ABSTRACT

This mini-project focuses on the design of an I2C slave microarchitecture using Verilog, a hardware description language. The I2C protocol, known for its simplicity and efficiency, is widely used for inter-IC communication on printed circuit boards. The project involves developing a single master-single slave setup, where the master device controls the communication and the slave device responds to the master's requests. Key features of the I2C protocol, such as its two-wire communication system using the Serial Data Line (SDA) and Serial Clock Line (SCL), multi-device addressability, and support for various data transfer modes, are explored. The project aims to demonstrate the practical application of the I2C protocol in digital design, focusing on synchronization, data transfer, and communication management

CONTENTS

CHAPTER 1

CONCEPTION OF THE PROJECT

1.1 INTRODUCTION

The Inter-Integrated Circuit (I2C) protocol, developed by Philips Semiconductors (now NXP Semiconductors) in 1982, is a widely used communication protocol designed to facilitate efficient data transfer between inte- grated circuits on a printed circuit board (PCB). This two-wire, bidirectional protocol employs a Serial Data Line (SDA) and a Serial Clock Line (SCL) to enable serial, 8-bit oriented data transfer, making it an ideal choice for connecting multiple devices with minimal wiring.

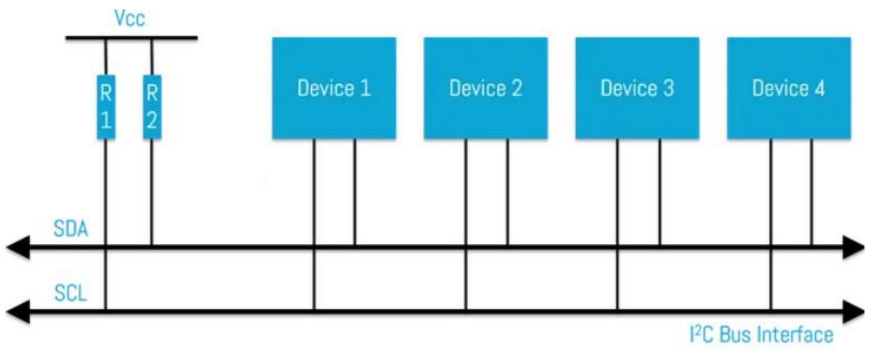


Figure 1.1: Basic block diagram of I2C

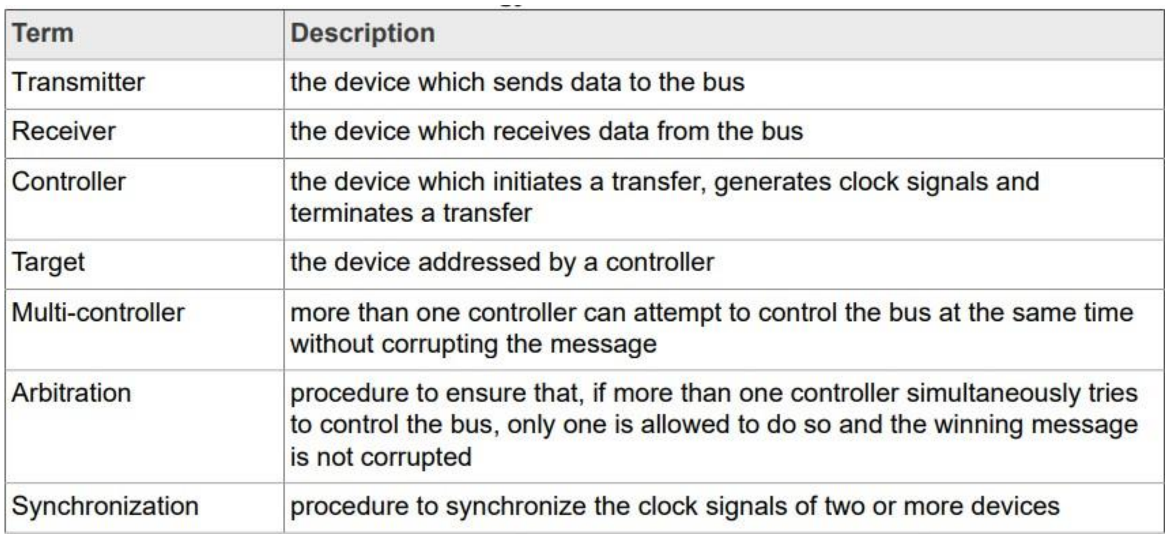
The figure illustrates a typical I2C bus configuration, showcasing the multi-master, multi-slave nature of the protocol. In this setup, multiple master devices, such as Master One and Master Two, and slave devices, such as Slave One, Slave Two, and Slave Three, are connected to the same bus. Masters can initiate communication by generating clock signals on the SCL line and sending data over the SDA line, while slaves respond when addressed. Pull-up resistors connected to the SDA and SCL lines ensure proper voltage levels, preventing bus floating and enabling reliable communication.

I2C is a versatile protocol included in various control architectures, such as the System Management Bus (SMBus) and the Power Management Bus (PMBus), providing a reliable means of communication for system management and power control. The I2C protocol supports various data transfer modes, each catering to different speed and application requirements, including Standard Mode (up to 100 kbit/s), Fast Mode (up to 400 kbit/s), Fast Mode Plus (up to 1 Mbit/s), High-Speed Mode (up to 3.4 Mbit/s), and Ultra-Fast Mode (up to 5 Mbit/s).

In this mini-project, we design an I2C slave module using Verilog, highlighting key aspects of the I2C protocol like synchronization, data transfer, and communication management. The project aims to demonstrate the practical application of the I2C protocol in digital design, providing a clear understanding of its functionality.

1.2 I2C Bus Terminologies

I2C communication protocol uses several terminologies like transmitter, receiver, controller, target, multi- controller, arbitration and synchronization.



Term	Description
Transmitter	the device which sends data to the bus
Receiver	the device which receives data from the bus
Controller	the device which initiates a transfer, generates clock signals and terminates a transfer
Target	the device addressed by a controller
Multi-controller	more than one controller can attempt to control the bus at the same time without corrupting the message
Arbitration	procedure to ensure that, if more than one controller simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted
Synchronization	procedure to synchronize the clock signals of two or more devices

Figure 2.1: I2C bus terminology

1.3 Features of I2C

The I2C protocol boasts several key features that make it a popular choice for inter-IC communication:

- Each device connected to the bus is software addressable by a unique address and simple controller/target relationships exist at all times; controllers can operate as controller-transmitters or as controller-receivers.
- It is a true multi-controller bus including collision detection and arbitration to prevent data corruption if two or more controllers simultaneously initiate data transfer.
- Serial, 8-bit oriented, unidirectional data (write only) transfers up to 5 Mbit/s in Ultra Fast-mode. On-chip filtering rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance. More capacitance may be allowed under some conditions.
- up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s .

CHAPTER 2

DESIGN OF THE PROJECT

2.1 SDA and SCL signals

The Serial Data Line (SDA) is the data line. All the data transfer among the devices takes place through this line. The Serial Clock Line (SCL) is the serial clock. I2C is a synchronous protocol, and hence, SCL is used to synchronize all the devices and the data transfer together.

Both SDA and SCL are bidirectional lines, connected to a positive supply voltage via a current-source or pull- up resistor. When the bus is free, both lines are HIGH. The output stages of devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function.

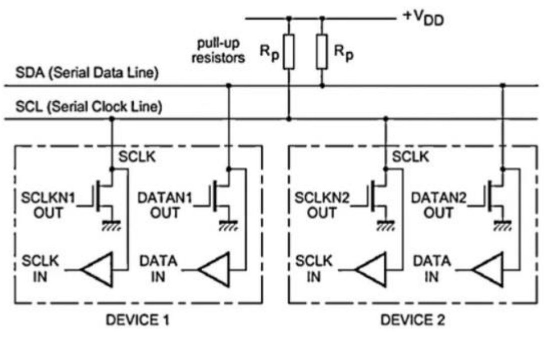


Figure 2.2: I2C bus with Pull-up resistors and Open drain lines

2.2 Open-Drain / Open-Collector

In open-drain (or open-collector for bipolar transistors) configuration, the output can either pull the line to a low voltage (ground) or leave it floating (high impedance). It cannot drive the line to a high voltage directly.

This type of output is essential for the wired-AND functionality in I2C, where multiple devices can share the same bus lines (SDA and SCL) without interfering with each other. It ensures that if any device pulls the line low, the line will be low, regardless of other devices.This is the standard output configuration for both SDA and SCL in I2C to allow for multiple masters and slaves to communicate on the same bus.

2.3 Pull-Up Resistors

Pull-up resistors are connected between the bus lines (SDA and SCL) and the supply voltage (Vcc). They pull the line to a high state when no device is actively pulling it low. These resistors ensure that the default state of the bus lines is high, which is necessary because the open-drain outputs can only pull the lines low. Pull-up resistors provide the necessary current to set the line to a high state when it is not being driven low. Essential for the proper functioning of the open-drain configuration, ensuring that the lines return to a high state when no device is pulling them low.

2.4 Bidirectional IO

The SDA line in I2C is bidirectional, meaning it can serve as both an input and an output. The direction depends on the phase of the communication, whether data is being sent to or received from the master.

This dual role is critical in I2C, as it allows the master and slave devices to communicate over the same line for both transmitting and receiving data. During an address phase, the master writes the address and read/write bit; subsequently, data may flow in either direction depending on the operation.

2.5 Tri-State Logic

Tri-state logic is used to implement the high-impedance state in the open-drain outputs. When a device is not actively driving the bus, it leaves the line in a high-impedance state, effectively disconnecting itself from the bus.This allows multiple devices to share the bus without interference, as only one device actively drives the line at any given time, while others are in a high-impedance state. It is mployed when devices on the bus are not communicating, ensuring no contention on the lines.

2.6 I2C Frame Structure

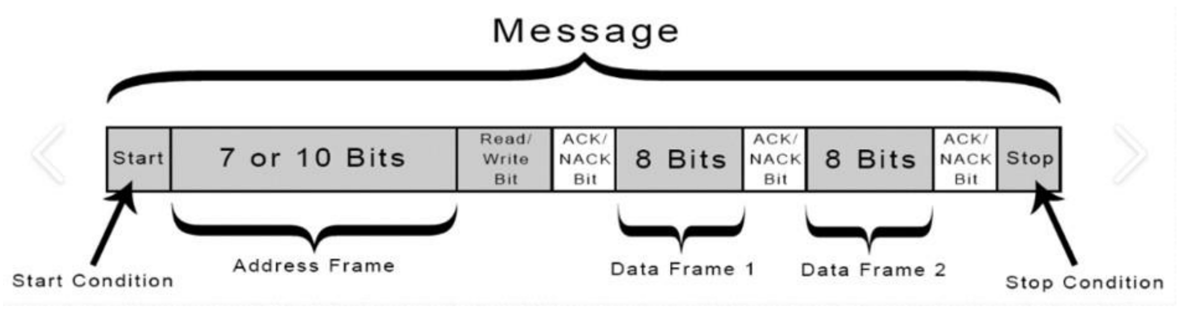


Figure 2.3: I2C frame structure

The I2C frame structure is consists of start bit, address bits, read/write bit, acknowledgement bit, data bytes and stop bit.

2.7 Data validity

The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW (see Figure 2.4). One clock pulse is generated for each data bit transferred.

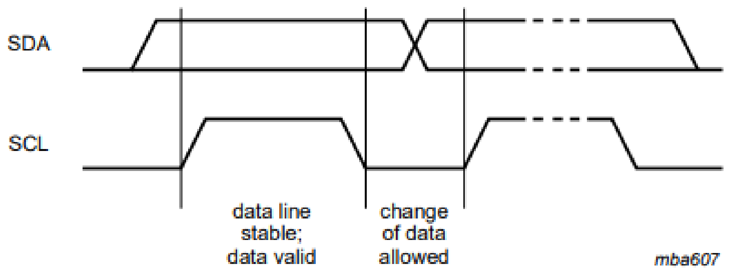


Figure 2.4: Bit transfer on the I2C-bus

2.8 START and STOP conditions

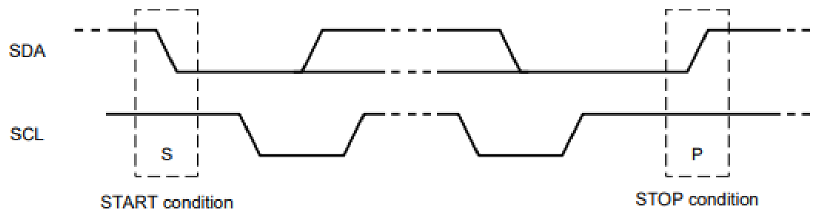


Figure 2.5: START and STOP conditions

All transactions begin with a START (S) and are terminated by a STOP (P) (see Figure 2.5). A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition.

START and STOP conditions are always generated by the controller. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition.

The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical

2.9 The target address and R/W bit

After the START condition (S), a target address is sent. This address is seven bits long followed by an eighth bit which is a data direction bit (R/W) — a ‘zero’ indicates a transmission (WRITE), a ‘one’ indicates a request for data (READ). Ten bits addressing is also possible. A data transfer is always terminated by a STOP condition
(P) generated by the controller. However, if a controller still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another target without first generating a STOP condition. Various combinations of read/ write formats are then possible within such a transfer.

2.10 Data Byte

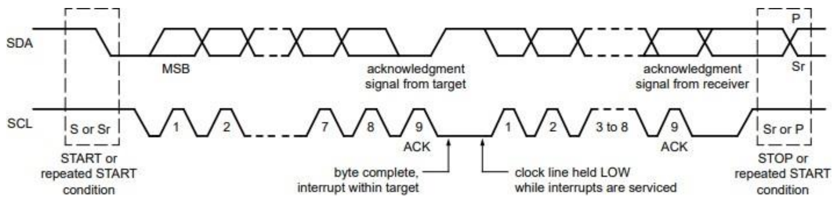

SDA
SCL
P
MSB
acknowledgment signal from target
acknowledgment signal from receiver
Sr
S or Sr
1
2
7
8
9
ACK
1
2
3 to 8
9
ACK
Sr or P
START or repeated START condition
byte complete, interrupt within target
clock line held LOW while interrupts are serviced
STOP or repeated START condition

Figure 2.6: Data transfer on the I2C-bus

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge bit. Data is transferred with the Most Significant Bit (MSB) first (see Figure 2.6). If a target cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the controller into a wait state. Data transfer then continues when the target is ready for another byte of data and releases clock line SCL.

2.11 Acknowledge (ACK) and Not Acknowledge (NACK)

The acknowledge takes place after every byte. The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. The controller generates all clock pulses, including the acknowledge ninth clock pulse.

The Acknowledge signal is defined as follows: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse (see Figure 2.4).

When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal. The controller can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer. There are five conditions that lead to the generation of a NACK:

2.11.1.1 No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.

2.11.1.2 The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the controller.

2.11.1.3 During the transfer, the receiver gets data or commands that it does not understand.

2.11.1.4 During the transfer, the receiver cannot receive any more data bytes.

2.11.1.5 A controller-receiver must signal the end of the transfer to the target transmitter.

2.12 Clock synchronization

Two controllers can begin transmitting on a free bus at the same time and there must be a method for deciding which takes control of the bus and complete its transmission. This is done by clock synchronization and arbitration. In single controller systems, clock synchronization and arbitration are not needed.

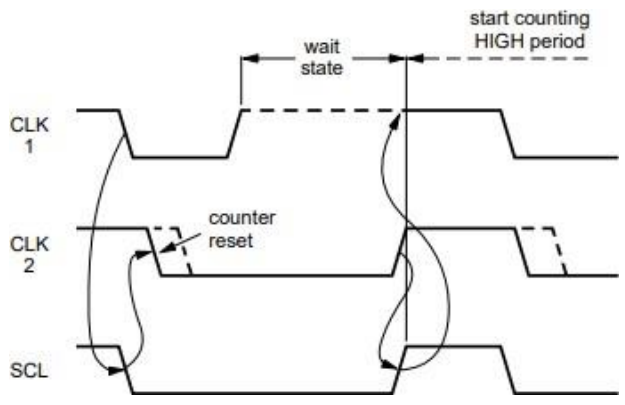


Figure 2.7: Clock synchronization during arbitration

Clock synchronization is performed using the wired-AND connection of I2C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line causes the controllers concerned to start counting off their LOW period and, once a controller clock has gone LOW, it holds the SCL line in that state until the clock HIGH state is reached (see Figure 2.7). However, if another clock is still within its LOW period, the LOW to HIGH transition of this clock may not change the state of the SCL line. The SCL line is therefore held LOW by the controller with the longest LOW period. Controllers with shorter LOW periods enter a HIGH wait-state during this time.

When all controllers concerned have counted off their LOW period, the clock line is released and goes HIGH. There is then no difference between the controller clocks and the state of the SCL line, and all the controllers start counting their HIGH periods. The first controller to complete its HIGH period pulls the SCL line LOW again.

CHAPTER 3

IMPLEMENTATION OF THE PROJECT

3.1 Top-level Master Slave Communication

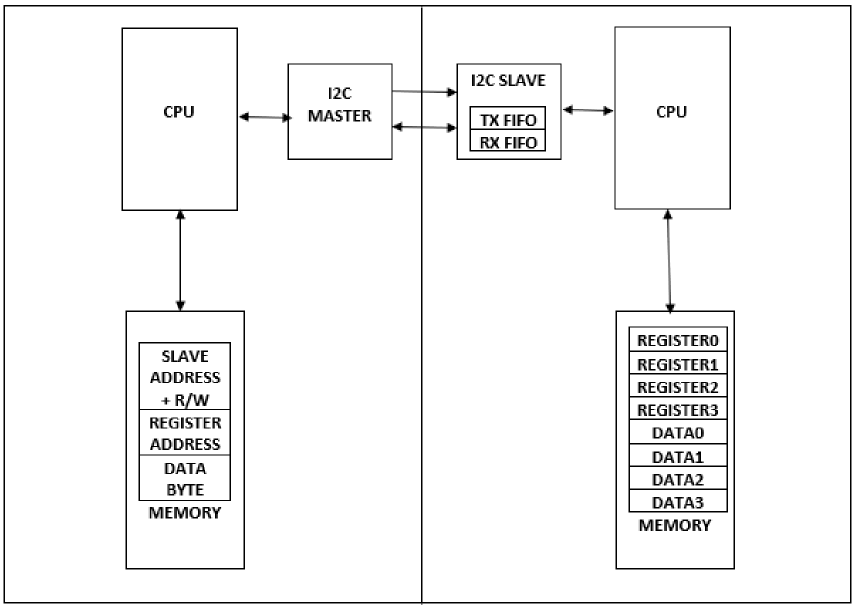


Figure 3.1: Top level functional block diagram

The top-level functional diagram depicts a typical I2C communication setup involving an I2C master and an I2C slave. Each device is connected to its own CPU and memory unit, facilitating data exchange and processing.

I2C Master: On the left side of Figure 3.1, the CPU interacts with memory to manage data, including the slave address, register address, and data bytes to be transmitted. This data is then sent to the I2C master controller, which initiates and manages communication on the I2C bus. The master generates the necessary clock signals (SCL) and controls the data line (SDA) to communicate with the I2C slave.

I2C Slave: The right side shows the I2C slave, equipped with separate transmit (TX) and receive (RX) FIFOs. The slave receives the address and data from the master. It stores received data in the RX FIFO and sends data back through the TX FIFO as needed. The CPU connected to the slave processes the received data, writes it to specific registers, or retrieves data from memory based on the commands received from the master.

Transmit FIFO (TX FIFO): It serves as a buffer for data that needs to be transmitted over the bus. This buffer allows the transmitting device, typically a microcontroller or an FPGA, to load multiple data bytes into the FIFO, which are then sequentially sent out on the I2C bus. The primary advantage of the TX FIFO is that it enables continuous data transmission without the need for constant CPU intervention. This is particularly important in applications requiring high-speed data transfer or in scenarios where the CPU must manage multiple tasks simultaneously. By offloading the immediate need to provide data byte-by-byte to the I2C bus, the TX FIFO improves overall system efficiency and throughput.

Receiver FIFO (RX FIFO): It is a buffer that stores incoming data from the I2C bus, allowing the receiving device to handle incoming data at its own pace. When data arrives on the I2C bus, it is immediately placed into the RX FIFO, ensuring that no data is lost even if the CPU is busy with other tasks. This buffering capability is crucial for maintaining data integrity, especially in environments where data is received in bursts. The RX FIFO reduces the risk of data overflow and enables the CPU to process larger blocks of data, rather than handling each byte individually. This helps in reducing the number of interrupts and the associated processing overhead, leading to more efficient and reliable system performance.

3.2 I2C Write operation

The Figure 3.2 shows the I2C frame to perform write operation from the master to a slave. To write on the I 2C bus, the master will send a start condition on the bus with the slave's address, as well as the last bit (the R/W bit) set to 0, which signifies a write. After the slave sends the acknowledge bit, the master will then send the register address of the register it wishes to write to. The slave will check for the register address and acknowledge again, letting the master know it is ready to store the data to the desired location. After this, the master will start sending the register data to the slave, until the master has sent all the data it needs to (sometimes this is only a single byte), and the master will terminate the transmission with a STOP condition.

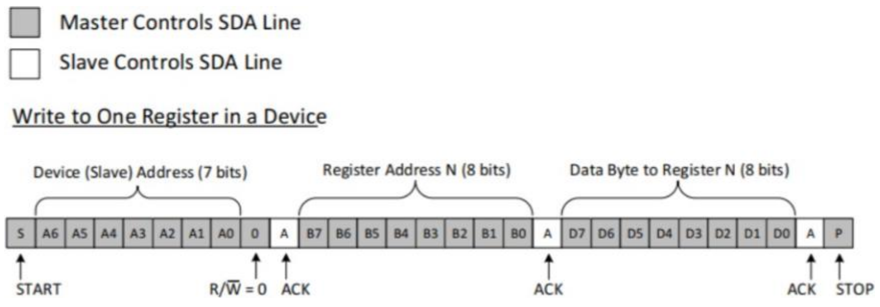
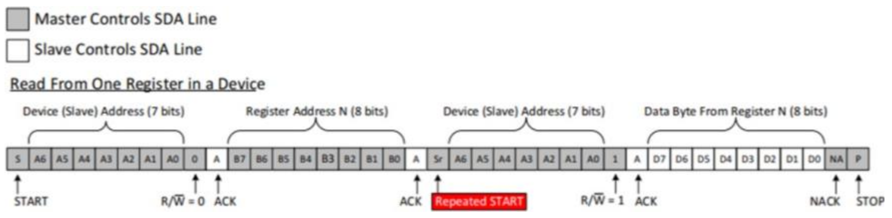

Master Controls SDA Line
Slave Controls SDA Line
Write to One Register in a Device
Device (Slave) Address (7 bits)
Register Address N (8 bits)
Data Byte to Register N (8 bits)
S A6 A5 A4 A3 A2 A1 A0 0 A B7 B6 B5 B4 B3 B2 B1 B0 A D7 D6 D5 D4 D3 D2 D1 D0 A P
START
R/W̄ = 0
ACK
ACK
ACK
STOP

Figure 3.2: i2c write to slave device's register

3.3 I2C Read operation

In order to read from a slave(Figure 3.3), the master will send a START condition, followed by the slave address with the R/W bit set to 1 (signifying a read). The slave will acknowledge the read request, and the master releases the SDA bus, but will continue supplying the clock to the slave. During this part of the transaction, the master will become the master-receiver, and the slave will become the slave-transmitter. The master will continue sending out the clock pulses, but will release the SDA line, so that the slave can transmit data. At the end of every byte of data, the master will send an ACK to the slave, letting the slave know that it is ready for more data. Once the master has received the number of bytes it is expecting, it will send a NACK, signaling to the slave to halt communications and release the bus. The master will follow this up with a STOP condition.



Master Controls SDA Line
Slave Controls SDA Line
Read From One Register in a Device
Device (Slave) Address (7 bits)
Register Address N (8 bits)
Device (Slave) Address (7 bits)
Data Byte From Register N (8 bits)
S A6 A5 A4 A3 A2 A1 A0 0 A B7 B6 B5 B4 B3 B2 B1 B0 A Sr A6 A5 A4 A3 A2 A1 A0 1 A D7 D6 D5 D4 D3 D2 D1 D0 NA P
START
R/W̄ = 0
ACK
ACK
Repeated START
R/W̄ = 1
ACK
NACK
STOP

Figure 3.3: i2c read from slave device's register

3.4 Single/Burst WRRD

Single Write/Read and Burst Write/Read refer to different methods of data transfer between the master and slave devices. These methods dictate how data is written to or read from the slave's registers or memory.

3.5 Single Write/Read

Involves transferring a single byte or a few bytes in a single operation.

Single Write: Involves writing a single byte or a small number of bytes (usually one or two) to the slave device(see Figure 2.9). The master sends the slave address with the write bit, followed by the data byte(s). After receiving each byte, the slave sends an acknowledgment (ACK). The communication can end with a stop condition after the transfer.

Single Read: Involves reading a single byte or a small number of bytes from the slave (see Figure 2.9). The master first writes the slave address with the write bit, sends the register address to be read, and then initiates a repeated start condition. The master then sends the slave address with the read bit, and the slave responds with the requested data byte(s). The master sends an acknowledgment for each byte received and ends the communication with a stop condition after the final byte.

3.6 Burst Write/Read

Allows for continuous writing or reading of multiple bytes, enhancing data transfer efficiency.

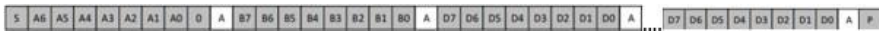


Figure 3.4: Burst Read/Write Frame

Burst Write: Involves writing multiple bytes continuously without needing to send the slave address and write bit for each byte. After the initial byte(s) are written and acknowledged, the master continues to send subsequent data bytes in succession. This mode is efficient for transferring large amounts of data as it reduces overhead from multiple start and stop conditions.

Burst Read: Similar to burst write, but for reading. After specifying the register address, the master reads multiple bytes consecutively from the slave. The master sends the slave address with the read bit and continues to receive data bytes from the slave. Each byte is acknowledged by the master, except the last one, where the master sends a NACK (not acknowledge) to indicate the end of the data transfer, followed by a stop condition.

3.7 I2C Slave Top Level Pinout

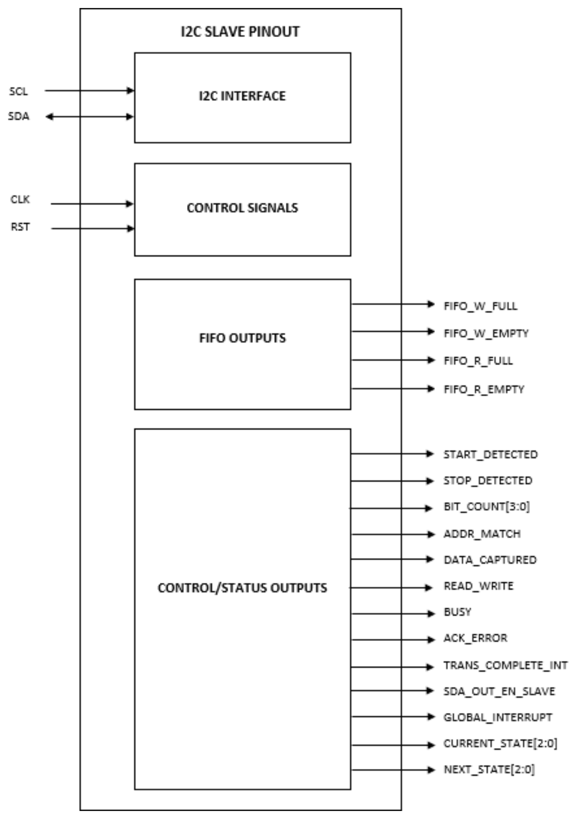


Figure 4.1: I2C Slave Top Level Pinout

The I2C Slave module's top-level pinout and port-map are designed to ensure efficient communication, control, and data handling within the I2C slave module, supporting seamless interaction with the I2C master and other system components. The design includes various interfaces and signal lines in the module's operation.

Bit	Type	Name	Description
1	I2C Interface	SCL	Serial Clock Line is the clock line for I2C communication, controlled by the master device.
1	I2C Interface	SDA	Serial Data Line is the data line for I2C communication, used for sending and receiving data.
1	Control Signal	CLK	Used to provide a clock signal to the I2C slave device, not directly related to the I2C clock.
1	Control Signal	RST	Used to reset the slave device hardware.
1	FIFO Outputs	FIFO-W-FULL	Indicates that the write FIFO buffer is full and cannot accept more data.
1	FIFO Outputs	FIFO-W-EMPTY	Indicates that the write FIFO buffer is empty.
1	FIFO Outputs	FIFO-R-FULL	Indicates that the read FIFO buffer is full.
1	FIFO Outputs	FIFO-R-EMPTY	Indicates that the read FIFO buffer is empty.
1	Control/Status Outputs	START-DETECTED	Indicates that a start condition has been detected on the I2C bus.
1	Control/Status Outputs	STOP-DETECTED	Indicates that a stop condition has been detected on the I2C bus.
[3:0]	Control/Status Outputs	BIT-COUNT	A counter that indicates the number of bits received or transmitted.
1	Control/Status Outputs	ADDR-MATCH	Indicates that the address received matches the slave address.
1	Control/Status Outputs	DATA-CAPTURED	Indicates that data has been successfully captured from the I2C bus.
1	Control/Status Outputs	READ-WRITE	Indicates the operation mode (read or write) following the address match.
1	Control/Status Outputs	BUSY	Indicates that the device is currently busy with an operation.
1	Control/Status Outputs	ACK-ERROR	Indicates an acknowledgment error in transmission, or the completion of a read operation.

1	Control/Status Outputs	TRANS-COMPLETE-INT	An interrupt signal which indicates the completeion of read or write operation.
1	Control/Status Outputs	SDA-OUT-EN-SLAVE	Controls the SDA line output driver, typically used in slave response mechanisms.
1	Control/Status Outputs	GLOBAL-INTERRUPT	A global interrupt signal, possibly used for indicating that an event has occurred.
[2:0]	Control/Status Outputs	CURRENT-STATE	Represents the current state of the state machine controlling the I2C slave device.
[2:0]	Control/Status Outputs	NEXT-STATE	Represents the next state of the state machine.

Table 4.1: I2C Slave Pinout Description

3.8 Slave Block Diagram

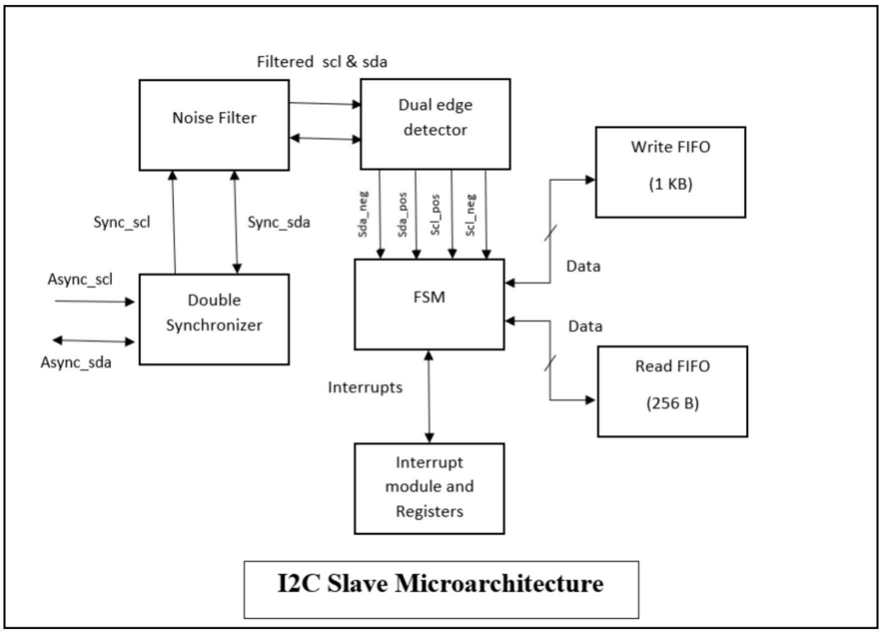


Figure 4.2: Block diagram of I2C slave micro-architecture

3.8.1 Double Synchronizer

The Double Synchronizer block is essential for synchronizing the asynchronous I2C clock (SCL) and data (SDA) lines with the internal clock domain of the I2C slave device. It takes the incoming asynchronous SCL and SDA signals and produces synchronized versions, referred to as Sync-scl and Sync-sda. This synchronization is crucial for ensuring that the internal logic can reliably sample the I2C signals without the risk of metastability, thereby maintaining the integrity of the data and clock signals as they enter the I2C slave module.

3.8.2 Dual Edge Detector

The Dual Edge Detector block is designed to detect both the rising and falling edges of the synchronized SCL and SDA signals. It outputs signals indicating the positive (rising) and negative (falling) edges of these synchronized signals, labeled as Scl-pos, Scl-neg, Sda-pos, and Sda-neg. Detecting these edges is vital for the I2C protocol, as they signify key events such as start and stop conditions, which are necessary for the correct interpretation and timing of data transfer sequences within the I2C communication.

3.8.3 FSM (Finite State Machine)

The FSM, or Finite State Machine, is responsible for managing the state transitions based on the I2C protocol's requirements. This block interprets the edge signals from the Dual Edge Detector along with the current state of the communication to perform necessary actions like address recognition, data reading and writing, and acknowledgment generation. The FSM interfaces with the Write FIFO and Read FIFO to facilitate data transactions between the I2C master and the slave device, ensuring that the data flow adheres to the protocol specifications.

3.8.4 Write FIFO

The Write FIFO block serves as a temporary storage area for data written by the I2C master to the I2C slave. This First-In-First-Out (FIFO) buffer holds the incoming data from the master until the slave's internal logic is ready to process it. By buffering the data, the Write FIFO ensures that the slave can handle varying data rates and process the incoming data efficiently without losing any information.

The size of Write FIFO we used in our design is 1kB.

3.8.5 Read FIFO

The Read FIFO block is designed to temporarily store data that the I2C slave intends to send to the I2C master. This FIFO buffer holds the data that the slave wants to transmit until it is requested by the master. The FSM reads the data from the Read FIFO and manages its transmission to the master, ensuring that the data is delivered in the correct sequence and in compliance with the I2C protocol.

The size of Read FIFO we used in our design is 256B.

3.8.6 Interrupt Module and Registers

The Interrupt Module and Registers block is crucial for handling interrupt generation and providing register access for configuration and status monitoring. This block generates interrupts based on specific events or conditions detected by the FSM, such as data reception, data request, or error occurrences. It also contains registers that can be read or written to for configuring the I2C slave and retrieving status information. These registers allow for flexible control and monitoring of the I2C slave's operation, facilitating smooth and efficient communication within the system

3.9 Clock and Reset Scheme

In an I2C slave design, the clock and reset mechanisms are pivotal for the correct functionality and reliability of the communication process. Here, we'll explore the typical implementation of these mechanisms, drawing on industry practices from Texas Instruments and Intel.

3.9.1 Clock Scheme

3.9.1.1 I2C Clock (scl): The I2C bus operates with an external clock signal (scl), which is provided by the I2C master. This clock is used to synchronize the communication between the master and the slave devices on the bus.

3.9.1.2 System Clock (clk): Internally, the I2C slave uses a system clock (clk) to drive its logic. This clock is typically much faster than the I2C clock to ensure that the slave can process data and respond within the scl cycles. The internal clock manages state transitions, signal processing, and other internal operations.

3.9.2 Reset Scheme

The reset mechanism is essential for initializing the I2C slave to a known state and for recovering from error conditions. The reset signal (rst) typically initializes the internal states and registers.

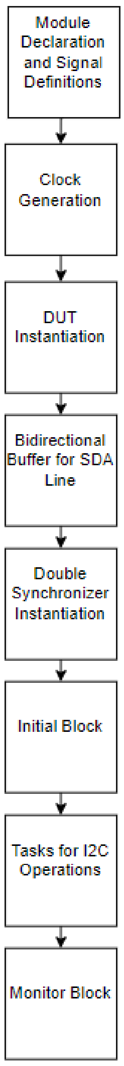
3.9.2.1 Initialization: Upon receiving a reset signal, the I2C slave's internal state machine, registers, and buffers are initialized to their default states. This ensures that the slave starts from a known state and is ready for communication.

3.9.2.2 Error Recovery: In case of communication errors, such as clock stretching issues or bus contention, the reset signal can be used to reinitialize the slave. This involves disabling the I2C peripheral, reconfiguring the I/O pins, and then re-enabling the peripheral.

3.10 Configurable Register Description

The I2C slave device contains several configurable registers to manage its operation and interaction with the master device. The Control Register allows enabling or disabling the device, managing acknowledge responses, and enabling interrupts. The Status Register provides current operational status, including busy, error, and data ready flags. The Address Register sets the I2C slave's address, ensuring it responds to the correct master commands. The Data Register holds the data byte being transmitted or received. Lastly, the Configuration register sets operational parameters such as clock stretching and timeout periods. These registers ensure the I2C slave device can be finely tuned for various applications, enhancing its functionality and reliability.

3.11 Test-bench architecture



Module Declaration and Signal Definitions
Clock Generation
DUT Instantiation
Bidirectional Buffer for SDA Line
Double Synchronizer Instantiation
Initial Block
Tasks for I2C Operations
Monitor Block

Figure 5.1: Testbench architecture

The testbench architecture for the I2C slave module includes the following components:

3.11.1 Module Declaration and Signal Definitions: This section declares the testbench module and defines signals for communication with the Device Under Test (DUT).

3.11.2 Clock Generation: A clock signal is generated to drive the timing of the I2C protocol and other operations in the testbench. DUT Instantiation: The I2C slave module (DUT) is instantiated to be tested against various scenarios.

3.11.3 Bidirectional Buffer for SDA Line: A bidirectional buffer is set up for the SDA line to handle both input and output signals, simulating the actual I2C data line behavior.

3.11.4 Double Synchronizer Instantiation: A double synchronizer is instantiated to ensure reliable data transfer between different clock domains, reducing the risk of metastability.

3.11.5 Initial Block: This block initializes the testbench environment, setting initial values and conditions for the simulation.

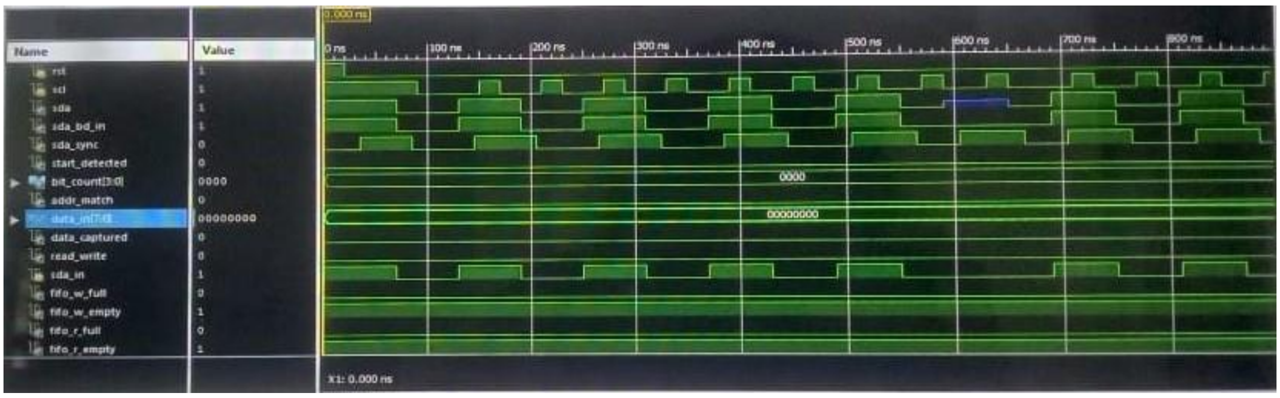
3.11.6 Tasks for I2C Operations: Custom tasks are defined to simulate various I2C operations, such as read, write, and address matching.

3.11.7 Monitor Block: This block observes the outputs and behavior of the DUT during the simulation, checking for correctness and logging results.

CHAPTER 4

RESULTS OF THE PROJECT

6.1 Simulation



Name	Value
rst	1
scl	1
sda	1
sda_bd_in	1
sda_sync	0
start_detected	0
bit_count[3:0]	0000
addr_match	0
data_in[7:0]	00000000
data_captured	0
read_write	0
sda_in	1
fifo_w_full	0
fifo_w_empty	1
fifo_r_full	0
fifo_r_empty	1

Figure 6.1: Simulation of I2C slave

The simulation results from the Xilinx Vivado tool demonstrate the correct functionality of the I2C slave mod- ule, including handling start and stop conditions, byte reception, and data transfer completion. Key signals such as 'start-detected' and 'addr-match' confirm the detection of I2C bus activities and address matching, while 'data-captured' indicates successful data reception. The state transitions of the FSM align with these operations, as shown by the 'current-state' and 'next-state' signals. Additionally, FIFO status signals like 'fifo-w-empty' and 'fifo-r-empty' monitor the buffer states, and the 'trans-complete-int' signal, along with the 'global-interrupt,' indicates the completion of data transfers. Overall, the waveform confirms the module's compliance with the I2C protocol, ensuring proper synchronization, data handling, and interrupt management.

6.2 CONCLUSION

In conclusion, this mini-project successfully demonstrates the design of an I2C slave microarchitecture us- ing Verilog, showcasing the fundamental aspects and practical applications of the I2C protocol. The project highlights the efficient data transfer capabilities of I2C, facilitated by its two-wire communication system com- prising the Serial Data Line (SDA) and Serial Clock Line (SCL). Through the development of a single master- single slave setup, we explored the critical functions of the I2C protocol, including device addressing, data synchronization, and the bidirectional exchange of data. The implementation provided valuable insights into the complexities of designing a reliable communication interface, essential for integrating multiple devices in a system. Overall, this project underscores the versatility and simplicity of the I2C protocol, making it a pre- ferred choice for inter-IC communication in various digital design applications. The successful completion of this project demonstrates the feasibility of using Verilog for implementing robust I2C interfaces, paving the way for more advanced and scalable communication solutions in future projects.

REFERENCES

[1] https://www.nxp.com/docs/en/userguide/UM10204.pdf?_gl=1*xy14su*_ga*MjIxMDEzMzM3LjE3MjE5MDU0NzI.*_ga_WM5LE0KMSH*MTcyMTkwNTQ3MS4xLjEuMTcyMTkwNTU2Mi4wLjAuMA..

[2] https://digilent.com/reference/microprocessor/basys-mx3/unit-4-lab4d/start

[3] https://www.bing.com/ck/a?!&&p=0b7963b265d10cd1JmltdHM9MTcyMTg2NTYwMCZpZ3VpZD0zOTVkZjJlYy04OThjLTY4ZDQtM2Q3Yi11NjVhODgxNzY5ODImaW5zaWQ9NTM3NQ&ptn=3&ver=2&hsh=3&fclid=395df2ec-898c-68d4-3d7b-e65a88176982&psq=I2C+PROTOCOL+IMPLEMENTATION+ON+VERILOG&u=a1aHR0cHM6Ly93d3cucmVzZWFyY2hnYXRlLm5ldC9wdWJsaWNhdGlvbi8yNzU3NzEzMzNfRGVzaWduX29mX0kyQ19TaW5nbGVfTWFzdGVyX1VzaW5nX1Zlcmlsb2c&ntb=1