compare the address with their chip address to determine whether it is the intended recipient. If the comparison fails, the device waits until a STOP signal is received. In contrast, a successful comparison prompts the device to send an acknowledgment signal (ACK) back to the master. After receiving the ACK, the master can start transmitting or receiving data in 8-bit frames, with each frame followed by a one-bit response signal. Specifically, the master sends data, and the slave responds during write operations, while the reverse is true during read operations. When the transmission is complete, the master sends a STOP signal to release the bus, allowing other devices to access it, and all devices return to their initial state.

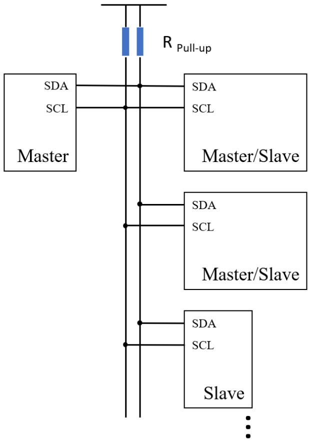


Figure 2.3: I^{2}C Structure

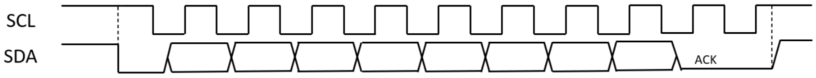


Figure 2.4: I^{2}C Timing diagram

Both the SCL and SDA lines in I^{2}C communication are open-drain and do not actively drive the bus high, ensuring that signal conflicts do not occur on the bus. When a device pulls the line to the ground, it transmits logic 0, and when it releases the line, the line returns to an idle state, transmitting logic 1. The following chapter will provide more detailed information on the specific structure and operation of I^{2}C.

Compared to the standard SPI protocol, which requires at least four signal lines and additional logic for multi-master structures, the I^{2}C protocol requires only two. Even when using multiple slave devices, the number of signals needed remains low. Although the I^{2}C protocol has a limited 7-bit address space, running out of 128 available addresses is rare. One significant advantage of the I^{2}C protocol is the lightweight architecture Open-Drain implements multi-master device arbitration and routing. Moreover, read and write timing is relatively fixed and uniform, and the device driver is straightforward to write. Considering all these factors, we have decided to replace the SPI protocol with the I^{2}C protocol in our current SoC design.

Chapter 3

Design Process of I^{2}C

This chapter presents the complete design process for the I^{2}C module. First, we formulated the design specifications by analyzing the I^{2}C protocol articles [26] [27] [28] [29]. Next, we used Verilog to complete the Register Transfer Level (RTL) design for the I^{2}C master and slave components. Then, we integrated these components and controlled them using an enable signal, resulting in higher integration and port savings. Additionally, we will conduct RTL simulation and analysis of the two design stages using Modelsim [30].

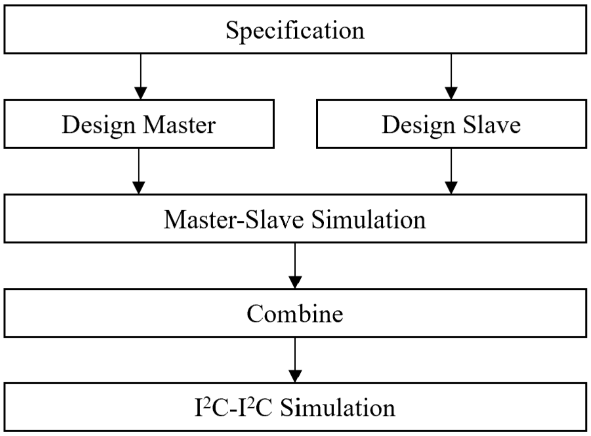


Figure 3.1: Flowchart of the Design Process

3.1 I^{2}C Components Specification

To discuss the design details of I^{2}C, this section will cover several aspects. Firstly, the port planning for the I^{2}C master and slave will be discussed. Next, the state machine for both read and write operations will be presented. Finally, the complete architecture block diagram will be introduced.

3.1.1 I^{2}C Master and Slave Architecture

The planning architecture for the I^{2}C master and slave modules is illustrated in the Fig.3.2, showing the input and output signals on either side of the block. The input signals include control signals, data input, SDA and SCL input signals, while the output signals consist of operating status display, data output, and SDA, SCL output control signals.

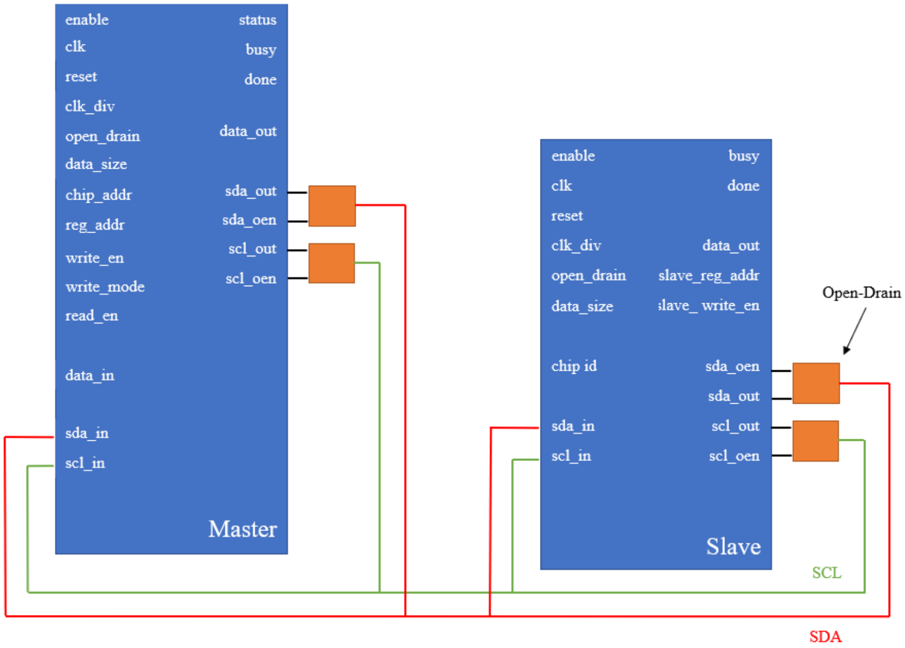


Figure 3.2: I^{2}C Master and Slave Architecture

The open-drain structure used for bidirectional communication is implemented through analog circuit design, which is not included in the digital design RTL. However, this behavior can be described using simple state statements in the testbench but cannot be achieved through synthesis.

3.1.2 I^{2}C module Port List

Since there are many repeated control signals in the port list in this chapter, the total ports are divided into five categories, Global, Master Logic, Slave Logic, Data path, SDA, and SCL. The I^{2}C port distribution described in this chapter is only used in this design.

Table 3.1: I^{2}C module Shared Port List

Classification	**Port**	**Input/Output**	**Function**
Global Setup	enable	Input	Master/Slave mode Switch
	clk	Input	System Clock
	reset	Input	Reset signal
	clk_div	Input	Clock Divider
	open_drain	Input	Open Drain Switch
	data_size	Input	Transmission data length
	done	Output	Finish Status Signal
	busy	Output	Work Status Signal
Data Path	data_in	Input	Data Input
	data_out	Output	Data Output
SDA and SCL	sda_in	Input	Serial Data Input
	sda_out	Output	Serial Data Output
	sda_oen	Output	Serial Data Output Enable
	scl_in	Input	Serial clock Input
	scl_out	Output	Serial clock Output
	scl_oen	Output	Serial clock Output Enable

The setting of the Enable signal does not entirely shut down the entire system but selects between master and slave. By integrating the I^{2}C master and slave components, controlled by the enable signal, we avoid any confusion that may arise from their simultaneous operation. When the enable signal is high, the master component works, and the slave is disabled. Conversely, when the enable signal is low, the slave is in the standby state.

The I^{2}C design is designed to provide flexibility in data length, allowing the switching between 1-byte and 2-byte transmission modes by simply adjusting the input data_size. Reviewing the actual usage of the I^{2}C protocol in the I^{2}C-based sensor purchased from Texas Instruments Incorporated shows that most data length in each communication is 2 Bytes. There also are some products whose data length in each communication is 1 Byte. Therefore, our design supports both data lengths to ensure communication with a wide range of I^{2}C-based sensors.

The input clock serves as the primary operating frequency for the I^{2}C module, while the clock divider is responsible for reducing the SCL output frequency. Modelsim and Cadence Spectre simulations show that a clock divider of at least 2 is required for stable operation. To ensure added stability and reliability, we have set the clock divider to 10 by default for all Modelsim and Cadence Spectre simulations and actual chip testing.

The several status ports are essential for checking the working status of the I^{2}C interface and facilitating the debugging of I^{2}C components. The Status port is particularly useful for monitoring the status of the I^{2}C master, while the Done and Busy ports ensure that I^{2}C transactions are completed. Additionally, these ports are reserved for the convenience of supervising the I^{2}C reservation and preparing for the SoC interrupt system design.

The I^{2}C master initiates the write or read operation when the write_en or read_en signal is high. If the slave does not support adjusting the data length, the write mode signal allows multiple I^{2}C write operations to be performed continuously. When the write mode signal is high, the master's state machine skips the stop state and proceeds directly to the next write operation. This design is intended to enhance data transmission efficiency, particularly when transmitting large amounts of data.

The output register slave_reg_addr is specifically designed to store the address provided by the

Table 3.2: I^{2}C module Independent Port List

Classification	**Port**	**Input/Output**	**Function**
Master Logic	chip_addr	Input	Target Chip Address
	reg_addr	Input	Target Register Address
	write_en	Input	Write Operation Start Signal
	write_mode	Input	Write Operation Mode Switch
	read_en	Input	Read Operation Start Signal
	status	Output	Master State Machine Status
Slave Logic	chip_id	Input	Self Chip Address
	slave_reg_addr	Output	Slave Register address
	slave_write_en	Output	Slave write enable

master. Upon receipt of the slave_write_en signal, the slave will then store the incoming data at the address stored in slave_reg_addr.

3.1.3 I^{2}C Integrated Architecture

Once the RTL designs of both the master and slave are complete, we merge them together to facilitate the synthesis and placement process described in the next chapter. However, this merging process can pose challenges due to the different port designs of the master and slave. To address this issue, we have created a complete structure diagram of the I^{2}C system, as shown in Fig.3.3, which includes the master and slave components mentioned previously but also the data register for controlling signals. The shared ports are integrated, while the independent ports are placed separately. The enable signal in the output ports controls the multiplexer to determine whether the output comes from the master or slave. The control signal is the input signal of the system, and the Enable signal can control the work of the master or slave, so the influence of the input signal on closing the device can be ignored. But for Data, the input and output registers of Master are shared by 16-Bit and Slave. In addition, Slave additionally designed a set of 16-Bit registers to temporarily place the transmitted data stored in different addresses.

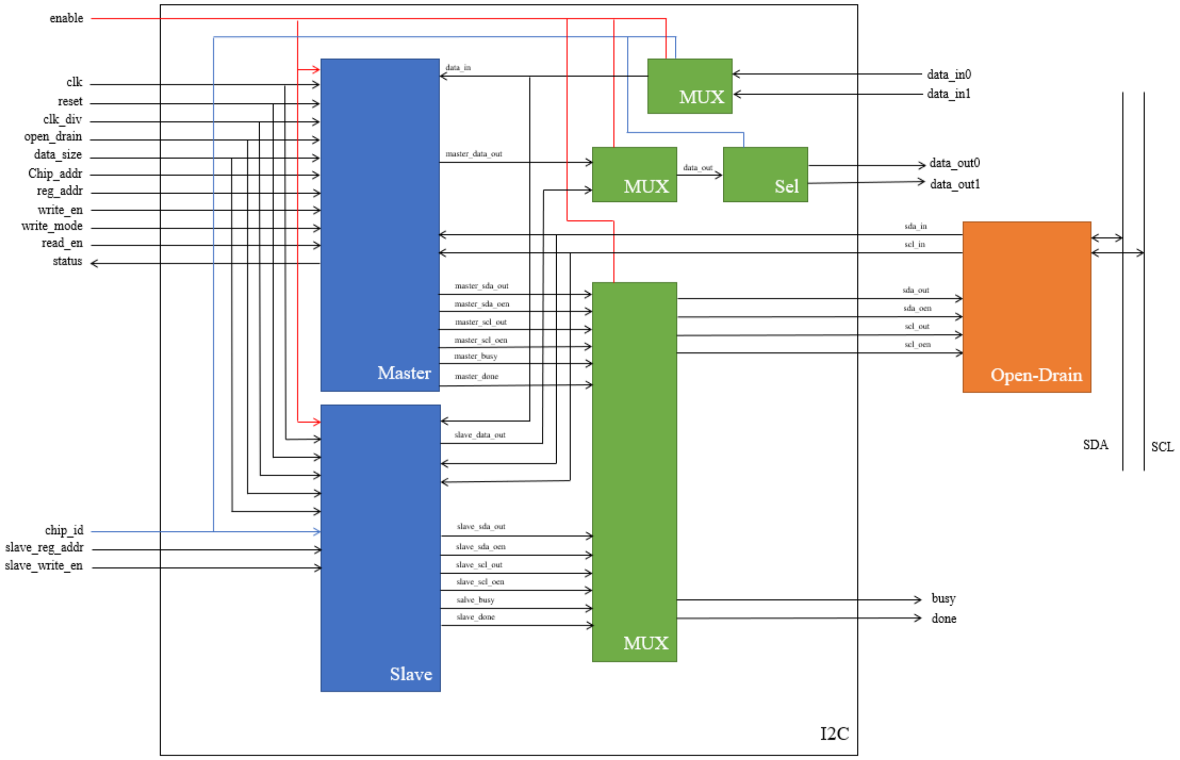


Figure 3.3: I^{2}C Architecture

3.1.4 Open-Drain Structure for Bidirectional Communication

Open_drain, consisting of a pull-up resistor and a pull-down transistor, is the transmission structure that supports multi-input coexistence and enables the interconnection of multiple devices on a single data line with the bidirectional data flow. When the output signal is high, the transistor can pull the line to low, which is usually grounded. Conversely, when the output signal is low, the pull-up resistor pulls the bus to a high voltage. As no device can output a high voltage on the line, the bus can never encounter the problem of communication signal collision. This enables the master and slave devises to communicate simultaneously on the same data line (SDA).

Fig.3.4 (a) depicts the Open-Drain structure. According to the I^{2}C protocol specification, this

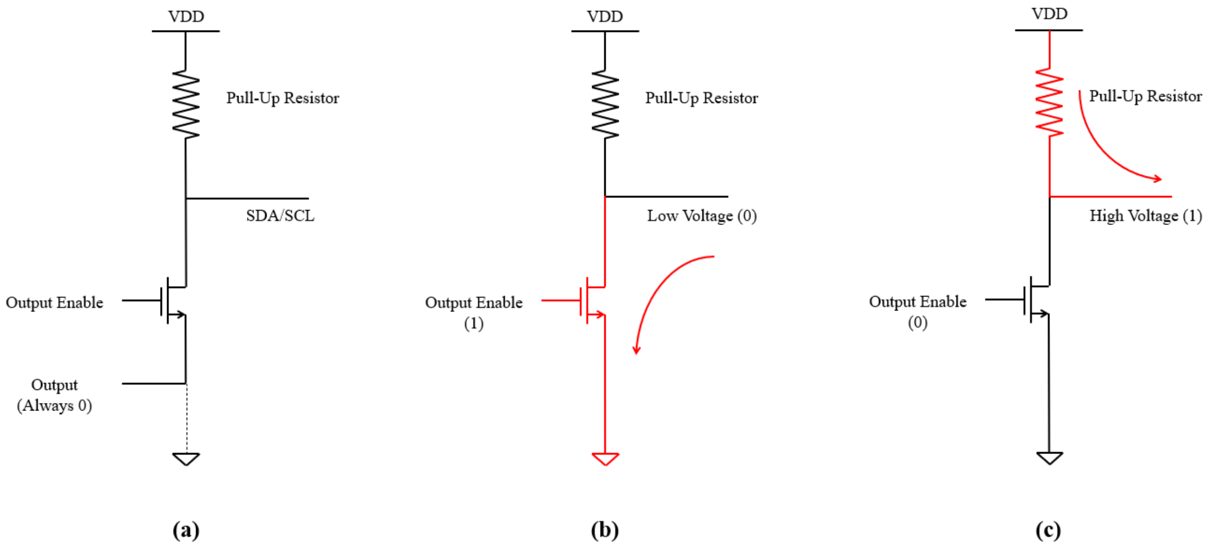
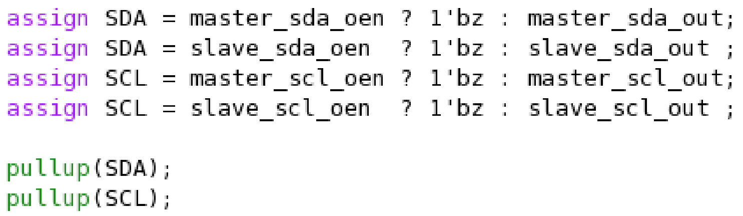


Figure 3.4: Open-Drain Structure

design has two input signals: ”output” (OUT) and ”output enable” (OEN). In the design and simulation presented in this article, the output is always 0, which means that the drain is grounded by default in the Open-Drain structure. Figures (b) and (c) illustrate the states of the Open-Drain circuit when the ”output enable” signal is 0 and 1. We use conditional statements in the testbench during the ModelSim simulation to describe the Open-Drain design since it does not include an analog circuit structure, as shown in Fig.3.5.



```
assign SDA = master_sda_oen ? 1'bz : master_sda_out;
assign SDA = slave_sda_oen  ? 1'bz : slave_sda_out ;
assign SCL = master_scl_oen ? 1'bz : master_scl_out;
assign SCL = slave_scl_oen  ? 1'bz : slave_scl_out ;

pullup(SDA);
pullup(SCL);
```

Figure 3.5: Open-Drain Verilog Code

3.1.5 SDA and SCL

In the I^{2}C protocol standard, Serial Data Bus (SDA) and Serial Clock Bus (SCL) are the only two data lines for data exchange, and there are some important rules to be aware of when using them for data transmission.

During data transmission, I^{2}C supports the use of different frequencies. Having sound transfer requirements in place is critical to ensuring data is correct. During data transmission, when SCL is high (1), the voltage on SDA is fixed, and the receiver will read the value on SDA. When SCL is low (0), SDA changes to the value stored in the transmitter's shift register, as illustrated in Fig.3.6.

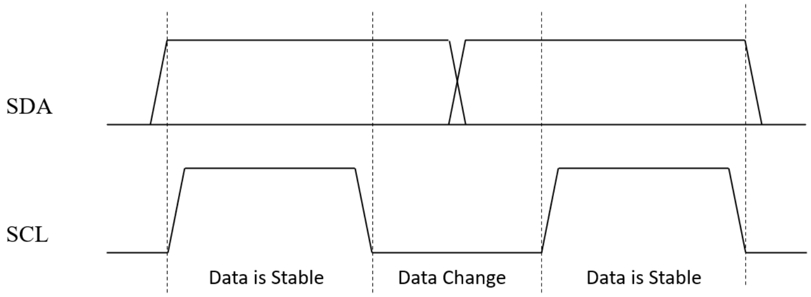


Figure 3.6: Data Transmission

Another important rule in the I^{2}C protocol is that the falling edge of SDA is used as the beginning of communication. Therefore, in our design, we ensure that SDA is pulled down from high (1) to low (0) when SCL is high (1) since SDA is high when the system is idle, as shown in Fig.3.7. At the same time, the change when SCL is high is used to distinguish it from the data transmission process. In a complex system containing multiple I^{2}Cs, the start signal (falling edge of SDA) wakes up all I^{2}C slaves, which then checks whether their chip ID matches the address received from SDA. If there is no match, the slave returns to the idle state. Otherwise, it sends a response signal.

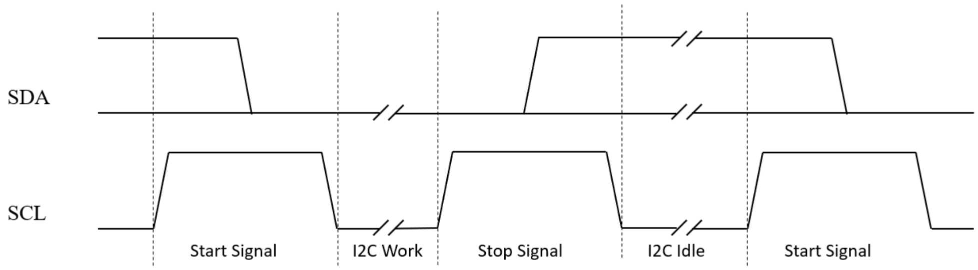


Figure 3.7: Start and Stop Signal

3.1.6 Master and Slave Finite state machine

Compared to SPI, the Finite State Machine (FSM) structure of I^{2}C is more complex due to its single data line(SDA), and the clock line (SCL) being generated only by the master to signal operation completion.

When the I^{2}C master performs read and write operations, it behaves differently. As shown in the flowchart in Fig.3.8, a write operation involves four frames transfer. Including a 7-bit target chip address and a 1-bit write operation signal, an 8-bit target register address, and two 8-bit data. On the other hand, a read operation requires five frames to complete. First, the master sends the 7-bit target chip address and 1-bit write operation signal, followed by the 8-bit target register address. After receiving the response from the slave, the master restarts and sends a 7-bit target chip address and a 1-bit read operation signal, and the slave then sends two 8-bit data.

The operation details of the slave become clear from the operation process of the master. Notably, the master's start is controlled by two enable signals, read_en and write_en, while the slave's start is triggered by the falling edge of the SDA data line, which is high when idle. Each exchange of information on the SDA data line is 9 bits, with the first 8 bits being messages and the last 1 bit being a response (ACK).

To better explain the operation of I^{2}C, we explain it from the signal changes on the SDA data bus. We don't need to focus on the signal on SCL that the master provides, as some details will be mentioned later.

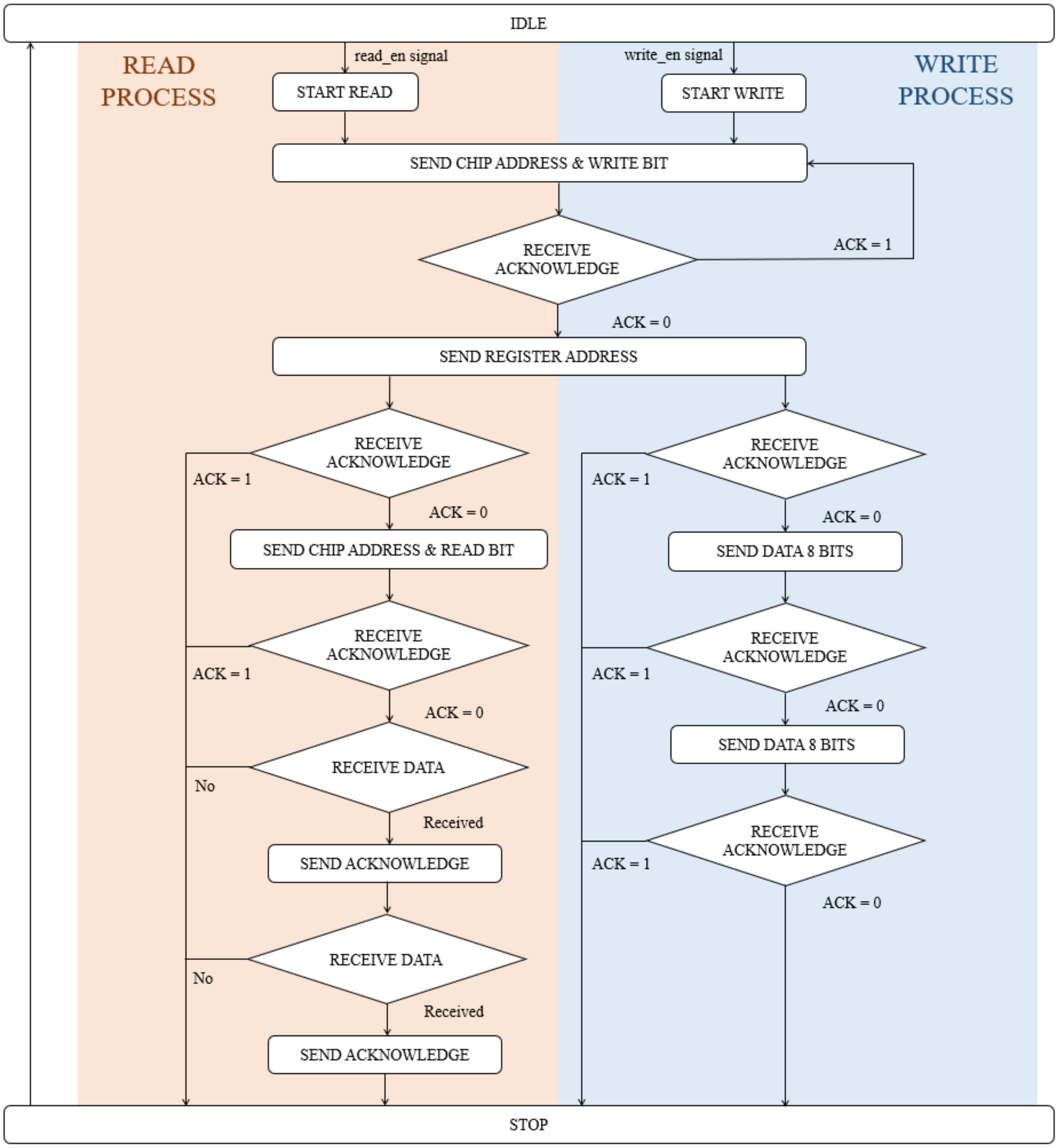


Figure 3.8: Flowchart of I^{2}C Master Operation

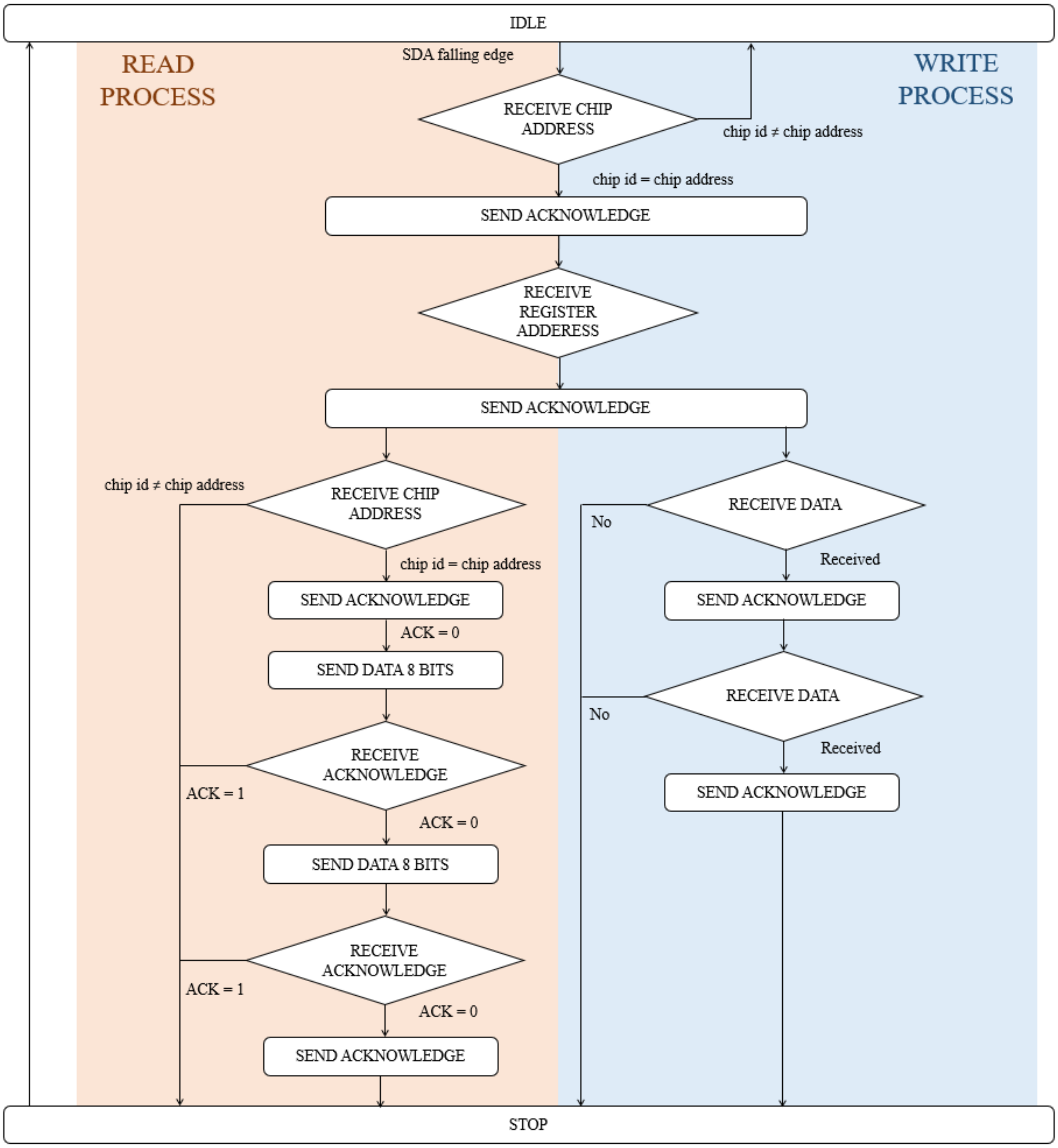


Figure 3.9: Flowchart of I^{2}C Slave Operation

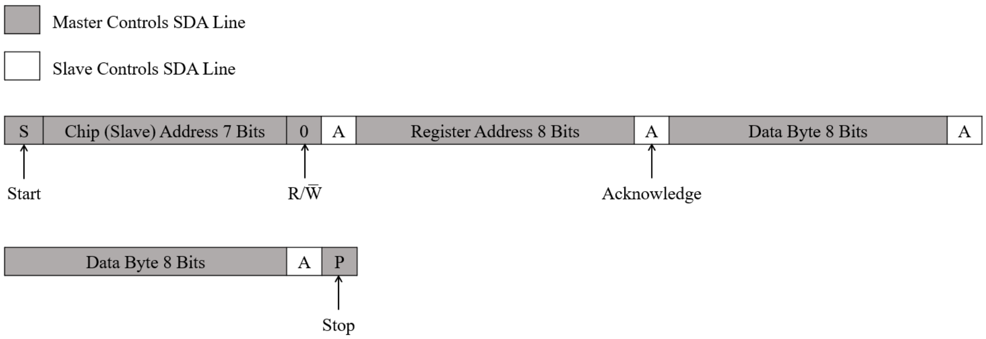


Figure 3.10: SDA Active When Write Operation

Fig.3.10 illustrates the activity of the SDA during a write operation. When the Start signal is sent, the master uses the SDA to send the target chip address and 'write' signal. If the slave chip does not respond, the master chip continues to send the chip address. When the slave chip receives the address from SDA that matches its chip id, it sends an acknowledge signal (low voltage, 0). The master then sends the target register address and the data, and the slave sends a 1 bit ACK signal when it receives each piece of data.

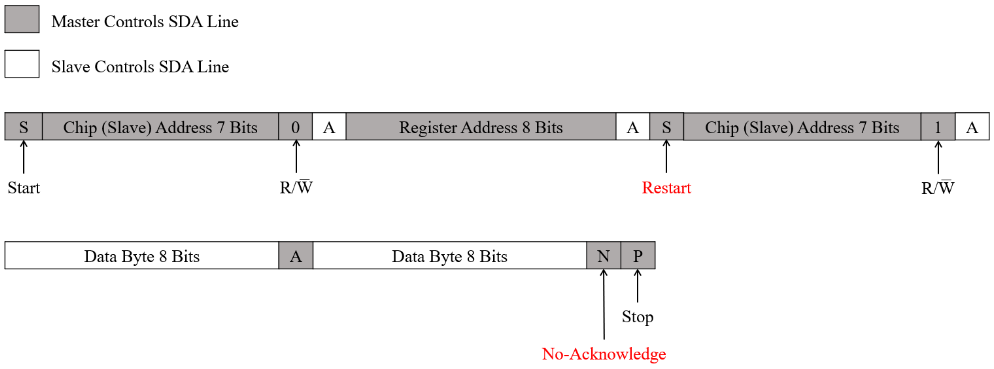


Figure 3.11: SDA Active When Write Operation

The process of I^{2}C's read operation is significantly different from that of the write operation, as shown in Fig.3.11. The articles about the I^{2}C protocol and some product manuals of sensors with I^{2}C components clearly define I^{2}C's read operation. The master sends the target chip address and 'write' signal, followed by the target register address. The master then enters the start_read state, shown in the SDA as the start signal is resent, and the SDA changes from a high to a low when the

SCL is high voltage. Later in the process, the slave will send data twice, and the master sends an ACK signal when the first completion is completed, then sends a NACK signal (high voltage, 1) when the second completion is completed. In the I^{2}C read operation, the master will only send a NACK signal after receiving the last byte of data when more information is transmitted.

3.1.7 I^{2}C Data Register

To ensure efficient data transfer in I^{2}C communication, it is essential to have a register file specifically designed for this purpose. Attempting to grab data from memory during each run can be impractical since it may take multiple clock cycles, which can slow down the process. Additionally, the I^{2}C slave cannot pause and wait for the arrival of data, further highlighting the importance of having a register file. Equipping the register file for I^{2}C allows the data to be readily available for transfer, reducing delays and improving overall system performance.

Given the limited number of registers available in the SPI-I^{2}C Protocol Converter chip, to test the I^{2}C read and write function at the specified position on the SPI-I^{2}C Protocol Converter chip's register, we have incorporated four 16-bit registers in the current design.

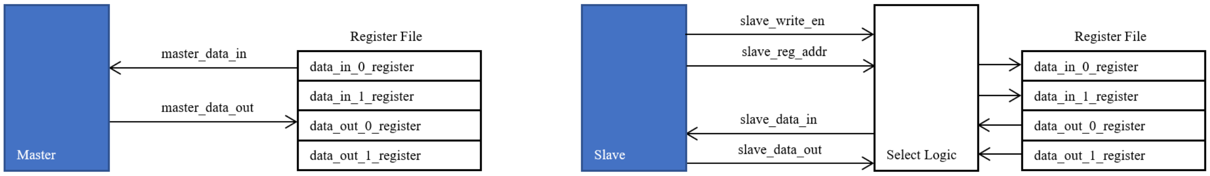


Figure 3.12: I^{2}C Data Register

3.2 Simulation and Result

Due to the unique nature of the I^{2}C module, it can be challenging to simulate and test the master and slave components separately. This is because both read and write operations require the transmission and reception of data on SDA by both the master and slave. Moreover, if the ACK signal is not received in a timely manner, the system may crash or terminate. To perform the simulation, we use the I^{2}C module depicted in Fig.3.3 and connect the SDA and SCL of the two I^{2}Cs using the wiring illustrated in Fig.3.2. In this setup, the first I^{2}C is configured as the master (enable = 1), and the second I^{2}C is set as the slave (enable = 0).

The I^{2}C write operation involves two times write processes and is depicted in Fig.3.13. In this simulation, I^{2}C_1 is set up as the master, operating at 100KHz, while I^{2}C_2 is configured as the slave, working at 25KHz. The clk_div value is set to 100, resulting in an SCL frequency of 1KHz. It's worth noting that the master and slave can operate at different frequencies. Still, the SCL frequency must be smaller than the input frequency of the I^{2}C. During the chip test, the I^{2}C operating frequency is usually at least three times that of the SCL. We increased the clk_div value from 10 to 100 to meet the requirements in this simulation. At begin of the simulation, we assign the target chip address (chip_addr) of the master and the chip address (chip_id) of the slave to 0001111 (0F). We then assign the target register address (reg_addr) of the two operations to 00 and 01, respectively, verifying that the I^{2}C slave can write to different registers. In the first operation, I^{2}C_1 sends the target register address 00 and 2-byte data 'A1A1' to I^{2}C_2. According to the address in slave_reg_addr, the data in data_out is stored in the corresponding register (data_out0) using the register pointer logic. Similarly, the second operation stores the data 'B2B2' in the register data_out1.

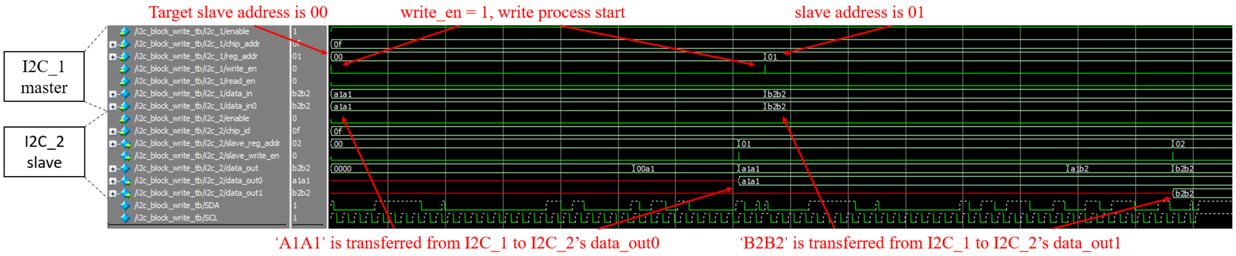

Target slave address is 00
write_en = 1, write process start
slave address is 01
I2C_1 master
I2C_2 slave
/i2c_block_write_tb/i2c_1/enable 1
/i2c_block_write_tb/i2c_1/chip_addr 0f
/i2c_block_write_tb/i2c_1/reg_addr 01
/i2c_block_write_tb/i2c_1/write_en 0
/i2c_block_write_tb/i2c_1/read_en 0
/i2c_block_write_tb/i2c_1/data_in b2b2
/i2c_block_write_tb/i2c_1/data_in0 b2b2
/i2c_block_write_tb/i2c_2/enable 0
/i2c_block_write_tb/i2c_2/chip_id 0f
/i2c_block_write_tb/i2c_2/slave_reg_addr 02
/i2c_block_write_tb/i2c_2/slave_write_en 0
/i2c_block_write_tb/i2c_2/data_out b2b2
/i2c_block_write_tb/i2c_2/data_out0 a1a1
/i2c_block_write_tb/i2c_2/data_out1 b2b2
/i2c_block_write_tb/SDA 1
/i2c_block_write_tb/SCL 1
0f
00
01
a1a1
b2b2
0f
00
01
02
0000
00a1
a1a1
a1b2
b2b2
a1a1
b2b2
'A1A1' is transferred from I2C_1 to I2C_2's data_out0
'B2B2' is transferred from I2C_1 to I2C_2's data_out1

Figure 3.13: I^{2}C Write Simulation

The changes in the Serial Data (SDA) and Serial Clock (SCL) lines during an I^{2}C write simulation are presented in Fig. 3.14. As shown in Fig. 3.10, the I^{2}C write operation consists of four segments, each with 9 bits. In this example, the master sends the chip address 0F, register address 00, and data

A1A1 to the slave. During transmission, when SCL is high, the value of SDA remains unchanged, and when SCL is low, the value of SDA changes. Sometimes, the value of SDA may change multiple times due to the control of SDA changing, which is approved because of the low voltage on SCL. Additionally, the start signal occurs when SDA changes from high to low while SCL is high, and the stop signal occurs when SDA changes from low to high while SCL is high.

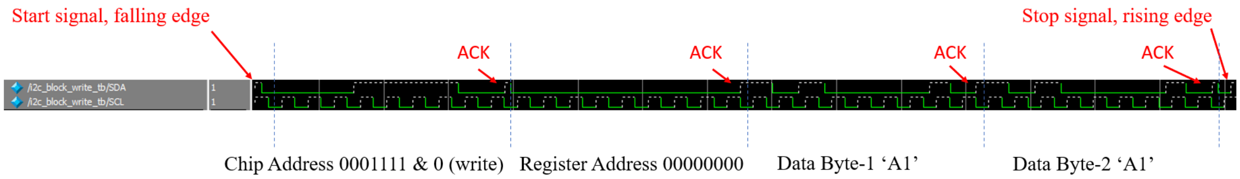

Start signal, falling edge
Stop signal, rising edge
ACK
ACK
ACK
ACK
/i2c_block_write_tb/SDA 1
/i2c_block_write_tb/SCL 1
Chip Address 0001111 & 0 (write)
Register Address 00000000
Data Byte-1 'A1'
Data Byte-2 'A1'

Figure 3.14: SDA and SCL in I^{2}C Write Simulation

Fig. 3.15 illustrates two I^{2}C read operations. In the first operation, the data from the data_in0 register of I^{2}C_2 is read and displayed in the data_out register of I^{2}C_1. In the second operation, the data from the data_in1 register of I^{2}C_2 is accessed. The simulation confirms that I^{2}C can read values from any register.

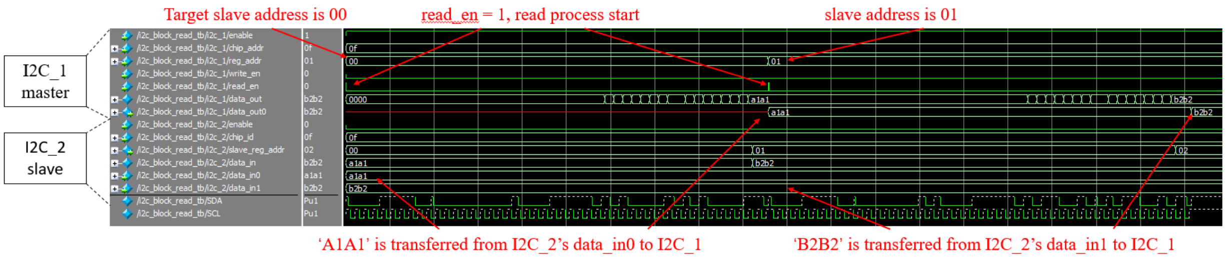
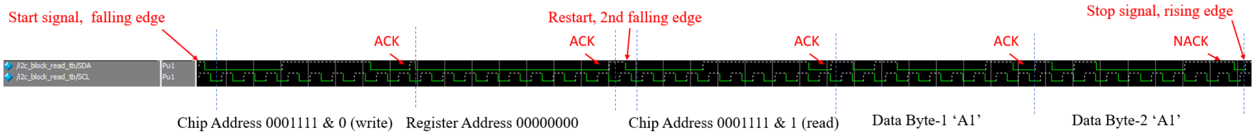

Target slave address is 00
read_en = 1, read process start
slave address is 01
I2C_1 master
I2C_2 slave
/i2c_block_read_tb/i2c_1/enable 1
/i2c_block_read_tb/i2c_1/chip_addr 0f
/i2c_block_read_tb/i2c_1/reg_addr 01
/i2c_block_read_tb/i2c_1/write_en 0
/i2c_block_read_tb/i2c_1/read_en 0
/i2c_block_read_tb/i2c_1/data_out b2b2
/i2c_block_read_tb/i2c_1/data_out0 b2b2
/i2c_block_read_tb/i2c_2/enable 0
/i2c_block_read_tb/i2c_2/chip_id 0f
/i2c_block_read_tb/i2c_2/slave_reg_addr 02
/i2c_block_read_tb/i2c_2/data_in b2b2
/i2c_block_read_tb/i2c_2/data_in0 a1a1
/i2c_block_read_tb/i2c_2/data_in1 b2b2
/i2c_block_read_tb/SDA Pu1
/i2c_block_read_tb/SCL Pu1
0f
00
01
0000
a1a1
a1a1
b2b2
b2b2
0f
00
01
02
b2b2
a1a1
a1a1
b2b2
'A1A1' is transferred from I2C_2's data_in0 to I2C_1
'B2B2' is transferred from I2C_2's data_in1 to I2C_1

Figure 3.15: I^{2}C Read Simulation

Fig. 3.16 presents the changes in the Serial Data (SDA) and Serial Clock (SCL) lines during the first read operation. Similar to Fig. 3.11, the SDA and SCL processes depicted in Fig. 3.16 are divided into five parts, each comprising 9 bits. However, unlike the write operation, the read operation includes two starts, indicated by two rising edges of SDA. Additionally, the master will send a NACK signal in the last response before the operation is completed.

In addition, we also simulated the scenario where the data length is limited to 1 byte. As previously



Start signal, falling edge

Restart, 2nd falling edge

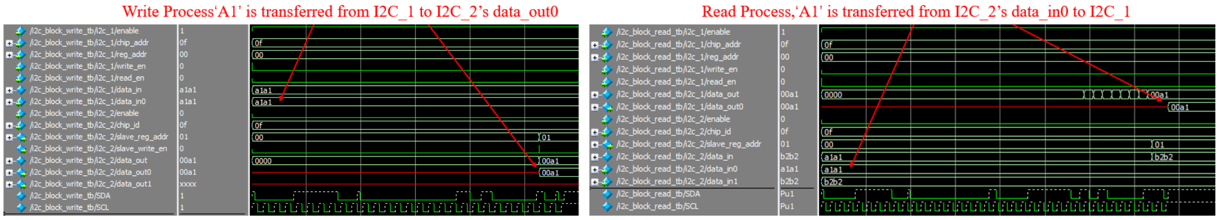
Stop signal, rising edge

```
/i2c_block_read_tb/SDA   Pu1
/i2c_block_read_tb/SCL   Pu1
```

ACK
ACK
ACK
ACK
NACK
Chip Address 0001111 & 0 (write)
Register Address 00000000
Chip Address 0001111 & 1 (read)
Data Byte-1 ‘A1’
Data Byte-2 ‘A1’

Figure 3.16: SDA and SCL in I^{2}C read Simulation

mentioned, I^{2}C supports both 1-byte and 2-byte transmission modes. And while most sensors on the Texas Instruments website use the 2-byte protocol, some only support the 1-byte I^{2}C protocol. Fig. 3.17 shows that the read operation comprises three parts, whereas the write operation consists of four parts. As we did not establish a separate testbench and I^{2}C register the device for the 1-byte transfer mode, the data is stored in the lower 8 bits of the 16-bit data register. In the process depicted in Fig. 3.17, both the read and write operations use 8-bit data ’A1’.



Write Process‘A1’ is transferred from I2C_1 to I2C_2’s data_out0

```
/i2c_block_write_tb/i2c_1/enable            1
/i2c_block_write_tb/i2c_1/chip_addr         0f
/i2c_block_write_tb/i2c_1/reg_addr          00
/i2c_block_write_tb/i2c_1/write_en          0
/i2c_block_write_tb/i2c_1/read_en           0
/i2c_block_write_tb/i2c_1/data_in           a1a1
/i2c_block_write_tb/i2c_1/data_in0          a1a1
/i2c_block_write_tb/i2c_2/enable            0
/i2c_block_write_tb/i2c_2/chip_id           0f
/i2c_block_write_tb/i2c_2/slave_reg_addr    01
/i2c_block_write_tb/i2c_2/slave_write_en    0
/i2c_block_write_tb/i2c_2/data_out          00a1
/i2c_block_write_tb/i2c_2/data_out0         00a1
/i2c_block_write_tb/i2c_2/data_out1         xxxx
/i2c_block_write_tb/SDA                     1
/i2c_block_write_tb/SCL                     1
```

Read Process,‘A1’ is transferred from I2C_2’s data_in0 to I2C_1

```
/i2c_block_read_tb/i2c_1/enable            1
/i2c_block_read_tb/i2c_1/chip_addr         0f
/i2c_block_read_tb/i2c_1/reg_addr          00
/i2c_block_read_tb/i2c_1/write_en          0
/i2c_block_read_tb/i2c_1/read_en           0
/i2c_block_read_tb/i2c_1/data_out          00a1
/i2c_block_read_tb/i2c_1/data_out0         00a1
/i2c_block_read_tb/i2c_2/enable            0
/i2c_block_read_tb/i2c_2/chip_id           0f
/i2c_block_read_tb/i2c_2/slave_reg_addr    01
/i2c_block_read_tb/i2c_2/data_in           b2b2
/i2c_block_read_tb/i2c_2/data_in0          a1a1
/i2c_block_read_tb/i2c_2/data_in1          b2b2
/i2c_block_read_tb/SDA                     Pu1
/i2c_block_read_tb/SCL                     Pu1
```

Figure 3.17: I^{2}C write and read Simulation with 1-Byte Operation

Chapter 4

Design and Implementation of SPI-I^{2}C Protocol Converter chip

Upon completing the simulation of the I^{2}C module, our next objective was to test the actual performance of I^{2}C by designing a chip with the SPI to I^{2}C protocol. The SPI-I^{2}C protocol converter chip includes an optimized SPI module and I^{2}C module with adjustable output voltage. The SPI-I^{2}C protocol converter has significant potential applications, not only because it facilitates protocol conversion from SPI to I^{2}C for connecting a microcontroller with an SPI interface and a sensor with only an I^{2}C interface but also because it enables better interconnection between two SoCs, which can be challenging to approach in SPI application scenarios.

This chapter will detail the process of implament the I^{2}C module, from RTL design to generating the GDSII file and final chip layout. Emulation is performed at various stages to ensure the proper functioning of the I^{2}C component. During the RTL stage, we use Modelsim simulation to verify the functionality of the Verilog file. After completing the synthesis and automated place and route, we use Cadence Specter simulation to verify the functionality of the device described by the netlist structure and ensure the success of the synthesis process. Moreover, we stitch the components together in Virtuoso to generate the complete chip.

4.1 RTL to GDSII flow for I^{2}C Module

According to the process in Fig.4.1, the schematic and layout of the I^{2}C module are generated through Synthesis and Automated Place and Route (APR).

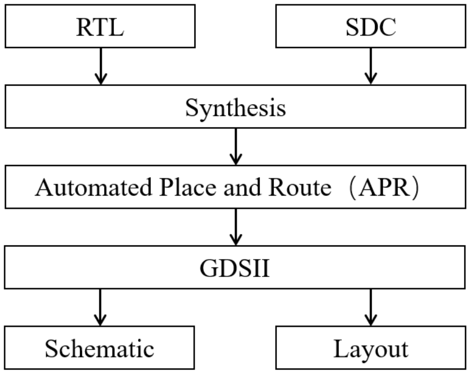
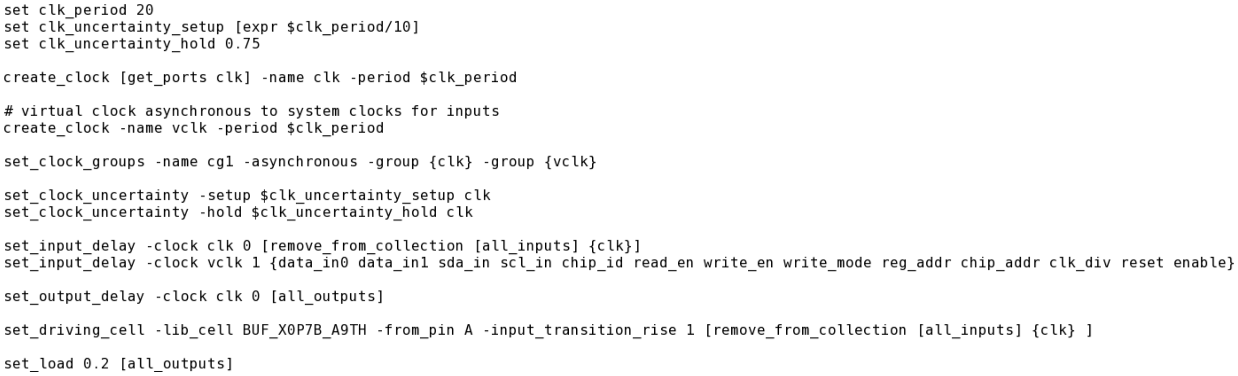


Figure 4.1: RTL to GDSII Flow

Synopsys Design Constraint (SDC) [31] is a standard format for constraining the design, supported by almost all Synthesis tools. Generally, timing, power, and area constraints of design are provided through the SDC file; this file has an extension .sdc.



```
set clk_period 20
set clk_uncertainty_setup [expr $clk_period/10]
set clk_uncertainty_hold 0.75

create_clock [get_ports clk] -name clk -period $clk_period

# virtual clock asynchronous to system clocks for inputs
create_clock -name vclk -period $clk_period

set_clock_groups -name cg1 -asynchronous -group {clk} -group {vclk}

set_clock_uncertainty -setup $clk_uncertainty_setup clk
set_clock_uncertainty -hold $clk_uncertainty_hold clk

set_input_delay -clock clk 0 [remove_from_collection [all_inputs] {clk}]
set_input_delay -clock vclk 1 {data_in0 data_in1 sda_in scl_in chip_id read_en write_en write_mode reg_addr chip_addr clk_div reset enable}

set_output_delay -clock clk 0 [all_outputs]

set_driving_cell -lib_cell BUF_X0P7B_A9TH -from_pin A -input_transition_rise 1 [remove_from_collection [all_inputs] {clk} ]

set_load 0.2 [all_outputs]
```

Figure 4.2: SDC Code

In the I^{2}C module, there's only one system clock CLK, but we also create a ”virtual clock” that's asynchronous to ”CLK.” By defining two clocks as asynchronous, Genus [32] and Innovus [33] will ignore and not clock any paths from one domain to another. We create a virtual asynchronous clock to associate the input signal ports that contain the I^{2}C block because they're asynchronous to CLK. If we skip this step, Genus/Innovus will waste resources optimizing timing paths for other registers in the input port designs. We don't need to specify any parameters for the virtual clock because it doesn't exist in the design, and it's asynchronous to the clock that does exist. We will determine the clock frequency during Synthesis and need to consider the trade-offs of the design. Adding input delay can lengthen the input-to-output and input-to-register data paths, so we set the input delay to 0 for all inputs. Finally, we specify a significant load on the output so that Genus/Innovus will enlarge the cell driving the output.

During Synthesis, we sweep the frequency from 10MHz to 100MHz to get the best results, as shown in Fig.4.3. When the frequency exceeds 30MHz, the generated area begins to increase, and when the frequency exceeds 50MHz, the frequency rises rapidly. We'll synthesize this design at 20 MHz since the application doesn't require the module to operate at a very high frequency. Moreover, the oscillator we provide for the I^{2}C module works below 100KHz. After Synthesis, we use Innovus software for automated place and route, where we need to focus on Floor Planning and IO Placement. Floor Planning defines the module's area, while IO Placement specifies the input and output port information, which provides the component layout for placement and arrangement in the final stage.

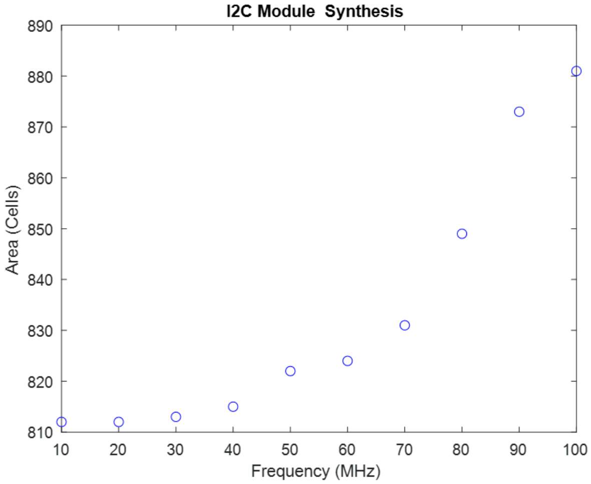


Figure 4.3: Frequency Sweep

After completing Synthesis and APR, Genus and Innovus automatically generate GDSII files and Verilog codes. We import these files into Cadence Virtuoso, where we can access the layout, schematic diagram, and symbols of the I^{2}C module.

4.2 I^{2}C Additional Circuit Design

In addition to using the I^{2}C module described by Verilog, we also need to add some circuits to ensure the regular operation of I^{2}C. When doing Modelsim simulation, we use some logic circuits instead of the Open-Drain structure. Now the Open-Drain structure will be drawn separately to meet the requirements of the I^{2}C protocol standard.

As shown in Fig.4.4, we designed an Open-Drain structure consisting of five pull-up resistors and two NMOS by calculating the transmission line's rising and falling edge times. In addition to the maximum size of the resistor, other components are controlled by a separate logic stored in the SPI register, which allows us to explore the impact of different pull-up capabilities on the SDA signal in the test. We have designed multiple sets of pull-up and pull-down structures for different connection situations. In addition, we explored the power consumption and performance of the I2C module through different structures.

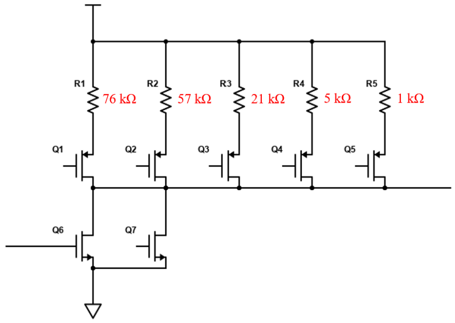


Figure 4.4: Open-Drain Structure

At the same time, to improve the compatibility between the I^{2}C module and commercial devices, we added a levell shifter to the circuit to control the voltage on the SDA line because the I^{2}C module works at 0.6V but almost Texas Instruments temperature Sensors work at the lowest is 1.02V. The level shifter can be adjusted to operate safely within the range of 0.6V-1.2V. Fig.4.5 shows the structure of the level shifter.

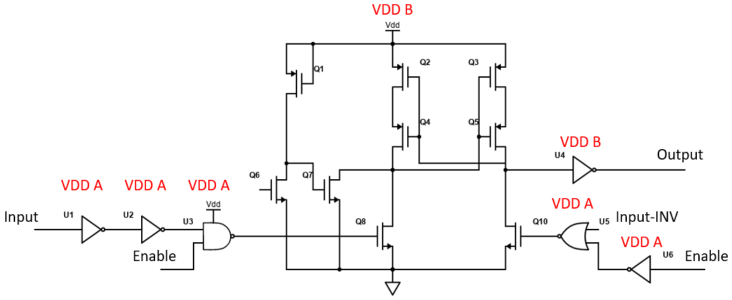


Figure 4.5: Level Shifter

4.3 Cadence Spactre Simulation

After importing the I^{2}C module into Cadence, perform the Spectre simulation to ensure the proper functioning of each component is crucial in completing the final design. The block diagram during simulation is shown in Fig. 4.6, which includes two complete I^{2}C modules.

The schematic during simulation is shown in Fig. 4.7. The blue part represents the voltage supplies of the I^{2}C and pull-up resistor, which correspond to the I^{2}C module and the Open-Drain circuit. Two I^{2}C clock signals, enable signals, and reset signals are also included. The red part shows two I^{2}C modules, and we added a buffer circuit before the SDA and SCL input ports. The green parts are two-level shifter circuits analyzed in detail in the previous section. The Open Drain circuit is marked in yellow, and this part provides the control for the Open-Drain circuit and determines the details of the pull-up and pull-down circuits. The black part represents two buses containing devices that simulate the PCB ports and the transmission line impedance. Finally, both read and write operations are shown, Fig. 4.8 and Fig. 4.9, and each part is consistent with the simulation results. Cadence simulation is an indispensable step in the design process. In the simulation, we consider the influence of various factors on the circuit, as described in the schematic. Through this

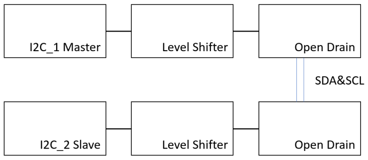


Figure 4.6: I^{2}C Simulation

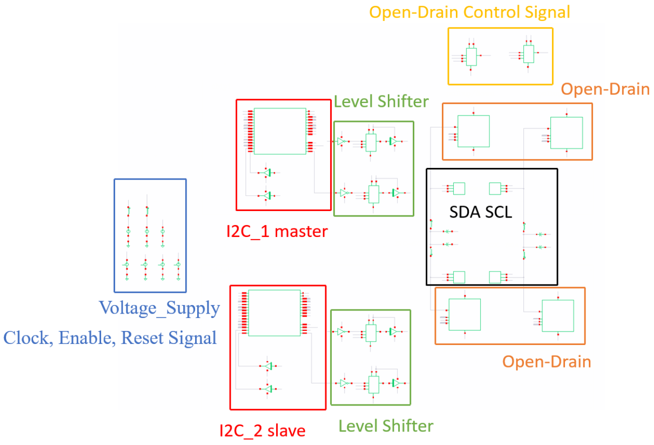


Figure 4.7: Schematic

simulation, we can estimate the actual operating conditions of the I2C module.

As described in the I^{2}C module state machine chapter and the Modelsim simulation results section, we analyzed the simulation results at this stage. In the write operation, the signal on the SDA transmission line is divided into four frames for analysis, and each frame is 9 bits. In Fig. 4.8, each frame signal is analyzed, which contains the target chip's address, the target register's address, and two bytes of data.

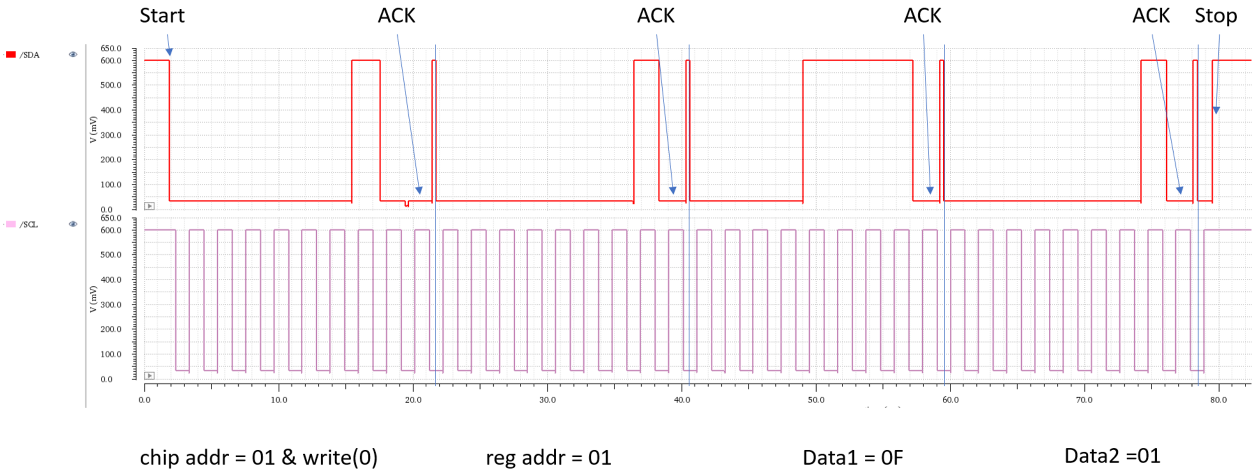


Figure 4.8: I^{2}C write Simulation

Similarly, in the I^{2}C write function test, the signal on the SDA line is divided into five frames. In addition, when the second frame ends, the master resends the start signal by sending the falling edge of the SDA line.

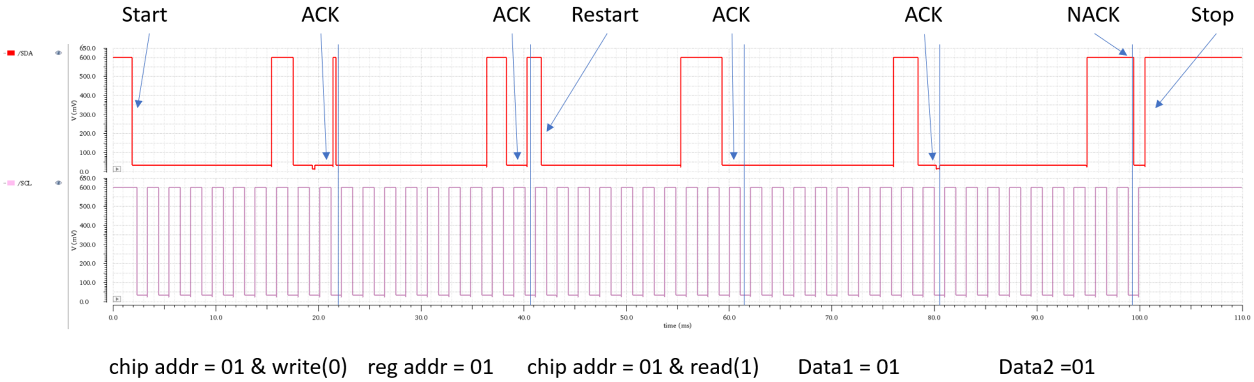


Figure 4.9: I^{2}C read Simulation

4.4 Schematic and Layout of Chip

After completing the simulation of the I^{2}C module, we integrated the SPI and I^{2}C parts to create the SPI-I^{2}C protocol converter chip, illustrated in Fig.4.12. Along with the SPI slave and I^{2}C modules, the chip includes an output voltage control module that allows selecting between 0.6V, 1.2V, and 1.8V and a temperature sensor for testing purposes. The SPI bidirectional components, which occupy a large area, are not directly related to the SPI-I^{2}C protocol converter and will not be discussed in detail in this article.

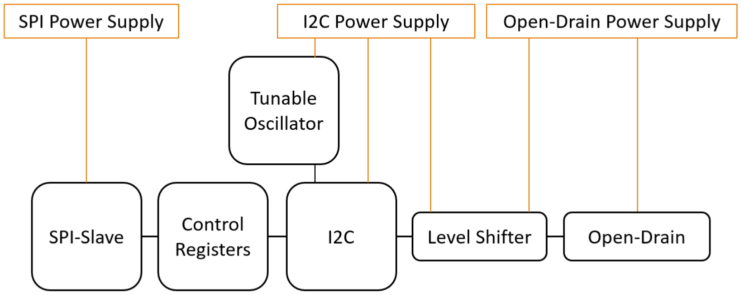
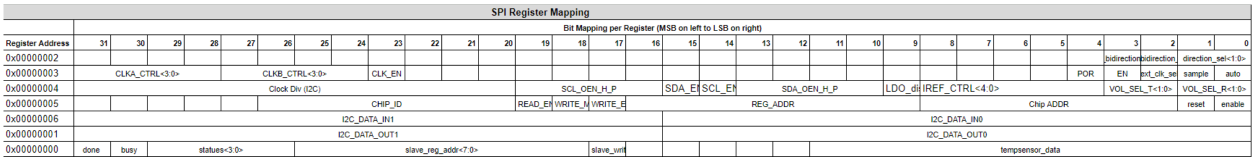
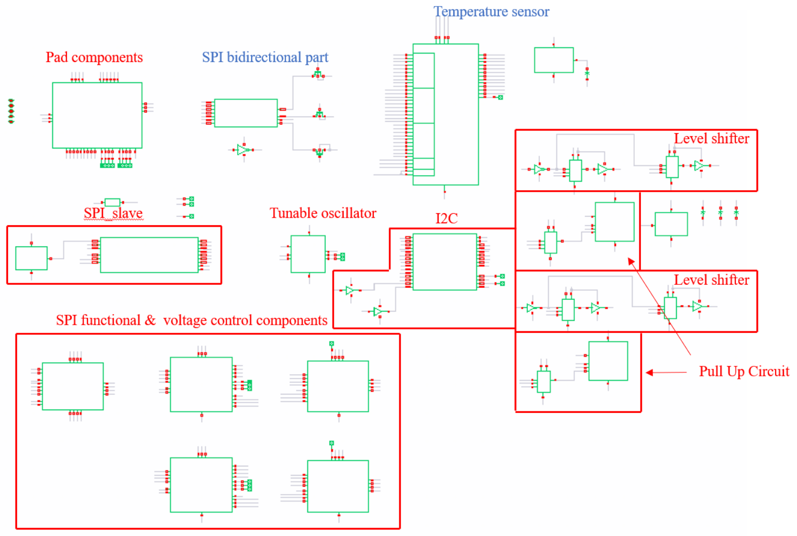


Figure 4.10: Chip Block Diagram



SPI Register Mapping																																
	Bit Mapping per Register (MSB on left to LSB on right)																															
Register Address	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00000002																													bidirection	bidirection_	direction_sel<1:0>	
0x00000003	CLKA_CTRL<3:0>				CLKB_CTRL<3:0>				CLK_EN																			POR	EN	ext_clk_se	sample	auto
0x00000004	Clock Div (I2C)												SCL_OEN_H_P				SDA_EI	SCL_EI	SDA_OEN_H_P				LDO_di	IREF_CTRL<4:0>					VOL_SEL_T<1:0>		VOL_SEL_R<1:0>	
0x00000005						CHIP_ID							READ_EI	WRITE_M	WRITE_E	REG_ADDR								Chip ADDR							reset	enable
0x00000006	I2C_DATA_IN1																I2C_DATA_IN0															
0x00000001	I2C_DATA_OUT1																I2C_DATA_OUT0															
0x00000000	done	busy	statues<3:0>				slave_reg_addr<7:0>								slave_writ						tempsensor_data											

Figure 4.11: SPI Register Table



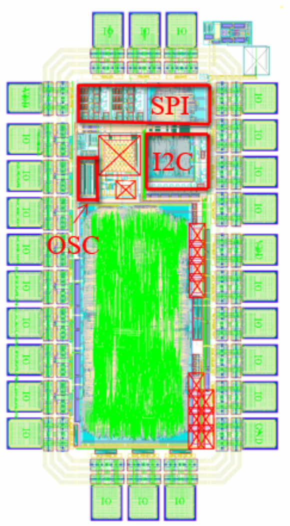


Figure 4.12: Schematic

Figure 4.13: Layout

Once SPI and I^{2}C are connected, we create a table to map the SPI register functions Fig.4.11 to the control port and registers of I^{2}C. This allows us to use SPI to control I^{2}C. Although this simulation involves too many components that can not be run, tests in the next chapter confirm that it works correctly.

The layout of the SPI and I^{2}C parts is marked in red in Fig.4.13. In the next chapter, we will test this part to ensure the proper functioning of controlling the I^{2}C through SPI. We will utilize a low-power oscillator to clock the I^{2}C. Besides, the red boxes with an ’X’ in the figure denote the capacitors in the TSMC library and do not imply any component errors or disabled features. Finally, we have completed the SPI-I^{2}C Protocol Converter chip design, and the next chapter will start chip testing.

Chapter 5

Chip Test Results and Analysis

This chapter will comprise a series of tests conducted to assess the functionality of the SPI-I^{2}C protocol converter chip, including function check, static power consumption measurement, dynamic power consumption estimation, and analysis. Additionally, the chapter will explore the implementation of a topology that utilizes three chips.

5.1 Build Testing Board

After successfully designing the PCB using PADS software, we soldered the necessary components onto the board. This included filter capacitors and linear and low-dropout (LDO) regulators, essential to ensuring a reliable power supply. ITWOC and PULLUP supply power to the I^{2}C module and Open-Drain pull-up resistors.

ESD_Board and ESD_I/O are crucial components for protecting electronic devices from damage due to high voltage. With each successive generation of IC development, achieving smaller form factors and thus smaller process nodes, these devices exhibit an increased vulnerability to damage or latch-up from electrostatic discharge (ESD) threats. Our ESD protection can ensure that the voltage exceeds the ESD_Board and ESD_I/O interface is grounded to protect the chip from

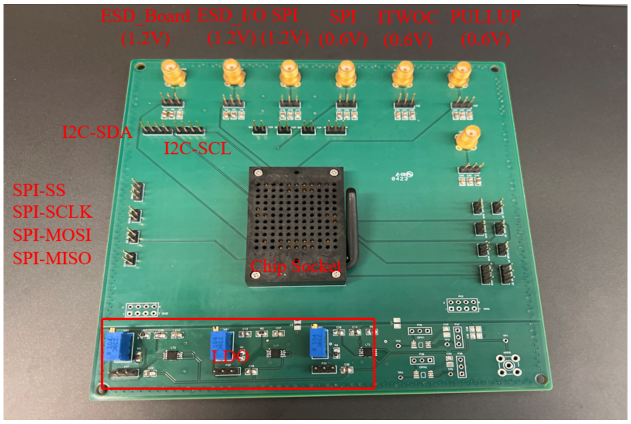


Figure 5.1: Testing Board

breakdown by excessive voltage. During testing, 1.2V is applied to ESD components, meaning any voltage exceeding this threshold in the power supply circuit will trigger automatic discharge through the ESD mechanism. This prevents the chip from being broken down and ensures optimal performance.

5.2 I^{2}C Basic Function Test

The block diagram used during the test is depicted in Fig.5.2. As previously stated, the I^{2}C test must be conducted on two chips since the Master and Slave cannot function independently and work together on one chip. The specifics of controlling SPI and I^{2}C using C code are illustrated in the left part of Fig.5.2. As two chips are utilized in the test, two SPI Master components are implemented on the SoC. The C code is compiled into machine code (.hex) by the compiler and subsequently uploaded to the SRAM in the SoC via the testbench. During runtime, the RISC core transmits relevant instructions to the SPI through the AXI4 and APB protocols in the SoC. Fig.5.3 displays a test code that effectively drives the SPI.

This code demonstrates how to use C code to control I^{2}C. The SPI1_write function writes values to the specified SPI register. The I^{2}C control information is described in the SPI register table created during the Chip top design and connected to I^{2}C in the previous chapter. I^{2}C can be configured and

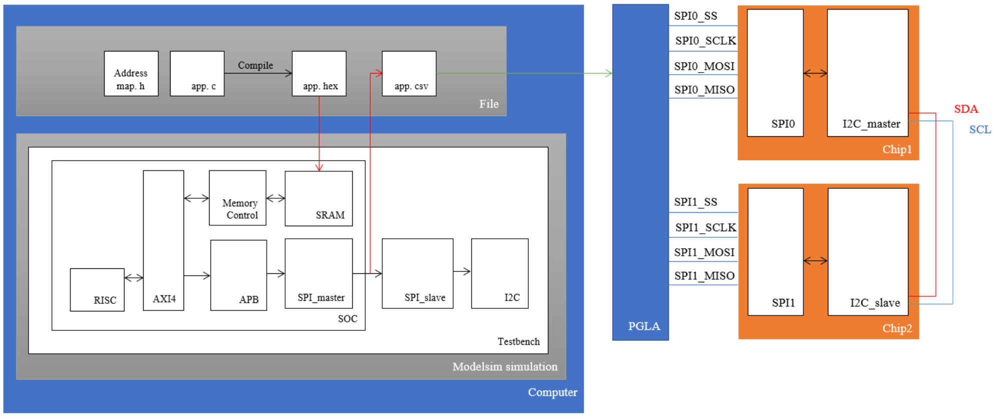
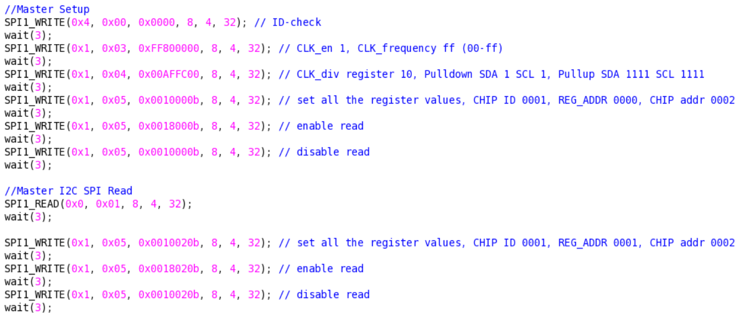


Figure 5.2: Block Diagram of Test

started by modifying the corresponding position's value. At the end of the configuration process, the read_en and write_en signals must finally be given. This is because the Pattern Generator Logic Analyzer (PGLA) has a slow transfer rate, and it's important to avoid starting the system before the configuration is completed.



```
//Master Setup
SPI1_WRITE(0x4, 0x00, 0x0000, 8, 4, 32); // ID-check
wait(3);
SPI1_WRITE(0x1, 0x03, 0xFF800000, 8, 4, 32); // CLK_en 1, CLK_frequency ff (00-ff)
wait(3);
SPI1_WRITE(0x1, 0x04, 0x00AFFC00, 8, 4, 32); // CLK_div register 10, Pulldown SDA 1 SCL 1, Pullup SDA 1111 SCL 1111
wait(3);
SPI1_WRITE(0x1, 0x05, 0x0010000b, 8, 4, 32); // set all the register values, CHIP ID 0001, REG_ADDR 0000, CHIP addr 0002
wait(3);
SPI1_WRITE(0x1, 0x05, 0x0018000b, 8, 4, 32); // enable read
wait(3);
SPI1_WRITE(0x1, 0x05, 0x0010000b, 8, 4, 32); // disable read
wait(3);

//Master I2C SPI Read
SPI1_READ(0x0, 0x01, 8, 4, 32);
wait(3);

SPI1_WRITE(0x1, 0x05, 0x0010020b, 8, 4, 32); // set all the register values, CHIP ID 0001, REG_ADDR 0001, CHIP addr 0002
wait(3);
SPI1_WRITE(0x1, 0x05, 0x0018020b, 8, 4, 32); // enable read
wait(3);
SPI1_WRITE(0x1, 0x05, 0x0010020b, 8, 4, 32); // disable read
wait(3);
```

Figure 5.3: C Code for I^{2}C Master Setup and Run

The most time-consuming aspect of gate-level simulation for digital circuits is generating input waveforms or stimuli. One efficient approach is to run the simulation initially as an RTL simulation and then export all the input waveforms to a .csv file. After a successful SoC simulation, we can use the testbench code to export the data of the specific segment we want to simulate. The Pattern Generator Logic Analyzer (PGLA) is a specialized tool for generating pulse signals. We use it to describe the data in the .csv file and generate accurate electrical signals. The PGLA is then connected to the test circuit board, enabling us to simulate communication between the SPI Master and the SPI slave on the chip and modify the value in the SPI register. Then we use an oscilloscope to observe the signals on the SDA and SCL lines to ensure the I^{2}C is working correctly.

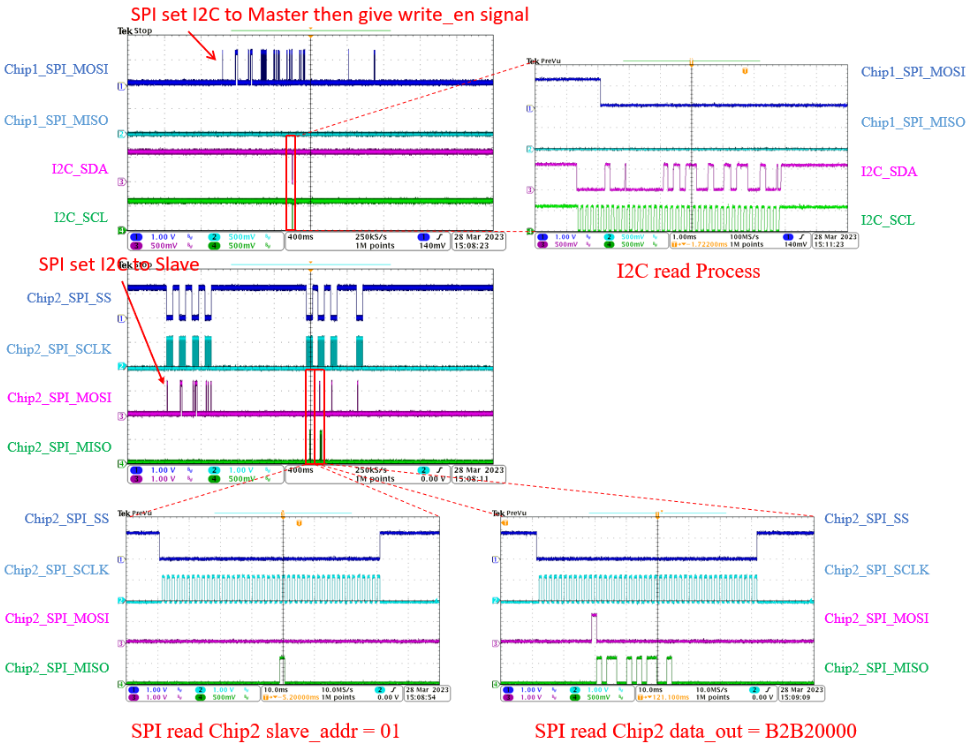


Figure 5.4: Write Process

The write operation results are shown in Fig.5.4. Once the I^{2}C operation is completed, we can query the value of the corresponding register through SPI to verify that the I^{2}C operation was performed correctly. We first set the address of our I^{2}C master to 01 and the address of the slave to 00. By comparing the C code, we want the master to change the register with a slave address of 01 to B2B2. Since the register with address 00 and the register with address 01 is placed in a 32-bit SPI register, and address 00 refers to the upper 16 bits in the SPI register, we can verify the test's success. Like the previous simulation results, the operation is divided into four segments of 9 bytes each. At the same time, the information transmitted is B2B2, and the transmitted address is 01, which is consistent with the previous observation of the register through SPI.

The reading test was conducted twice, displayed in Fig.5.5, with the master successfully reading the information on the two registers of the slave chip. To confirm the test's success, we cross-checked the C code, the register information read by SPI, and the oscilloscope's reading of the SDA line.

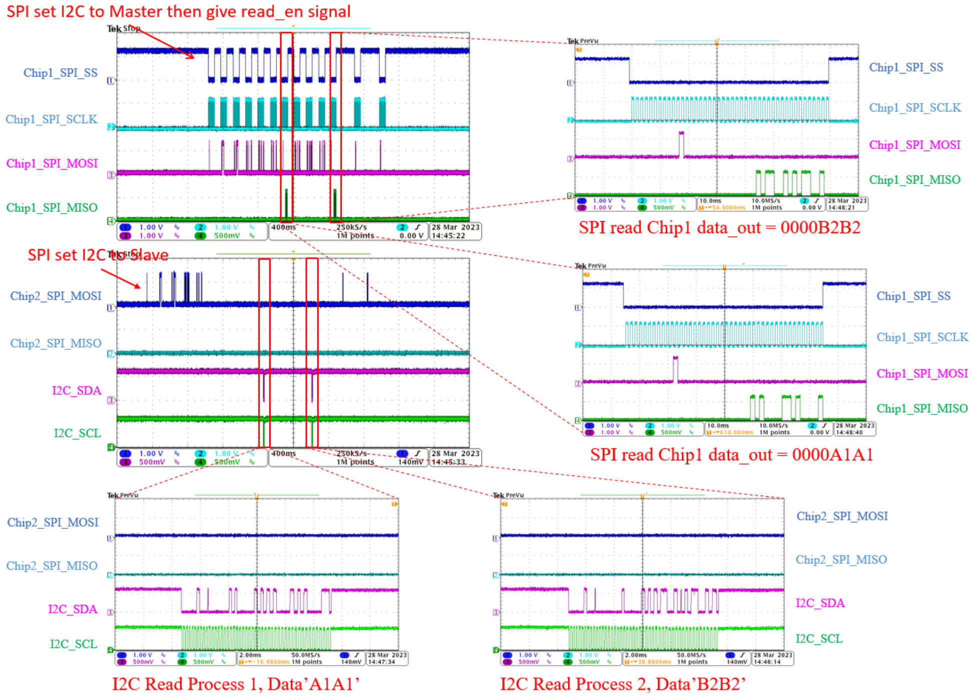


Figure 5.5: Read Process

In this part of the test, we checked the I2C ability to read and write to the specified register. The regular operation of the I2C module is verified by reading the value in the I2C register through SPI and comparing it with the expected value.

5.3 Power Consumption Measurement and Modeling

5.3.1 Static Power

We first tested the static power consumption of SPI and I^{2}C. At the same time, based on the simulation results, we drew a comparison chart of SPI and I^{2}C from 0.4V to 1.2V, by simulation and test. We found that the SPI-I^{2}C protocol converter can operate at a minimum of 0.4V, which is lower than expected. When we tested the static operating voltage of I^{2}C, we discovered that it only consumes 2.49nW at 0.4V and 4.19nW at 0.6V, but as much as 40.9nW at 1.2V. The figure shows that static power consumption increases exponentially as the supply voltage increases. We chose I^{2}C as the communication protocol because its static power consumption is lower than SPI's. However, we found that our simulation and test results had errors between 0.4V and 1.2V, likely due to the mutual influence of multiple power supply ports on the chip. The significant error occurred when SPI operated at 1.2V, at about 9.2%.

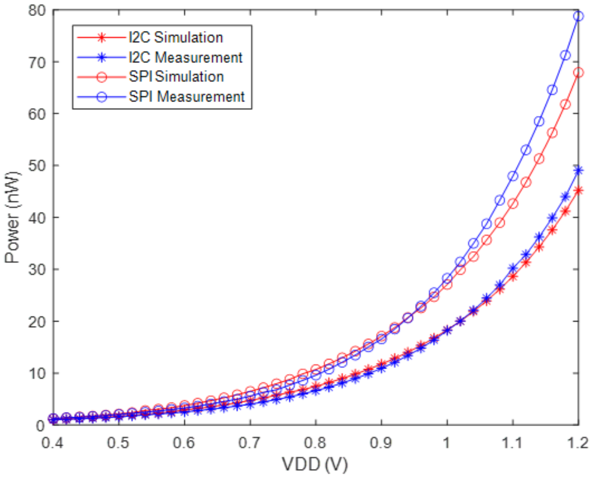


Figure 5.6: Dynamic power

5.3.2 Active Power

Our investigation revealed that the power consumption during the operation of the I^{2}C component is primarily due to the loss resulting from the short circuit of the pull-up resistor in the Open-Drain design. In the previous chapter, we discussed the design of the pull-up resistor, which involved creating various resistance values and developing a backup solution for the pull-down MOS design. Fig.5.7 illustrates that using only one pull-down MOS is insufficient to handle the pull-down capability of the circuit, resulting in abnormal operation of the I^{2}C module. Furthermore, as the resistance value of the pull-up resistor decreases, the waveforms on SDA and SCL become more standardized, but the power consumption increases. Ultimately, we selected the scheme indicated by the red arrow and tested its working power consumption.

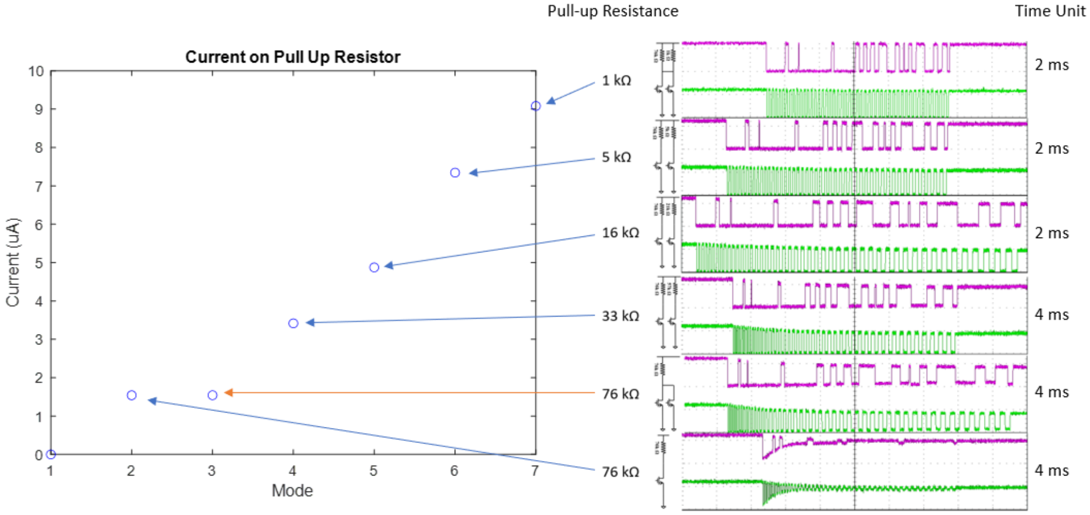


Figure 5.7: Current on Pull Up Resister

We conducted a simulation with a duration of 10mS to analyze the power consumption of the I^{2}C module during a read operation. Our findings indicate that the average power consumption of the I^{2}C module during this operation is 30.4nW, while the pull-up resistor consumes 195nW. This suggests that the dynamic power consumption of the I^{2}C module is significantly lower than the short-circuit power consumption of the pull-up resistor. To estimate the overall power consumption of the I^{2}C module in its working state, we considered both the dynamic and static power consumption observed in our simulation. Additionally, we tested the power consumption of the I^{2}C module using the SPI-I^{2}C chip. We conducted the test at different frequencies: once per minute, twice per minute, and four times per minute. We observed some deviation in the test results, which can be attributed to the difference between the actual operating frequency of the I^{2}C module and our estimate. By averaging multiple measurements of the three operating states, we estimated the overall power consumption of the I2C block operation based on the duty cycle.

The current in Fig.5.7 results from the Cadence simulation. At the same time, the power consump-

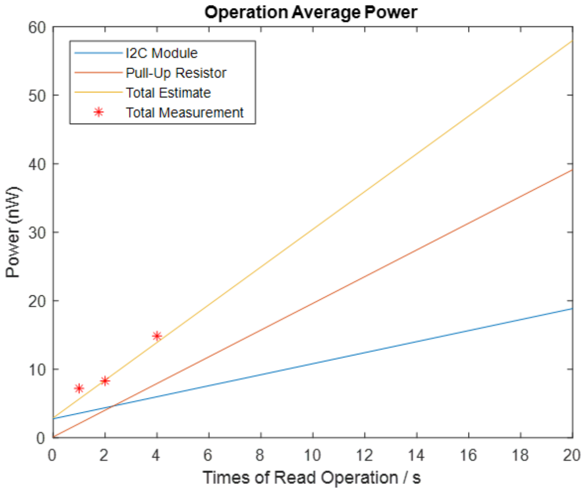


Figure 5.8: Active Power

tion measured in Fig.5.8 is the measurement result of the actual chip. There is a gap between the two because there is a certain error in the measurement process, and the simplest model is used in the simulation in order to use the gap between the current and the performance. In an actual chip, the magnitude of the current cannot be directly measured, so we cannot obtain the magnitude of the current flowing through the resistor.

5.4 System Topology Test

After completing the basic test of the I^{2}C module, we conduct a systematic topology test on the SPI-I^{2}C protocol converter chip. Since there are only two SPI masters in the current test platform, we used some optimized functions during the test. In the SPI-I^{2}C chip, SPI differs from the traditional design and can pass the MOSI and MISO signals to the next level through the switch, which means that we can connect the SPI in series and identify the communication target through the chip ID, which is similar to how the I^{2}C module completes chip ID identification, as Fig.5.9.

Through the connection of the three chips, we can set their SPI module and I2C module in turn and then read the registers in the chip through the SPI Master to realize the detection of the I2C

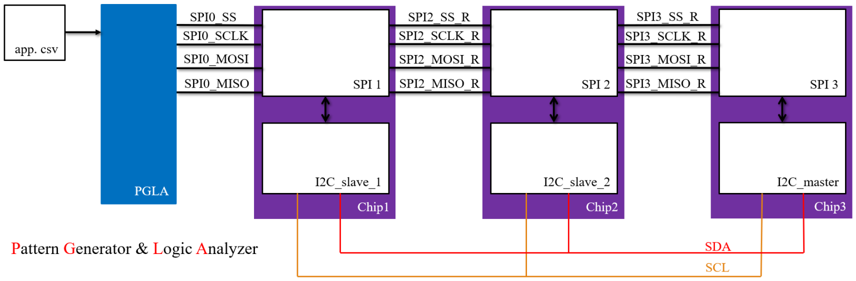


Figure 5.9: 3 Chip I^{2}C Write Test

operation. In Fig.5.10, we did two write operations using the I^{2}C master. The first time, we write ’A1A1’ to slave1, whose chip address is 02, and the target register address is 01. And the second time, we write ’B2B2’ to slave2, whose chip address is 03, and the target register address is 00. In the following verification process, since we can only perform operations on the SPI sequentially, we sequentially read ”A1A1” stored in Chip2 (Slave1) and ”B2B2” stored in Chip3 (Slave2).

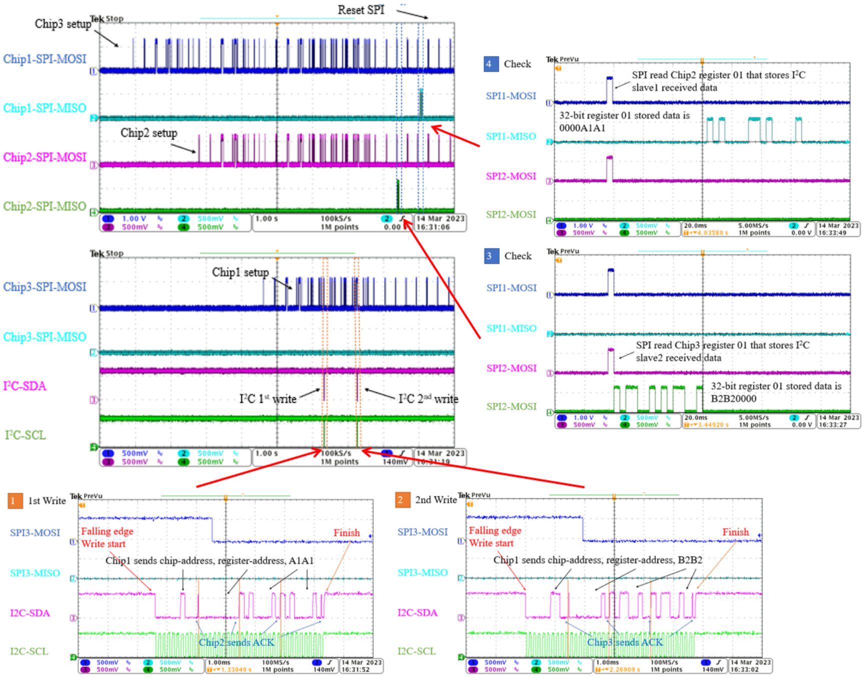


Figure 5.10: 3 Chip I^{2}C Write Test

When performing a read operation, we first read the ’B2B2’ stored in Chip2 Register 00 and then use SPI to read the data from the I^{2}C master for verification. Then read ’C3C3’ in Chip3 Register

01. The data out of the I^{2}C master is always connected to the last 16 bits of the 32-bit register, consistent with what we described in the I^{2}C data register.

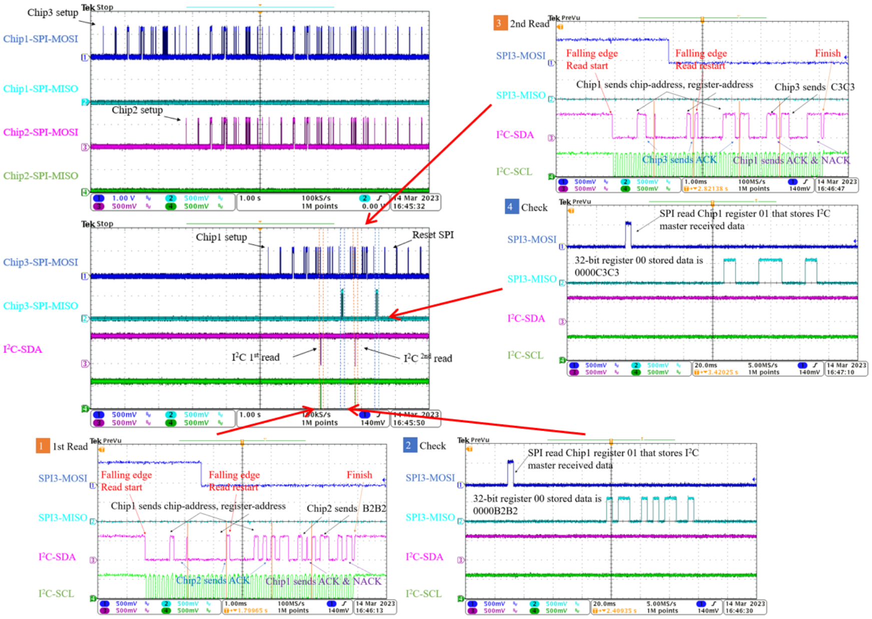


Figure 5.11: 3 Chip I^{2}C Read Test

Our latest testing round has proven the successful functionality of the I^{2}C module, as we were able to read from and write to various registers on different chips. This success highlights the I^{2}C's ability to interconnect chips effectively within a distributed system, ultimately contributing to its efficiency. We also plan to test the four-chip interconnection to prove that the protocol conversion chip can do the job in the mesh network structure. The test results are the same as those shown in this chapter, and the structure is complex, so it is not shown.

Chapter 6

Integrate the I^{2}C Module into the SoC

Now that the SPI-I^{2}C Protocol Converter Chip testing is complete, the next step is integrating the I^{2}C block into the System-on-Chip (SoC). The current SoC features multiple SPI masters, a necessary compromise to facilitate communication with several chips. Adding the I^{2}C protocol block to the SoC can reduce the number of required SPI protocol blocks, resulting in lower power consumption.

Integrating I^{2}C into an existing SoC can be challenging, as it requires a thorough understanding of the system architecture and the internal protocols of the SoC. This chapter will discuss these aspects in detail and explore the process of adding components to a complete system.

6.1 SoC Architecture

The currently established SOC includes RISC-V Core, Boot ROM, memory management unit, SRAM, and system interconnects. The core named BottleRocket [34] is a 32-bit, RISC-V microcontroller-class processor core built as a customized microarchitecture from the Free Chips Project Rocket core components. The SoC uses two different interconnect protocols, ARM Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) [35] [36] and ARM

AMBA Advanced Peripheral Bus (APB) [37]. AMBA AXI4-Lite has a more straightforward interface than the complete AXI4 and is suitable for connecting the microcontroller with boot ROM, SRAM memory controller, and APB bridge in this project. APB bridge [38]. The APB bridge connects peripheral components, such as General-Purpose Input/Output (GPIO), SPI, and I^{2}C. By setting addresses for different components, the CPU can write to the control registers of different components to achieve the purpose of register control.

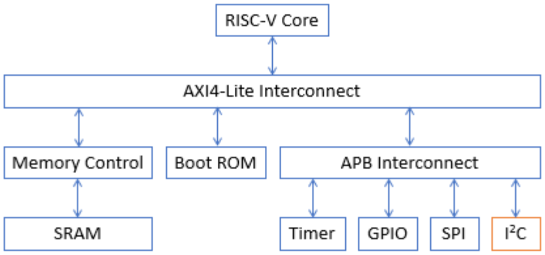


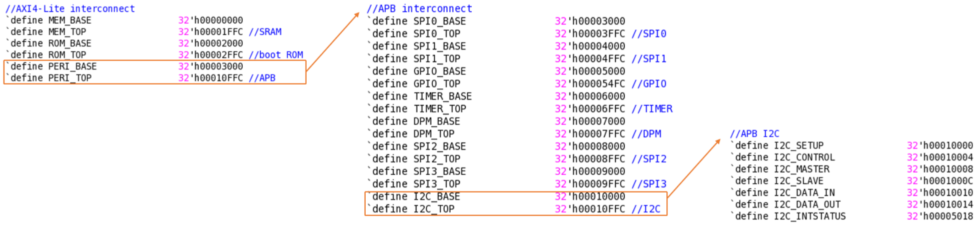
Figure 6.1: SoC Architecture

6.2 Connect I^{2}C module with APB bus

6.2.1 Update Address Map

In the address map, all components included, such as the corresponding addresses of SRAM, Boot ROM, and APB interconnect, are defined first because this part is the essential component requirement of SOC, which is prepared for AXI4-Lite interconnect identification. AXI4-Lite interconnect transfers instructions of different addresses to the corresponding modules. The APB interconnect works similarly, distributing instructions to additional peripherals.

As shown in Fig.6.2, in the AXI4 interface, the address bits covered by the APB interface and I^{2}C components present an inclusion relationship. Among the address bits covered by I^{2}C, six sets of registers are enabled and have corresponding addresses corresponding to them. In addition, there is a single address for implementing the interrupt mechanism. Because the virtual space allocated



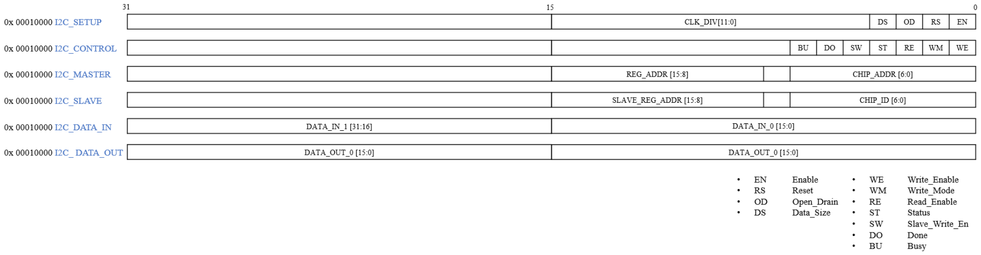
```
//AXI4-Lite interconnect
`define MEM_BASE                32'h00000000
`define MEM_TOP                 32'h00001FFC //SRAM
`define ROM_BASE                32'h00002000
`define ROM_TOP                 32'h00002FFC //boot ROM
`define PERI_BASE               32'h00003000
`define PERI_TOP                32'h00010FFC //APB
```

```
//APB interconnect
`define SPI0_BASE               32'h00003000
`define SPI0_TOP                32'h00003FFC //SPI0
`define SPI1_BASE               32'h00004000
`define SPI1_TOP                32'h00004FFC //SPI1
`define GPIO_BASE               32'h00005000
`define GPIO_TOP                32'h000054FC //GPIO
`define TIMER_BASE              32'h00006000
`define TIMER_TOP               32'h00006FFC //TIMER
`define DPM_BASE                32'h00007000
`define DPM_TOP                 32'h00007FFC //DPM
`define SPI2_BASE               32'h00008000
`define SPI2_TOP                32'h00008FFC //SPI2
`define SPI3_BASE               32'h00009000
`define SPI3_TOP                32'h00009FFC //SPI3
`define I2C_BASE                32'h00010000
`define I2C_TOP                 32'h00010FFC //I2C
```

```
//APB I2C
`define I2C_SETUP               32'h00010000
`define I2C_CONTROL             32'h00010004
`define I2C_MASTER              32'h00010008
`define I2C_SLAVE               32'h0001000C
`define I2C_DATA_IN             32'h00010010
`define I2C_DATA_OUT            32'h00010014
`define I2C_INTSTATUS           32'h00005018
```

Figure 6.2: Address Map

to I^{2}C is abundant, it is easy to find and supplement corresponding functions.



	31	15							0
0x 00010000 I2C_SETUP		CLK_DIV[11:0]				DS	OD	RS	EN
0x 00010000 I2C_CONTROL			BU	DO	SW	ST	RE	WM	WE
0x 00010000 I2C_MASTER		REG_ADDR [15:8]		CHIP_ADDR [6:0]					
0x 00010000 I2C_SLAVE		SLAVE_REG_ADDR [15:8]		CHIP_ID [6:0]					
0x 00010000 I2C_DATA_IN	DATA_IN_1 [31:16]	DATA_IN_0 [15:0]							
0x 00010000 I2C_ DATA_OUT	DATA_OUT_0 [15:0]	DATA_OUT_0 [15:0]							

• EN	Enable	• WE	Write_Enable
• RS	Reset	• WM	Write_Mode
• OD	Open_Drain	• RE	Read_Enable
• DS	Data_Size	• ST	Status
		• SW	Slave_Write_En
		• DO	Done
		• BU	Busy

Figure 6.3: I^{2}C Control and Data Register Map

Then, I put all the signals related to I^{2}C in the planned address area, as shown in Fig.6.3. Essential parameters are stored in I^{2}C_SETUP to determine the working condition of I^{2}C, and each parameter is introduced in detail when introducing the I^{2}C port. I^{2}C_CONTROL contains the startup and status signals when I^{2}C is running. Monitoring this register can effectively understand the running status of I^{2}C. The last two registers are connected to the data input and output of I^{2}C. They store two pieces of data, each of which is 2 bytes.

6.2.2 Build APB to I^{2}C Logic

The APB_I^{2}C file contains all the logic from the APB interface to the I^{2}C control registers. According to the structure in Fig.6.4, the output of the APB slave is matched to the corresponding control register. At the same time, the internal logic of APB also needs to be modified because new functions are expanded.

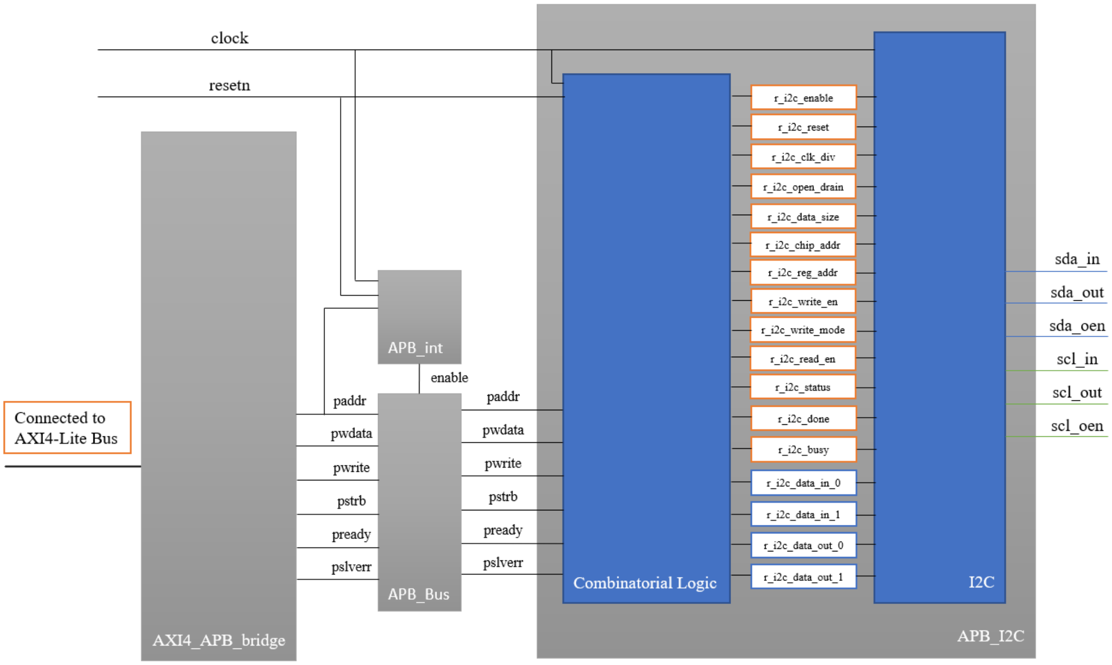
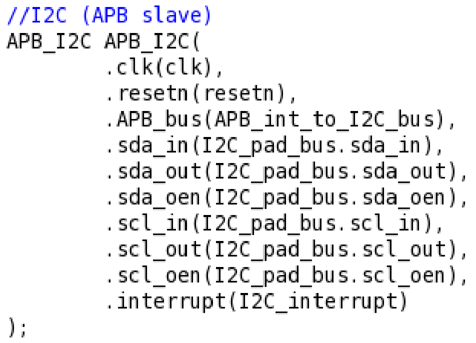


Figure 6.4: AXI4-Lite to APB to I^2C Structure

After the information is delivered to APB, according to the target address stored in APB_paddr, APB_int will transmit the APB data to the corresponding APB_slave. In APB_I^2C, the output of APB_slave will store the message in the corresponding register.

In the Digital_Top file, put the APB_I^2C module, we can see that the control signal terminal of the module and the APB interface are connected through APB_int_to_I^2C_Bus. On the other hand, the I/O port is connected to the SOC pad via I^2C_Pad_Bus. In addition, a dedicated interrupt signal port is provided for I^2C to use when it is written into data. I^2C_interrupt occurs when the slave_write_en signal is generated, and its function is to wake up the core to ensure that the written data is stored in the correct position.



```
//I2C (APB slave)
APB_I2C APB_I2C(
        .clk(clk),
        .resetn(resetn),
        .APB_bus(APB_int_to_I2C_bus),
        .sda_in(I2C_pad_bus.sda_in),
        .sda_out(I2C_pad_bus.sda_out),
        .sda_oen(I2C_pad_bus.sda_oen),
        .scl_in(I2C_pad_bus.scl_in),
        .scl_out(I2C_pad_bus.scl_out),
        .scl_oen(I2C_pad_bus.scl_oen),
        .interrupt(I2C_interrupt)
);
```

Figure 6.5: Connection of APB_I^{2}C

6.3 Modelsim Simulation

Compared to the simulation of the I^{2}C module, the difficulty of completing the system simulation in Modelsim is very serious, even if the SoC is complete. Designing the simulation environment directly in the testbench for system testing is not feasible. Fortunately, this complete system can run C code, so we can design corresponding code to control SoC and finally achieve the purpose of controlling I^{2}C.

6.3.1 Structure of SoC and Testbench

A complete set of the simulation process is used to simulate the digital circuit system. According to the description of the testbench in Fig.6.6, from the C code to the signal generation on the I^{2}C data line, we successfully put I^{2}C into the system.

To show the process of SOC simulation implementation more clearly, the whole process of realizing SOC Modelsim simulation is described with simple C code. First, compile the code ”app.c” and the address mapping table to save the machine code in hexadecimal. Import the machine code into SRAM through the statement in Testbench. RISC modifies the registers that control I^{2}C located in the APB_I^{2}C module by running instructions stored in SRAM. The interface of the AXI4 protocol is complex, directly connected to RISC, and can respond quickly. As shown in Fig.6.4,

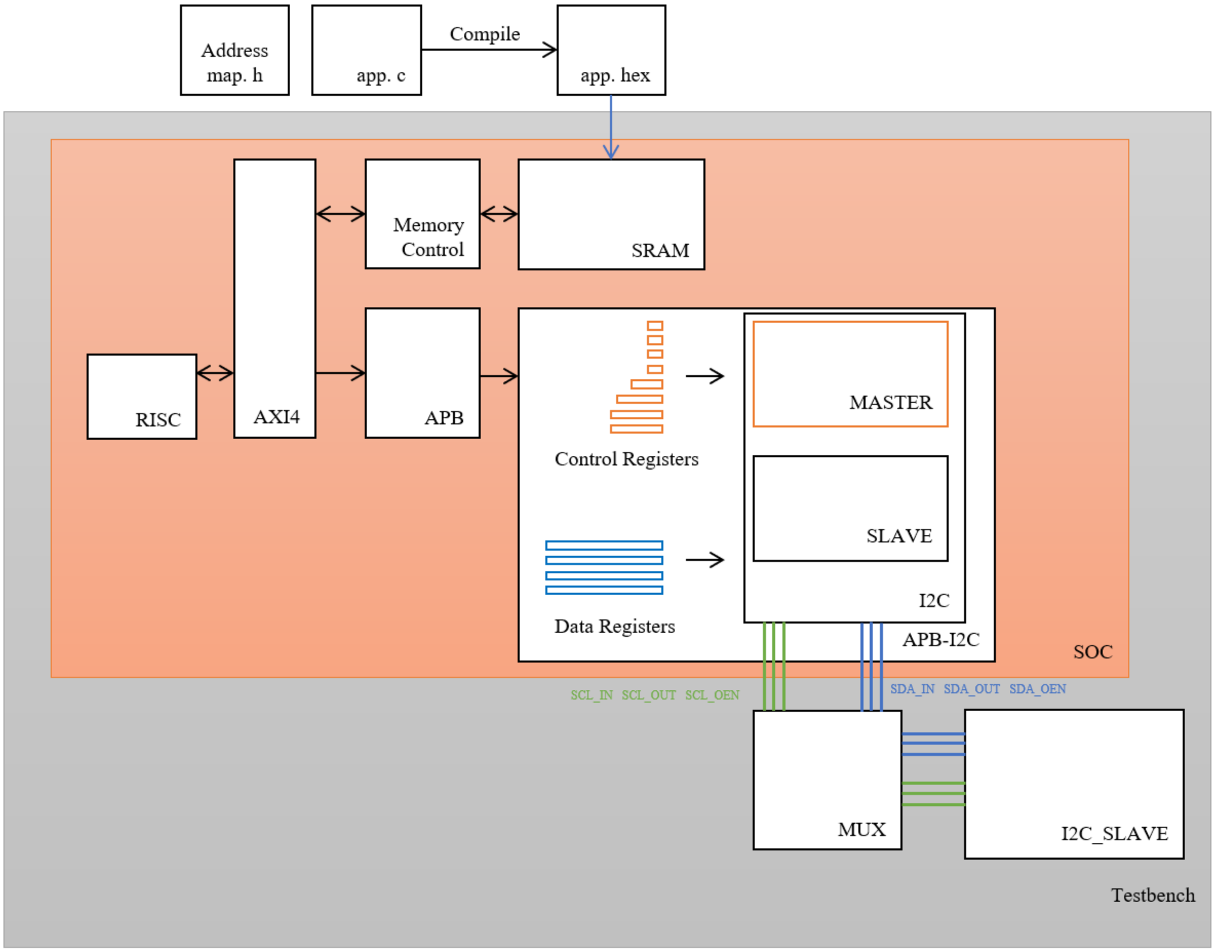


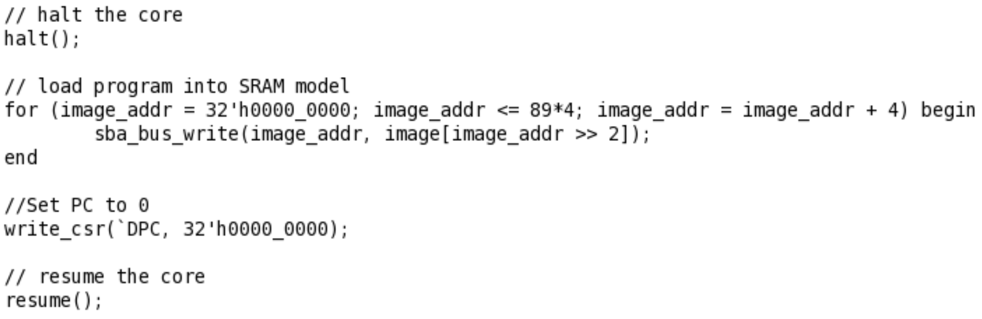
Figure 6.6: SoC-I^{2}C Simulation Flow

when the target address is a peripheral component, with AXI4_int, the AXI4 slave will pass the information to the APB master. Similarly, APB_int will help the APB slave connect with the corresponding address in the APB_I^{2}C module, and then the internal logic will store the data in the register. As described in Fig.6.6, the registers are connected to the input and output of the I^{2}C module. In short, through the RISC, AXI4, and APB protocols, the I^{2}C is implemented using the C program.

6.3.2 Design I^{2}C operation program

To run the simulation program, we must write the corresponding C code, similar to the operation when debugging SPI. Fig.6.8 is the code to complete the write operation for the code written to drive I^{2}C, and the code for the read operation is similar.

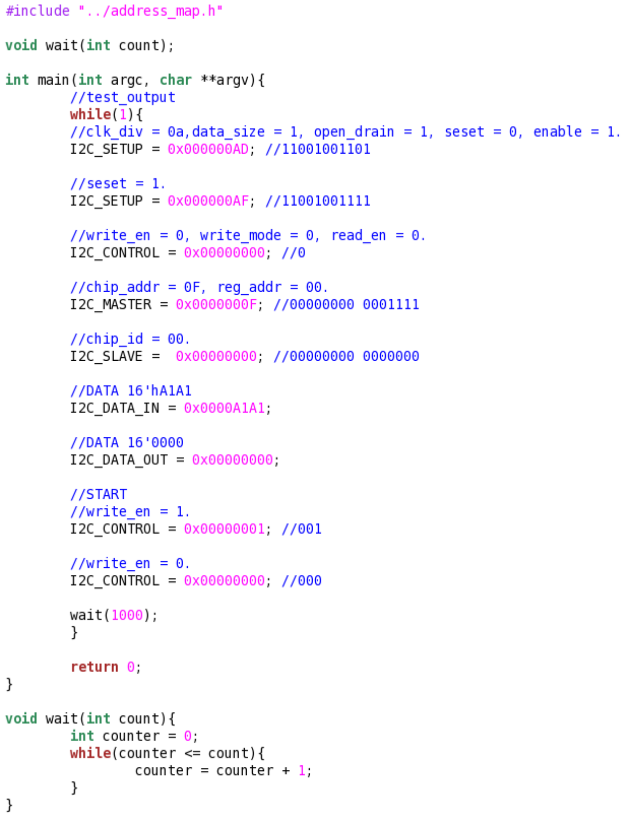
Firstly, the code must be used with the address map because it contains the access address and the corresponding name. In the main program, the I^{2}C control registers are assigned sequentially, and the specific functions of the control registers are described in detail in 'Build APB to I^{2}C Logic'. Unlike the SPI-I^{2}C protocol converter chip test, the wait ( ) function does not need to be added after each assignment. However, it is used at the end of the C code because we need a period of time when the SoC works stably to observe the operation of the I^{2}C.



```
// halt the core
halt();

// load program into SRAM model
for (image_addr = 32'h0000_0000; image_addr <= 89*4; image_addr = image_addr + 4) begin
        sba_bus_write(image_addr, image[image_addr >> 2]);
end

//Set PC to 0
write_csr(`DPC, 32'h0000_0000);

// resume the core
resume();
```

Figure 6.7: Loading Program Code

Before loading data into SRAM, the core needs to be halted. Then load the contents of the .hex file into the SRAM model used by the testbench. There is code in the testbench that loads the contents of the .hex file into the register array called image. After the program is finished loading, we need to set the Program Counter (PC) and resume the core.



```c
#include "../address_map.h"

void wait(int count);

int main(int argc, char **argv){
        //test_output
        while(1){
        //clk_div = 0a,data_size = 1, open_drain = 1, seset = 0, enable = 1.
        I2C_SETUP = 0x000000AD; //11001001101

        //seset = 1.
        I2C_SETUP = 0x000000AF; //11001001111

        //write_en = 0, write_mode = 0, read_en = 0.
        I2C_CONTROL = 0x00000000; //0

        //chip_addr = 0F, reg_addr = 00.
        I2C_MASTER = 0x0000000F; //00000000 0001111

        //chip_id = 00.
        I2C_SLAVE =  0x00000000; //00000000 0000000

        //DATA 16'hA1A1
        I2C_DATA_IN = 0x0000A1A1;

        //DATA 16'0000
        I2C_DATA_OUT = 0x00000000;

        //START
        //write_en = 1.
        I2C_CONTROL = 0x00000001; //001

        //write_en = 0.
        I2C_CONTROL = 0x00000000; //000

        wait(1000);
        }

        return 0;
}

void wait(int count){
        int counter = 0;
        while(counter <= count){
                counter = counter + 1;
        }
}
```

Figure 6.8: Write Program Code

6.3.3 Simulation Result

The complete simulation results show that adding the I^{2}C module to the SoC is successful, and the specific results are shown in Fig.6.9 read process and Fig.6.10 write process. The execution flow of the C code is shown in Fig.6.6. The simulation results section shows that the C code is loaded into the SRAM, the data passes through the APB slave to the I^{2}C control register, and the I^{2}C executes the instruction. Compared with the I^{2}C module simulation, the system simulation includes more content, such as loading and running the program. The details of the operation of I^{2}C in Modelsim will not be discussed in depth because it has been analyzed in detail in the previous chapters.

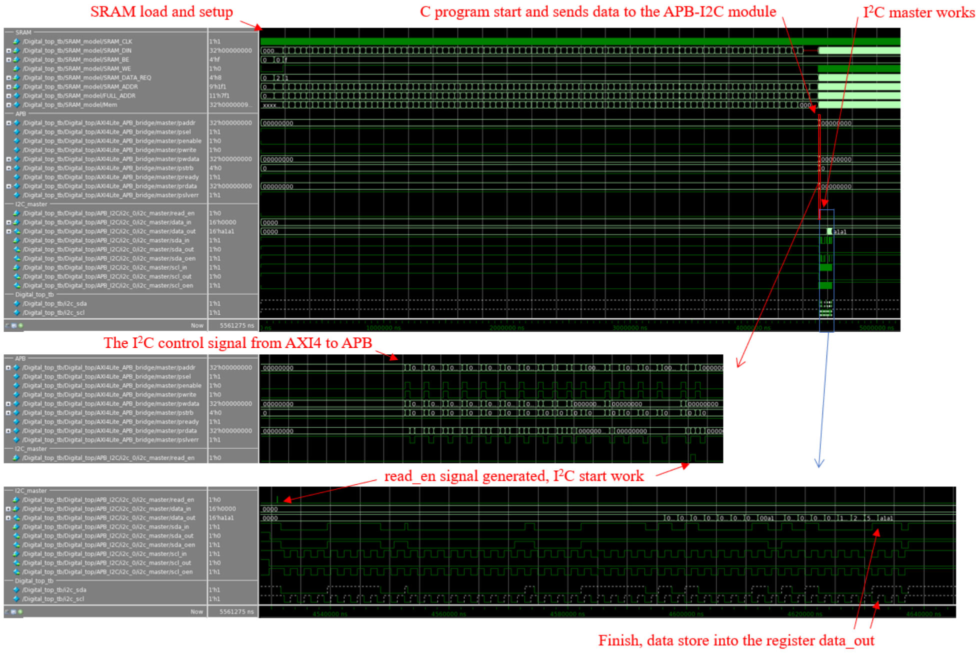
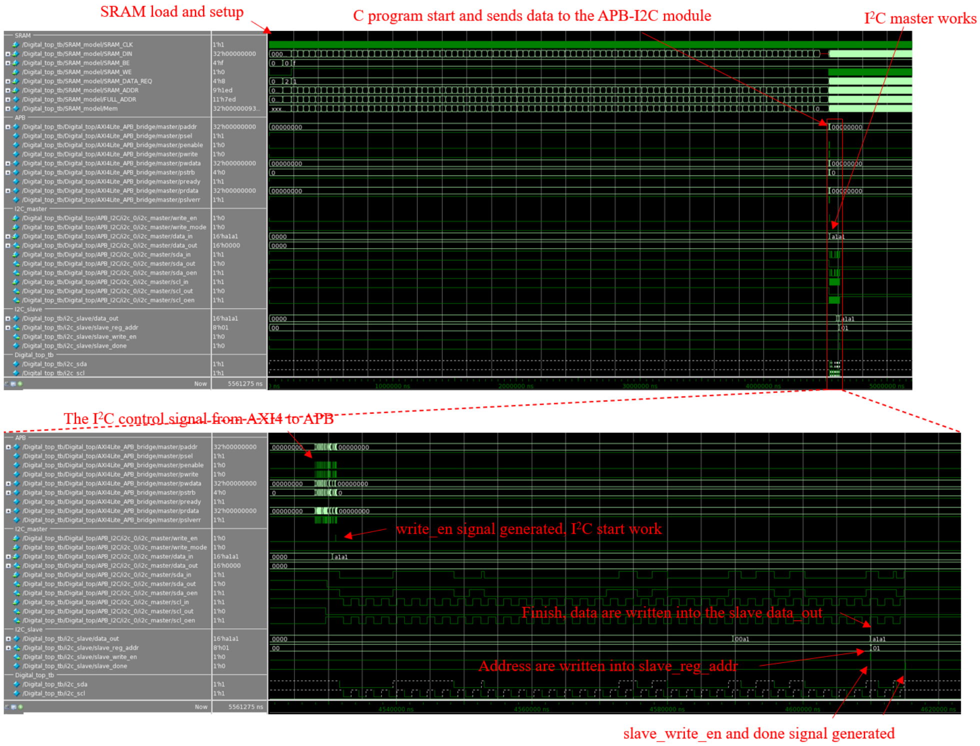

SRAM load and setup
C program start and sends data to the APB-I2C module
I2C master works
The I2C control signal from AXI4 to APB
read_en signal generated, I2C start work
Finish, data store into the register data_out

Figure 6.9: Simulation of SoC-I^{2}C read Operation

By analyzing the two figures, we can infer that the simulation appears to take a considerable amount of time. Still, most of this time is taken up by the testbench code writing to the SRAM, as highlighted in the figure. Once the paused core is restarted, the instruction stored in SRAM is sent through the core, AXI4, and APB. Finally reaches the control register, which triggers the I2C module. Although the simulation results do not show the signals of the AXI4 interface, the interface is highly complex, and displaying the signals would not be informative. However, assuming that the AXI4 signals coincide with the APB interface signals is reasonable. We have completed the basic design of the I^{2}C module, integrated it into the existing SoC, and used C code to control it. We have made significant progress in this regard and have successfully simulated the system. However, we have identified a key issue - the lack of an interrupt system in the SoC. Incorporating an interrupt system will improve read and write efficiency, a critical aspect in building larger systems.

Although the simulation of SoC-I2C can only be carried out on Modelsim, we have proved that the I2C integrated into the SoC can normally work in the chip. We aim to optimize the performance and power consumption of I2C to realize the work of communication in distributed systems.


SRAM load and setup
C program start and sends data to the APB-I2C module
I^{2}C master works
The I^{2}C control signal from AXI4 to APB
write_en signal generated, I^{2}C start work
Finish, data are written into the slave data_out
Address are written into slave_reg_addr
slave_write_en and done signal generated

```
SRAM
/Digital_top_tb/SRAM_model/SRAM_CLK                                 1'h1
/Digital_top_tb/SRAM_model/SRAM_DIN                                 32'h00000000
/Digital_top_tb/SRAM_model/SRAM_BE                                  4'hf
/Digital_top_tb/SRAM_model/SRAM_WE                                  1'h0
/Digital_top_tb/SRAM_model/SRAM_DATA_REQ                            4'h8
/Digital_top_tb/SRAM_model/SRAM_ADDR                                9'h1ed
/Digital_top_tb/SRAM_model/FULL_ADDR                                11'h7ed
/Digital_top_tb/SRAM_model/Mem                                      32'h00000093...
APB
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/paddr        32'h00000000
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/psel         1'h1
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/penable      1'h0
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pwrite       1'h0
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pwdata       32'h00000000
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pstrb        4'h0
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pready       1'h1
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/prdata       32'h00000000
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pslverr      1'h1
I2C_master
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/write_en       1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/write_mode     1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/data_in        16'ha1a1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/data_out       16'h0000
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/sda_in         1'h1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/sda_out        1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/sda_oen        1'h1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/scl_in         1'h1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/scl_out        1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/scl_oen        1'h1
I2C_slave
/Digital_top_tb/i2c_slave/data_out                                  16'ha1a1
/Digital_top_tb/i2c_slave/slave_reg_addr                            8'h01
/Digital_top_tb/i2c_slave/slave_write_en                            1'h0
/Digital_top_tb/i2c_slave/slave_done                                1'h0
Digital_top_tb
/Digital_top_tb/i2c_sda                                             1'h1
/Digital_top_tb/i2c_scl                                             1'h1
                                                               Now  5561275 ns
```

```
APB
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/paddr        32'h00000000
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/psel         1'h1
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/penable      1'h0
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pwrite       1'h0
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pwdata       32'h00000000
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pstrb        4'h0
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pready       1'h1
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/prdata       32'h00000000
/Digital_top_tb/Digital_top/AXI4Lite_APB_bridge/master/pslverr      1'h1
I2C_master
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/write_en       1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/write_mode     1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/data_in        16'ha1a1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/data_out       16'h0000
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/sda_in         1'h1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/sda_out        1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/sda_oen        1'h1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/scl_in         1'h1
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/scl_out        1'h0
/Digital_top_tb/Digital_top/APB_I2C/i2c_0/i2c_master/scl_oen        1'h1
I2C_slave
/Digital_top_tb/i2c_slave/data_out                                  16'ha1a1
/Digital_top_tb/i2c_slave/slave_reg_addr                            8'h01
/Digital_top_tb/i2c_slave/slave_write_en                            1'h0
/Digital_top_tb/i2c_slave/slave_done                                1'h0
Digital_top_tb
/Digital_top_tb/i2c_sda                                             1'h1
/Digital_top_tb/i2c_scl                                             1'h1
                                                               Now  5561275 ns
```

Figure 6.10: Simulation of SoC-I^{2}C Write Operation

Chapter 7

Conclusions and Future Directions

Based on Fabric-Based Self-Powered Systems, we hope to solve the interconnection problem between chips by exploring two synchronous serial communication protocols, SPI and I^{2}C. This paper proposes the complete process of digital design of the I^{2}C module. It explores the performance and power consumption performance of I^{2}C and SPI protocol modules under ultra-low power consumption. The SPI-I^{2}C protocol converter chip and optimized SoC proposed in this paper provide the basis for future Self-Powered Fabric-Based Systems. Furthermore, as we designed and implemented the I^{2}C module, we identified several limitations in the current design. We intend to address these issues in future iterations to enhance its functionality and performance.

7.1 I^{2}C Protocol Module

7.1.1 Optimize Open-Drain Circuit

Open-drain has been a successful design in facilitating interconnection among multiple devices. However, the issue of the short-circuit current on the pull-up resistor can cause problems, mainly when the bus is at a low voltage (0). This significant pain point demands attention in ultra-low power consumption circuits. Balancing the need for pull-up capability with power consumption is crucial for optimizing the open-drain structure design. We want to find a balance between power consumption and pull-up capability. In addition, the overall system design should be considered to optimize the open-drain structure. For instance, reducing the number of interconnected devices or grouping them into smaller segments can lower the demand for pull-up capability and reduce power consumption. Another approach is implementing bus arbitration techniques that can reduce competition for pull-up capability and, in turn, lower power consumption.

7.1.2 Design I^2C Slave without Clock

Our testing found that the oscillator consumes significantly more power than the I^2C module. However, it's worth noting that the Master generates the SCL signal. Therefore, if the slave can operate at the same frequency as the SCL, it can save energy and reduce the sensor component's area that doesn't depend on the clock. We have conducted experiments to verify that when the Slave's operating frequency is approximately equal to that of SCL, we can observe the waveform on SDA. Unfortunately, the current results have shown that the waveform is unstandard.

7.1.3 Reliability and Efficiency

At this stage, the I^2C module has a simple design and can only perform basic functions. In addition, it lacks an automatic reset capability, which means that if an error occurs during operation, it requires manual debugging and restarting. To avoid such inconvenience, it's essential to have automatic reset functionality, especially in a complete system operation. Therefore, we aim to improve the I^2C module by adding this feature. Additionally, we plan to optimize the code to enhance the module's functionality and enable it to perform more tasks efficiently.

7.2 System on Chip

To enhance chip communication efficiency and reduce power consumption, we suggest incorporating an interrupt signal into the SoC. We have already allocated specific ports in the I^2C module design for this purpose, which can simplify the design process. This will improve chip communication efficiency while reducing the microcontroller operation required, resulting in lower power consumption. However, implementing this change will require revising the RSIC-V core structure and AXI4 agreement, which may be time-consuming. Nevertheless, the benefits of improved communication efficiency and reduced power consumption make it a worthwhile endeavor.

Additionally, it is crucial to implement direct access to the I2C block to the SRAM. After receiving the read command, the I2C Slave needs to locate the data of the target memory address through the AXI4 and APB interfaces with the help of RISC. It is best to design a memory direct access module to optimize the process to enable direct reading of SRAM. This will reduce energy consumption and time spent on the operation.

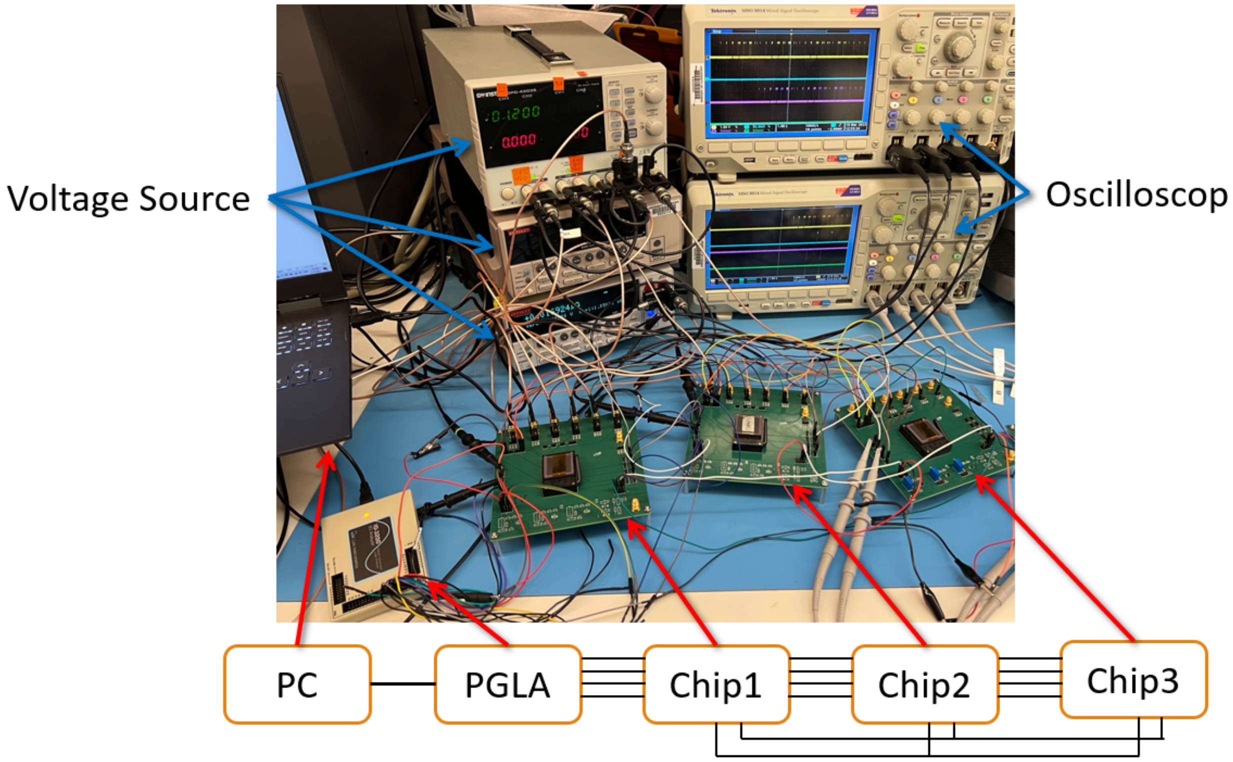
Appendix A

Verilog Code of I^{2}C

https://github.com/Zhenghong-C-LM/i2c_module_desgin.git

Appendix B

SPI-I^{2}C Protocol Converter Chip Test Environment



Glossary

Acronyms and Abbreviations	
ADC	Analog to Digital Converter
AI	Artificial Intelligence
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DDR	Double Data Rate
DRAM	Dynamic Random-Access Memory
EDA	Electronic Design Automation
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GPIO	General Purpose Input/Output
HDL	Hardware Description Language
IC	Integrated Circuit
I2C	Inter-Integrated Circuit
I^2C	Inter-Integrated Circuit
IoT	Internet of Things

I/O	Input/Output
I/F	Interface
ISA	Instruction Set Architecture
LSB	Least Significant Bit
MHz	Mega Hertz
MISO	Master In Slave Out
MOSI	Master Out Slave In
MSB	Most Significant Bit
ML	Machine Learning
OS	Operating System
PGLA	pattern generator logic analyzer
PPAE	Power, Performance, Area and Energy
RAM	Random-Access Memory
RISC-V	Reduced Instruction Set Computer Fifth Generation
ROM	Read Only Memory
RTL	Register Transfer Level
SCL	Serial Clock
SCK	Serial Clock
SDA	Serial Data
SoC	System on Chip
SS	System Select
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
UART	Universal Asynchronous Receiver-Transmitter
UVA	University of Virginia

Bibliography

[1] T. Ajmal, D. Jazani, and B. Allen, “Design of a compact rf energy harvester for wireless sensor networks,” in *IET Conference on Wireless Sensor Systems (WSS 2012)*, 2012, pp. 1–5.

[2] J. Zhang, R. Wang, Y. Qian, and Q. Wang, “A coverage control algorithm based on probability model for three-dimensional wireless sensor networks,” in *2012 11th International Symposium on Distributed Computing and Applications to Business, Engineering Science*, 2012, pp. 169–173.

[3] M. P. Kumar and U. Rani. Nelakuditi, “Iot and i2c protocol based m-health medication assistive system for elderly people,” in *2019 IEEE 16th India Council International Conference (INDICON)*, 2019, pp. 1–4.

[4] M. I. M. Bakri, A. Zakaria, S. M. M. S. Zakaria, L. M. Kamarudin, A. Y. M. Shakaff, F. S. A. Saad, M. F. Ibrahim, and M. H. M. Razali, “Plant bio-absorber for ammonia gas absorption using i2c interface data acquisition system,” in *2014 2nd International Conference on Electronic Design (ICED)*, 2014, pp. 488–492.

[5] J. H. Moon, B.-J. Kim, Y. Jang, T. J. Mun, H. Kim, and S. J. Kim, “Self-powered inertial sensor based on carbon nanotube yarn,” *IEEE Transactions on Industrial Electronics*, vol. 68, no. 9, pp. 8904–8910, 2021.

[6] F.-R. Fan, Z.-Q. Tian, and Z. Lin Wang, “Flexible triboelectric generator,” *Nano Energy*, vol. 1, no. 2, p. 328–334, 2012.

[7] A. J. Bandodkar, J.-M. You, N.-H. Kim, Y. Gu, R. Kumar, A. M. Mohan, J. Kurniawan, S. Imani, T. Nakagawa, B. Parish, and et al., “Soft, stretchable, high power density electronic skin-based biofuel cells for scavenging energy from human sweat,” *Energy amp; Environmental Science*, vol. 10, no. 7, p. 1581–1589, 2017.

[8] A. Lanata, “Wearable systems for home monitoring healthcare: The photoplethysmography success pros and cons,” *Biosensors*, vol. 12, no. 10, p. 861, 2022.

[9] Y. Su and R. Gururajan, “The determinants for adoption of wearable computer systems in traditional chinese hospital,” in *2010 Asia-Pacific Conference on Wearable Computing Systems*, 2010, pp. 375–378.

[10] “Ieee standard for wearable consumer electronic devices–overview and architecture,” *IEEE Std 360-2022*, pp. 1–35, 2022.

[11] V. Gyanchandani, S. N. Masabi, and H. Fu, “A self-powered wearable device using the photovoltaic effect for human heath monitoring,” in *2021 IEEE 20th International Conference on Micro and Nanotechnology for Power Generation and Energy Conversion Applications (PowerMEMS)*, 2021, pp. 60–63.

[12] G. Loke, T. Khudiyev, B. Wang, S. Fu, S. Payra, Y. Shaoul, J. Fung, I. Chatziveroglou, P.-W. Chou, I. Chinn, and et al., “Digital electronics in fibres enable fabric-based machine-learning inference,” *Nature Communications*, vol. 12, no. 1, 2021.

[13] F. P. Oikonomou, J. Ribeiro, G. Mantas, J. M. C. Bastos, and J. Rodriguez, “A hyperledger fabric-based blockchain architecture to secure iot-based health monitoring systems,” in *2021 IEEE International Mediterranean Conference on Communications and Networking (MeditCom)*, 2021, pp. 186–190.

[14] D. Trivedi, A. Khade, K. Jain, and R. Jadhav, “Spi to i2c protocol conversion using verilog,” in *2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA)*, 2018, pp. 1–4.

[15] L. N. Pintilie, T. Pop, I. C. Gros, and A. Mihai Iuoras, "An i2c and ethernet based open-source solution for home automation in the iot context," in *2019 54th International Universities Power Engineering Conference (UPEC)*, 2019, pp. 1–4.

[16] "Tmp114." [Online]. Available: https://www.ti.com/product/TMP114?keyMatch=TMP114&amp;tisearch=search-everything&amp;usecase=GPN

[17] "Tmp126." [Online]. Available: https://www.ti.com/product/TMP126?keyMatch=TMP126&amp;tisearch=search-everything&amp;usecase=GPN

[18] "Tmp144." [Online]. Available: https://www.ti.com/product/TMP144?keyMatch=TMP144&amp;tisearch=search-everything&amp;usecase=GPN

[19] "Hdc3022." [Online]. Available: https://www.ti.com/product/HDC3022?keyMatch=HDC3022&amp;tisearch=search-everything&amp;usecase=GPN

[20] "Opt3005." [Online]. Available: https://www.ti.com/product/OPT3005

[21] "serial peripheral interface (spi™) - microchip technology." [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/70067b.pdf

[22] M. B. Aykenar, G. Soysal, and M. Efe, "Design and implementation of a lightweight spi master ip for low cost fpgas," in *2020 28th Signal Processing and Communications Applications Conference (SIU)*, 2020, pp. 1–4.

[23] M. Kuhrmann and J. Müench, "Spi is dead, isn't it? clear the stage for continuous learning!" in *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, 2019, pp. 9–13.

[24] "i2c-bus specification and user manual - nxp community." [Online]. Available: https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/nxp-designs/931/1/UM10204.pdf

[25] S. M. Kalyankar and S. Sawarkar, “Design of multinode reconfigurable multiprocessor network for embedded systems,” in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 2017, pp. 3239–3243.

[26] P. Bagdalkar and L. Ali, “Interfacing of light sensor with fpga using i2c bus,” in *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*, 2020, pp. 843–846.

[27] R. S. S. Kumari and C. Gayathri, “Interfacing of mems motion sensor with fpga using i2c protocol,” in *2017 International Conference on Innovations in Information, Embedded and Communication Systems (ICIIECS)*, 2017, pp. 1–5.

[28] I. Ali, S. H. Cho, D. G. Kim, M. R. U. Rehman, and K.-Y. Lee, “A design of ultra low power i2c synchronous slave controller with interface voltage level independency in 180 nm cmos technology,” in *2017 International SoC Design Conference (ISOCC)*, 2017, pp. 262–263.

[29] L. Bacciarelli, G. Lucia, S. Saponara, L. Fanucci, and M. Forliti, “Design, testing and prototyping of a software programmable i2c/spi ip on amba bus,” in *2006 Ph.D. Research in Microelectronics and Electronics*, 2006, pp. 373–376.

[30] “Modelsim hdl simulator.” [Online]. Available: https://eda.sw.siemens.com/en-US/ic/modelsim/

[31] D. Goswami, K.-h. Tsai, M. Kassab, T. Kobayashi, J. Rajski, B. Swanson, D. Walters, Y. Sato, T. Asaka, and T. Aikyo, “At-speed testing with timing exceptions and constraints-case studies,” in *2006 15th Asian Test Symposium*, 2006, pp. 153–162.

[32] “genus synthesis solution.” [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/genus-synthesis-solution.html

[33] “innovus implementation system.” [Online]. Available: https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html

[34] D. S. Dawoud and P. Dawoud, *6 Serial Peripheral Interface (SPI)*, 2020, pp. 191–244.

[35] R. Chandaluri and U. Nelakuditi, "Design and implementation of axi4-lite interface in zynq soc," in *2022 IEEE Delhi Section Conference (DELCON)*, 2022, pp. 1–4.

[36] C. Duran, D. L. Rueda, G. Castillo, A. Agudelo, C. Rojas, L. Chaparro, H. Hurtado, J. Romero, W. Ramirez, H. Gomez, J. Ardila, L. Rueda, H. Hernandez, J. Amaya, and E. Roa, "A 32-bit risc-v axi4-lite bus-based microcontroller with 10-bit sar adc," in *2016 IEEE 7th Latin American Symposium on Circuits Systems (LASCAS)*, 2016, pp. 315–318.

[37] K. Rawat, K. Sahni, and S. Pandey, "Rtl implementation for amba asb apb protocol at system on chip level," in *2015 2nd International Conference on Signal Processing and Integrated Networks (SPIN)*, 2015, pp. 927–930.

[38] C. Ma, Z. Liu, and X. Ma, "Design and implementation of apb bridge based on amba 4.0," in *2011 International Conference on Consumer Electronics, Communications and Networks (CECNet)*, 2011, pp. 193–196.