A DESIGN UNDER TEST IN VERILOG, TESTBENCH ENVIRONMENT FOR I2C PROTOCOL WITH ASSERTION AND A PERL SCRIPT FOR I2C BUS LOG ANALYZER

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

B. Tech

in

Electronics and Communication Engineering

By

PRANES S (108121091)

ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTEOF TECHNOLOGY

TIRUCHIRAPALLI-620015

MAY 2025

A DESIGN UNDER TEST IN VERILOG, TESTBENCH ENVIRONMENT FOR I2C PROTOCOL WITH ASSERTION AND A PERL SCRIPT FOR I2C BUS LOG ANALYZER

A thesis submitted in partial fulfillment of the requirements for the award of the degree of

B. Tech

in

Electronics and Communication Engineering

By

PRANES S (108121091)

ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTEOF TECHNOLOGY

TIRUCHIRAPALLI-620015

MAY 2025

BONAFIDE CERTIFICATE

This is to certify that the project titled '**A Design under test in Verilog, create a testbench environment for I2C protocol with assertion and a Perl script for I2C bus log analyzer**' is a bonafide record of the work done by

Pranes S (108121091)

in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Electronics and Communication Engineering** of the **NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI,** during the year 2024-2025.

Dr. Srinivasulu Jogi
Guide

Dr. M.Bhaskar
Head of the Department

Project Viva-voce held on ________________________________

Internal Examiner

External Examiner

ABSTRACT

In modern digital systems, verifying communication protocols like I2C (Inter-Integrated Circuit) is essential for ensuring robust and reliable data exchange between integrated components. This project focuses on developing a Design Under Test (DUT) for an I2C-based communication system using Verilog, and constructing a self-checking testbench environment that includes assertions and a Perl-based I2C bus log analyser. Additionally, the potential intersection with SERDES (Serializer/Deserializer) in high-speed data environments is briefly explored to highlight broader applications.

The DUT comprises two key modules: an I2C Master and an I2C Slave, implemented in Verilog. The Master module initiates communication, controls clock generation, and handles read/write operations based on input stimuli, while the Slave responds accordingly based on addressing and control signals. To simulate real-world bus behaviour, open-drain characteristics of I2C are modelled using pullup primitives on the shared SDA and SCL lines.

The testbench is developed using System Verilog and incorporates UVM-inspired verification principles. It includes interfaces, randomized stimulus generation, scoreboard comparison, and synchronizing logic. A key aspect of the environment is the integration of System Verilog assertions to enforce protocol correctness, such as START and STOP condition checks, SDA/SCL toggling, and acknowledgment behaviour. Both SDA (Serial Data Line) and SCL (Serial Clock Line) are bidirectional that provides a simple and efficient communication between devices.

For post-simulation analysis, a Perl script is written to parse simulation logs and extract I2C transactions. The script identifies start/stop conditions, byte-level data transfer, ACK/NACK signalling, and highlights protocol violations. This enhances debug efficiency and acts as a lightweight transaction-level monitor without GUI-based tools.

Though I2C is inherently low-speed, the discussion extends into how SERDES blocks, typically used in high-speed serial links, can be involved. In advanced SoCs, SERDES can carry I2C-like control signals in embedded management channels or configuration buses, especially when aggregating multiple protocol streams onto high-speed serial links. Integrating protocol verification, assertion-based checking, and log analysis in such mixed-speed environments becomes crucial.

This work demonstrates a comprehensive and scalable approach to I2C protocol verification using Verilog, System Verilog assertions, and Perl scripting. It showcases how verification environments can be extended beyond simulation to include assertion-based checks and automated log analysis, while also being adaptable for integration with SERDES-based interconnects in complex chip architectures.

Keywords: SERDES, I2C, UVM, PERL

ACKNOWLEDGEMENT

I would like to express our heartfelt gratitude to all those who have supported and guided us throughout the course of this project. Without their valuable input, encouragement, and assistance, the successful completion of this work would not have been possible.

First and foremost, we extend our sincere thanks to **Dr. Srinivasulu Jogi**, my project guide, for his constant support, insightful feedback, and guidance throughout the project. His mentorship was instrumental in shaping our ideas and steering our efforts in the right direction.

My sincere thanks to acknowledge the support given by the Head of the Electronics and Communication Engineering Department, **Dr. M. Bhaskar**, who encouraged my efforts. I would also like to express my deep gratitude to my faculty members, and lab assistants for their invaluable guidance and technical support throughout the duration of this project.

We would also like to acknowledge our internal reviewers—**Dr. G. Lakshminarayanan**, and **Dr. V. Sudha**—for their valuable suggestions and constructive feedback during the review sessions, which helped us refine and improve our project.

My sincere appreciation also goes to the **Department of Electronics and Communication Engineering, NIT Trichy**, and the ECE Lab facilities for providing us with the resources and environment necessary to carry out our research and development activities effectively.

Last but not least, we thank our parents and friends for their unwavering encouragement, emotional support, and belief in us throughout this journey.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 General Motivation

I²C (Inter-Integrated Circuit) is a widely used synchronous, multi-master, multi-slave, two-wire serial communication protocol. Developed by Philips (now NXP), I²C facilitates efficient short-distance communication between components like microcontrollers, EEPROMs, sensors, and peripherals on a single PCB. It operates using just two signals: SDA (Serial Data Line) and SCL (Serial Clock Line), making it ideal for low-complexity interconnects.

In the field of VLSI verification engineering, I²C protocol plays a significant role during the design verification of SoCs and IPs that integrate embedded communication blocks[1]. I²C controllers often serve as configuration or control interfaces for peripheral IPs such as ADCs, temperature sensors, or PMICs within a chip. As a verification engineer, it becomes crucial to validate both functional correctness and protocol compliance of these I²C components.

To ensure this, System Verilog testbenches are created that mimic real-world I²C master or slave devices, generate random transactions, monitor response behaviour, and flag protocol violations. The use of System Verilog assertions (SVA) enhances the reliability of this process by checking for conditions like valid START and STOP sequences, correct clock-to-data relationships, and proper acknowledgment handling. These assertions help catch corner-case bugs early in simulation[2].

To complement simulation-based checks, Perl scripting is often used in the verification workflow for post-processing simulation logs. Perl is well-suited for parsing large log files, identifying transaction boundaries, extracting payloads, flagging anomalies, and generating summarized reports. For instance, a Perl script can automatically analyse a waveform log to find whether a STOP condition occurred after every complete data transaction or if NACK was sent after an incorrect address. This allows verification engineers to automate validation and debugging without manually inspecting waveforms.

Furthermore, in advanced chip designs, I²C signals might be routed over SERDES (Serializer/Deserializer) lanes in compact, high-speed environments[3]. This makes protocol verification more complex, as control signals may be serialized along with other traffic. Nevertheless, fundamental I²C compliance must still be ensured, and the verification principles remain applicable even in these high-speed.

1.2 My Contributions

- To learn about System Verilog, UVM (Universal Verification Methodology), Perl scripting language
- Write Verilog Code for master and slave of I2C and testbench code in System Verilog
- Create a Perl script for I2C bus log analyzer
- Code Coverage for a proc in Synopsys

CHAPTER 2

LITERATURE REVIEW

2.1 RELATED WORKS

2.1.1 Design of High-Speed I²C Interface Protocol Design and Its Verification Using System Verilog [1]

This paper presents the development of a high-speed I²C protocol interface and its verification using System Verilog. The authors focus on enhancing data transfer rates while maintaining low wiring complexity, making the protocol suitable for efficient communication between master and slave devices. The design includes modules for generating start and stop conditions, address recognition, read/write operations, and acknowledgment handling. Verification is performed using System Verilog, ensuring the reliability and correctness of the protocol implementation. The study demonstrates that the high-speed I²C interface is effective for applications requiring rapid and reliable data exchange.

2.2.2 Design and Implementation of I²C Communication Protocol on FPGA for EEPROM [2]

This presents the development of an I²C master controller implemented on an FPGA using VHDL, aimed at enabling reliable communication with EEPROM memory devices. The I²C protocol, known for its simplicity and efficiency using just two lines (SDA and SCL), is ideal for connecting multiple devices with minimal wiring. The authors designed modules to handle start/stop conditions, address matching, data transfer, and acknowledgment signaling in accordance with the I²C standard. Their design was tested through simulation and successfully verified for correct functionality and timing. The implementation was shown to be resource-efficient on FPGA and scalable for broader applications. Overall, the work demonstrates how hardware-based protocol implementation can support robust and efficient memory communication in embedded systems.

2.2.3 Design of I2C Protocol[3]

This paper presents an in-depth exploration of the I²C communication protocol, focusing on its effectiveness for low-speed, short-distance communication between integrated circuits. The authors detail the core principles of I²C, including its two-wire design using the SDA (Serial Data Line) and SCL (Serial Clock Line), and explain how it facilitates synchronized communication between a master and multiple slaves. Important protocol features like the start and stop conditions, acknowledgment signals, addressing schemes (including 7-bit and 10-bit modes)[4], and arbitration are described clearly. A key highlight of the paper is the practical implementation of the I²C protocol using Verilog HDL, where the authors employ finite state machines to model communication states and ensure proper data handling. This implementation supports efficient, low-resource hardware design suitable for embedded systems and VLSI applications[5]. The paper concludes that the I²C protocol, owing to its simplicity and scalability, remains highly applicable in modern electronics and SoC architectures, and the Verilog-based design approach provides a valuable framework[6] for designers looking to integrate I²C in their hardware projects.

2.2.3 Design and Verification of I2C Protocol

A paper [7] presents a comprehensive design of the I²C protocol with various features, including combined messages, multiple addressing modes (7-bit and 10-bit), different data patterns, and clock frequencies. The implementation involves FPGA acting as the master and a test card as the slave. The study also includes stress testing by randomizing features and running the system for extended periods to ensure reliability. This research [8] focuses on implementing the I²C protocol using Verilog HDL. It details the design of an I²C master controller capable of generating start and stop conditions, handling read/write operations, and managing acknowledgments. The paper emphasizes the protocol's simplicity and efficiency, making it suitable for communication between multiple devices with minimal wiring[9].

CHAPTER 3

METHODOLOGY

3.1Master_Module

This Verilog module models an I²C master controller that communicates with an I²C slave using standard protocol sequences. It includes inputs such as clock (clk), read/write control (rd_wr), start and stop control, a reset signal, a 7-bit slave address, and 8-bit input data. The outputs include an 8-bit data output (dout) and open-drain lines for the I²C clock (SCL) and data (SDA). The master controls both SCL and SDA lines to initiate communication, transmit the slave address, read or write data, and properly terminate the session[1]. The bidirectional nature of SDA is handled using conditional assignments to support reading acknowledgments and data from the slave.

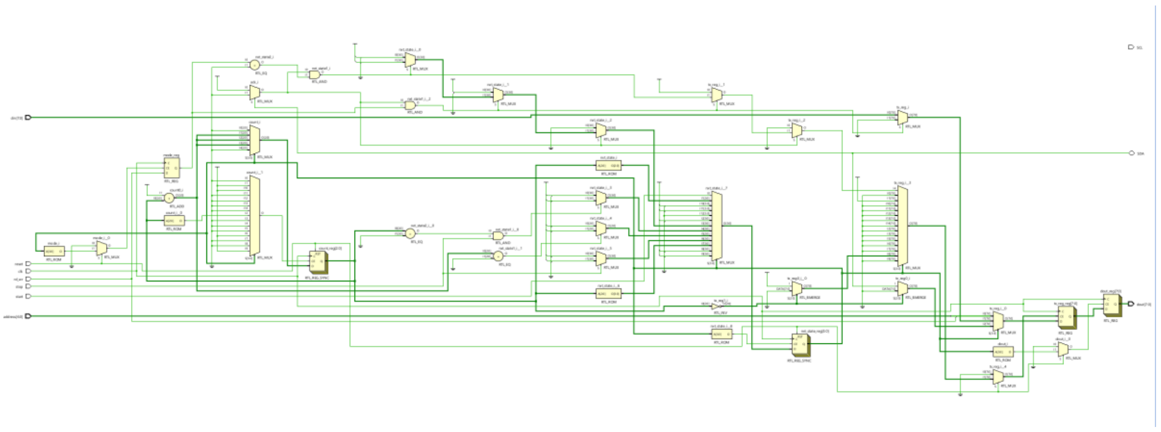


Fig. 3.1. Overall Schematic of I2C Master

3.1.1 State Machine Design

The I²C master logic is governed by a finite state machine (FSM) with 15 defined states, from S0 (idle) to S14 (stop)[1]. These states manage the sequential flow of the protocol. The FSM begins in the idle state and waits for a high signal on the start input. When this occurs, the FSM transitions through states to generate the start condition, transmit the address and R/W bit, check for acknowledgment, transfer or receive data, and finally generate the stop condition. This clear state separation makes the design structured and easier to manage for both write and read transactions.

3.1.2 Start and Address Phases

The communication begins in state S0 when the master is idle. Upon receiving a high start signal, the FSM moves to S1, where it pulls SDA low while SCL remains high to create a valid I²C start condition[2]. Then, in S1 to S3, the master serially transmits the 7-bit address concatenated with the R/W bit through the SDA line. This is done by shifting out bits from a transmission register (tx_reg) and toggling the SCL line appropriately to latch the bits.

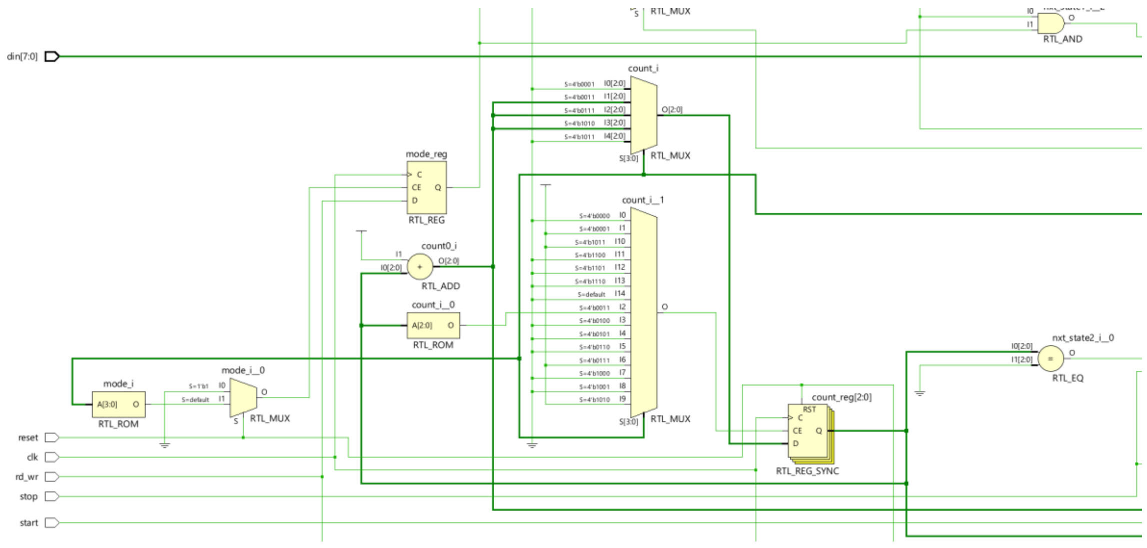


Fig. 3.2. I2C Master's Input side

3.1.3 Acknowledge Phase

Once the address and R/W bit are transmitted, the FSM moves into the acknowledgment phase (S4 and S5)[3]. The master releases the SDA line and waits for the slave to pull it low, indicating an ACK. If the slave sends a NACK (SDA remains high), the master resets to the idle state. If ACK is received, the FSM proceeds to either the transmit (write) or receive (read) phase based on the value of the rd_wr signal.

3.1.4 Write Operation

In write mode (rd_wr = 0), the FSM enters states S6 and S7[1], where the 8-bit data from the din input is loaded into tx_reg and sent to the slave, bit by bit, on the SDA line. As each bit is placed on SDA during the falling edge of SCL, the slave samples it on the rising edge. After transmitting 8 bits, the FSM checks for ACK and either sends more data or concludes with a stop condition if the stop signal is high.

3.1.5 Read Operation

In read mode (rd_wr = 1), the FSM proceeds to states S8 through S12. The master releases the SDA line (sets it to 1) and begins reading one bit at a time from the slave on each rising edge of SCL. These bits are stored into a temporary register (tx_reg), and after 8 bits, the register value is assigned to dout. The master then sends an ACK and prepares for the next byte or finishes the transaction if the stop signal is asserted.

3.1.6 Stop Condition

The FSM enters states S13 and S14 to generate a proper stop condition. In S13, the SCL line is driven high, and in S14, the SDA line is released to go high. This sequence (SDA going high while SCL is high) is recognized by the slave as a stop condition, ending the communication session and returning the FSM to the idle state S0.

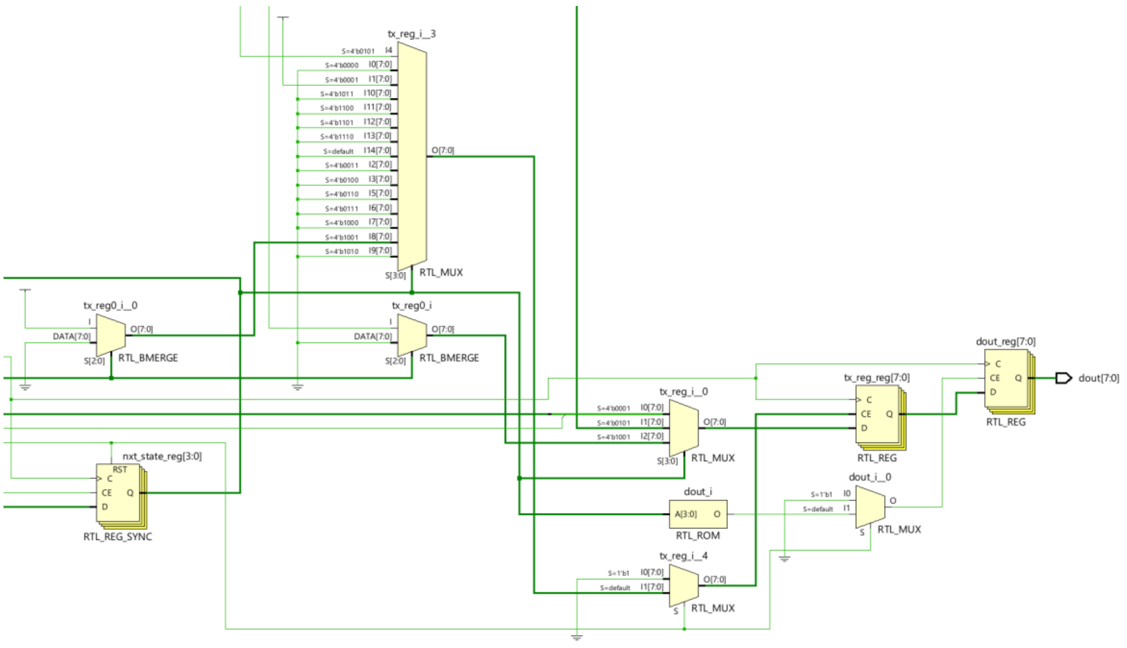


Fig. 3.3. I2C Master's Output side

3.1.7 Open-Drain Handling for SDA and SCL

Since I²C buses operate on open-drain (wired-AND) signaling, the SDA and SCL lines are handled using conditional assignments: assign SDA = (sda == 0) ? 0 : 1'bz and similarly for SCL. This ensures the master only pulls the line low when necessary and otherwise leaves it floating ('z)[3] so that other devices can drive it. Pull-up resistors (external or simulated with pullup) ensure the line returns to high when not driven.

3.2 Slave_Module

The i2c_slave module represents an I²C slave device that can receive or transmit data based on the master's command. It interacts with the SDA (bidirectional data line) and SCL (clock line), and it uses a reset signal to initialize its operation. The slave receives an address and data (din) from the system and responds with output data (dout) after receiving the correct read/write command from the I²C master[2]. Internally, it uses a finite state machine (FSM), a data register, and control signals such as ACK, mode, and count to manage protocol operations.

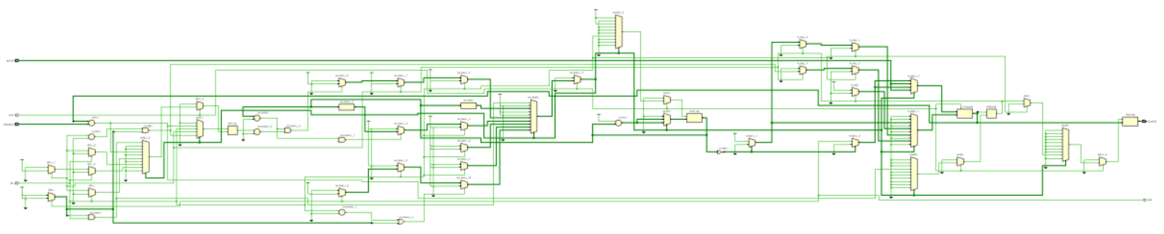


Fig. 3.4. Overall Schematic of I2C Slave

3.2.1 Start Detection and State Machine Initialization

The FSM starts in S0, where it waits for a valid START condition, which is indicated by SDA going low while SCL is high. This is detected using an additional signal sda_sense, which gates the sensitivity of the FSM to avoid unnecessary triggering. Upon detecting a START[4], the slave enters S1 and initializes counters and flags. The FSM uses 10 states (S0 to S9) to handle the full I²C transaction, transitioning between them based on SCL, SDA, and internal logic.

3.2.2 Address Recognition

In states S1 and S2, the slave shifts in 8 bits of data from the SDA line, storing them in tx_reg. These bits represent the 7-bit address and 1-bit R/W command sent by the master. In S3, the module compares the received address (top 7 bits of tx_reg) with its own. If matched, the slave acknowledges by pulling SDA low (sda = 0) and determines the operation mode (read or write) based on the LSB of tx_reg.

3.2.3 Read/Write Data Phase

In S4 and S5, the slave either prepares to send data (din) if in transmit mode, or prepares to receive data if in receive mode. The FSM uses count to keep track of bits transferred. In transmit mode, data bits are placed on the SDA line during SCL = 0. In receive mode, it keeps SDA high and reads bits into tx_reg on SCL = 1. The ACK flag tracks successful byte reception or transmission.

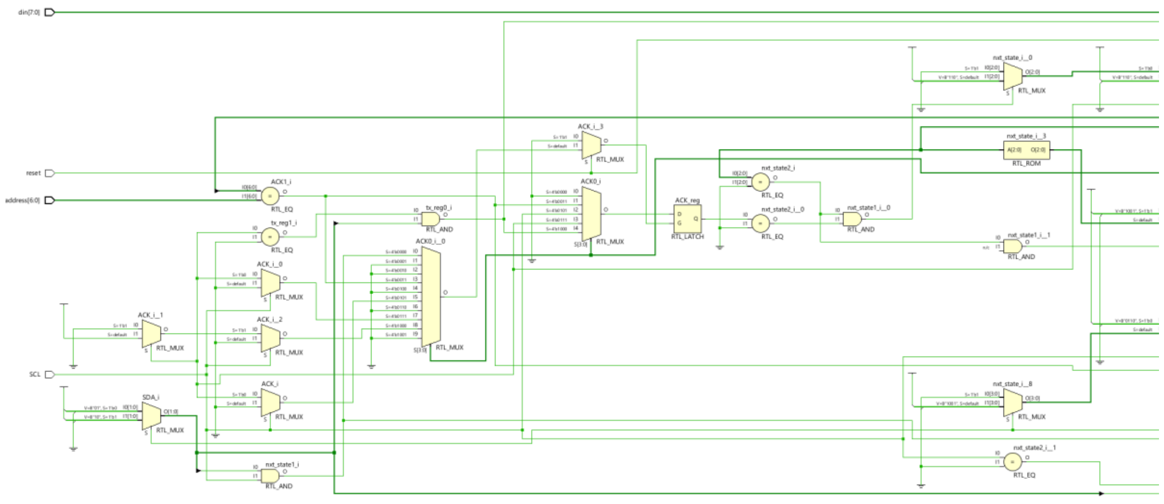


Fig. 3.5. I2C Slave's Input side

3.2.4 Data Acknowledgment and Latching

In S6 and S7, the slave handles acknowledgment and processes the received byte. If in receive mode, once 8 bits are received, the slave pulls SDA low to ACK the reception. The received byte is latched to dout in state S8. If in transmit mode and the master ACKs the sent byte, the slave prepares the next data byte. If the master doesn't ACK, the slave resets or waits for a STOP.

3.2.5 Stop Condition and Recovery

The STOP condition is detected in state S9, where SDA goes high while SCL is high. If STOP is detected, the FSM resets to S0 and sets sda_sense = 1, preparing to detect the next START condition. If not, and SCL = 0, the FSM continues to the data transfer state (S6)[6] depending on the mode. The FSM is also designed to handle repeated starts.

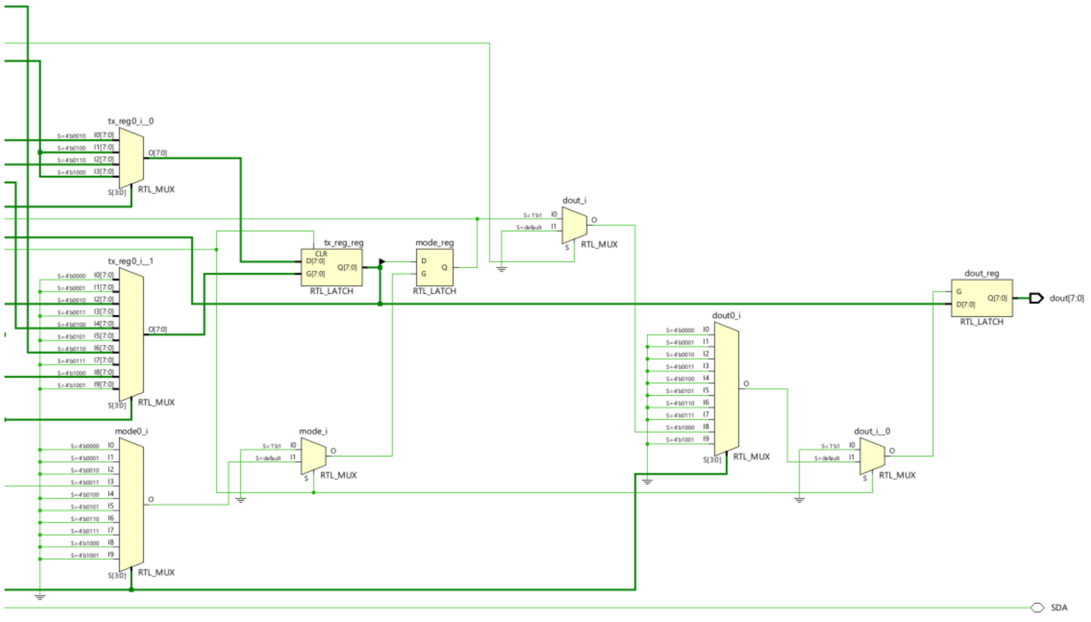


Fig. 3.6. I2C Slave's Output side

3.2.6 Open-Drain Output Handling

Since the I²C protocol requires open-drain (wired-AND) signalling for the SDA line, the module uses a conditional assign: assign SDA = (sda == 1) ? 1'bz : 1'b0;.[7] This ensures the slave only pulls the line low when needed (ACK, data transmit), and leaves it in high-impedance otherwise, allowing proper bus sharing.

3.3 Verification Environment

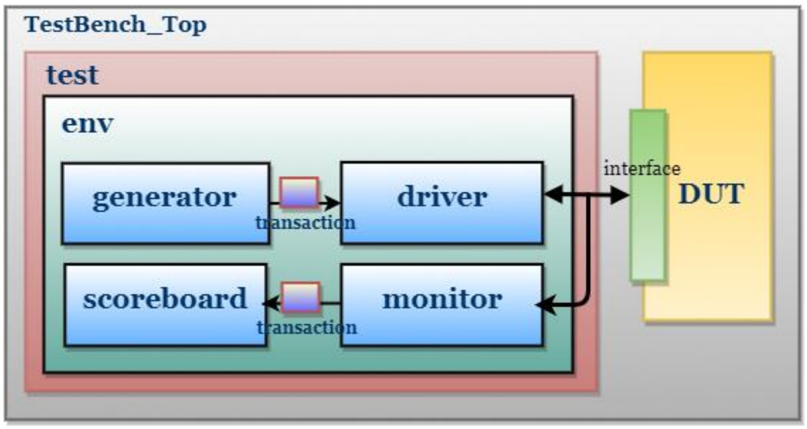


Fig. 3.7. Testbench Architecture

These are the main Components of System Verilog

3.3.1 packet (transaction)

The packet class in System Verilog models a data transaction for I^{2}C verification. It contains randomized fields like address, data, and mode to simulate various inputs, while an error flag is used to detect mismatches during simulation. The display() task prints the packet contents, and the compare() task checks the received packet against an expected one, reporting any mismatch. This class plays a key role in testbenches by enabling structured stimulus generation and result validation, which is essential for functional verification.

3.3.2 Generator

The generator module plays a vital role in producing randomized I^{2}C protocol stimuli by generating address, mode (read/write), and data signals, encapsulated within a packet class object. These signals are randomized using the randomize() method and then propagated to the master and slave drivers. The address signal defines the target I^{2}C device, mode specifies whether the operation is a read (1) or write (0), and data carries the actual 8-bit payload[8]. This randomized generation ensures diverse test coverage, allowing the verification environment to rigorously test different I^{2}C communication sequences and uncover edge-case failures.

3.3.3 Driver

This SystemVerilog code defines two classes—master_driver and slave_driver—that model the behavior of I^{2}C master and slave components in a UVM-like verification environment. Each driver interfaces with its corresponding virtual interface (master_interface or slave_interface) and communicates with a scoreboard through a mailbox. The master_driver class handles generating randomized transactions (read or write), toggling the start and stop signals appropriately, and sampling data at specific clock edges. It supports both masters transmit and receive operations. Similarly, the slave_driver class responds to master transactions based on the mode; in receiver mode, it samples data from the master, and in transmitter mode, it generates random data to send back. Both classes utilize clock-based synchronization to model protocol timing and ensure accurate data exchange during testbench simulation.

3.3.4 Interface

This module has the declaration of all the signals of master and slave. All the datatypes should be of logic datatype.

3.3.5 Monitor

The monitor component in the context of the I²C verification environment plays a passive yet critical role by observing signal activity on the bus without driving or altering any values. It captures transactions from the master and slave interfaces, extracting relevant data such as address, mode (read/write), and data bytes, and sends this information to the scoreboard for comparison[9]. The monitor ensures protocol compliance and enables functional coverage collection by analyzing actual communication patterns, helping to detect mismatches or violations when compared to the expected behavior defined in the testbench.

3.3.6 Scoreboard

This System Verilog scoreboard class is used in the testbench to compare the data exchanged between the I²C master and slave during simulation. It uses two mailboxes (sdrv2sb and mdrv2sb) to receive data from the slave driver and master driver respectively. The start task runs for a given number of bytes, retrieves the transmitted and received values, and compares them. If the values match, a "PASS" message is displayed; otherwise, an error is counted and a "FAIL" message is shown[9]. This helps in validating functional correctness of the data transfer. The class also includes commented-out code for an assertion to check correct start condition behavior, showing how assertions can be integrated to monitor protocol-level requirements.

3.3.7 Testcase

This System Verilog testcase program sets up and runs a test scenario for verifying an I²C communication system using the environment class. It begins by creating an instance of the environment with master and slave interfaces (mintf and sintf). The test sequence includes building the environment, applying a reset, and then initiating data transactions using the start() method with a specified number of bytes (5 in this case). The code structure utilizes fork...join to allow concurrent execution of the test stimulus. After the first transaction, the test mistakenly attempts to rebuild the environment mid-simulation (env.build()), which may cause simulation errors due to reinitialization. Following this, another reset is applied and a second transaction is run. The test ends after a delay using $finish. This testbench mimics a realistic verification flow for validating read and write operations over an I²C protocol.

3.3.8 Assertion

The assertions module defines a comprehensive set of System Verilog assertions to validate key I²C protocol rules and ensure signal correctness throughout communication. These assertions monitor events such as START and STOP conditions[11], signal stability, and acknowledgment behaviour. The module uses the property-assert constructs to catch violations and report meaningful error messages during simulation. Each property enforces a specific protocol requirement, helping ensure reliable bus behaviour and proper master-slave interactions.

Here are the assertions implemented:

- **sda_changes_when_scl_low**: Ensures SDA only changes when SCL is low (except for START/STOP).
- **start_condition**: Confirms that SDA falls when SCL is high during a START.
- **stop_condition**: Checks that SDA rises when SCL is high during a STOP.
- **bus_idle_after_reset**: Ensures the bus is idle (SCL and SDA high) after reset.
- **address_stable**: Verifies that the address remains stable after a START condition.
- **rd_wr_valid_after_start**: Ensures a valid read/write signal is present shortly after START.
- **data_stable_when_scl_high**: Makes sure din is stable when SCL is high.
- **dout_update_after_read**: Ensures dout gets updated properly after a read operation.
- **no_multiple_starts**: Prevents issuing multiple STARTs without a STOP in between.

3.4 I2C Format

3.4.1 Start Condition

The start condition marks the beginning of communication on the I²C bus. It is generated by the master by pulling the SDA line low while the SCL line remains high. This specific transition is recognized by all devices connected to the bus, signalling them to listen for a potential address match. Without a valid start condition, no device on the bus will respond. The two types of addressing mode are present in the I2C bus, a 7-bit addressing mode and 10-bit addressing mode. In 7-bit addressing mode 128 devices can be connected to the bus and in 10-bit addressing mode 1024 devices can be connected to the bus[12].

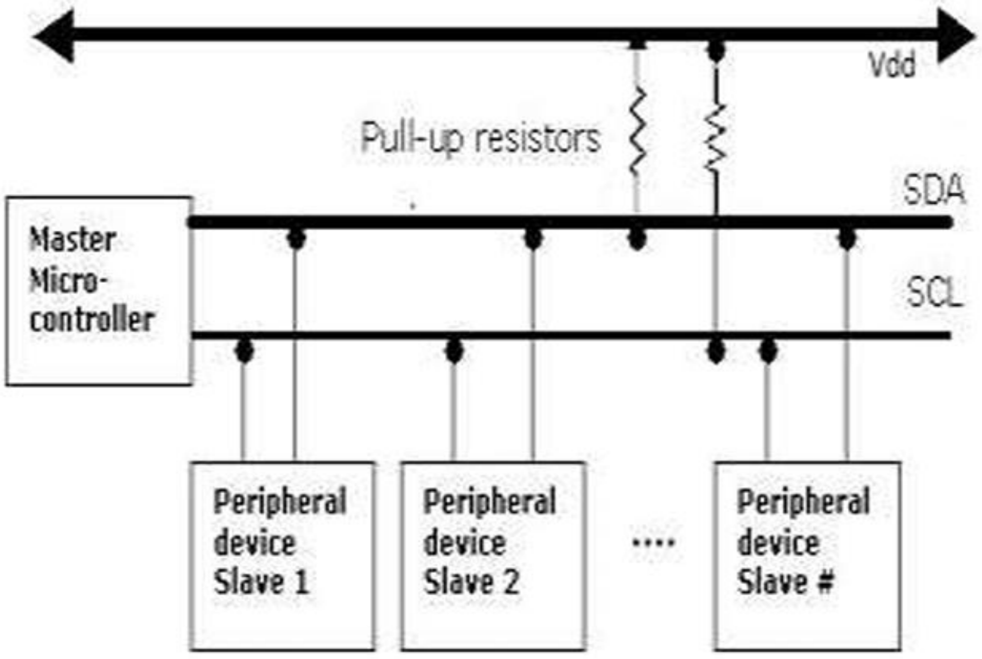

Vdd
Pull-up resistors
SDA
SCL
Master Micro-controller
Peripheral device Slave 1
Peripheral device Slave 2
....
Peripheral device Slave #

Fig. 3.8 I2C Master Slave Configuration with Pull up Resistors [3]

3.4.2 Address Phase

After the start condition, the master sends a 7-bit slave address followed by an R/W bit, making it an 8-bit address frame[12]. The R/W bit determines the direction of the data transfer:

- 0 indicates a write operation (master to slave)
- 1 indicates a read operation (slave to master)

All slaves receive this frame, but only the one whose address matches responds. Addressing is critical as it selects which slave the master will communicate with.

3.4.3 Acknowledgment (ACK/NACK)

Once a slave detects its address, it sends an ACK (acknowledge) bit to the master by pulling the SDA line low during the ninth clock pulse. If no device matches the address, the SDA line stays high, resulting in a NACK[13] (not acknowledged). Similarly, after every 8 bits of data, the receiver (slave or master, depending on direction) sends either an ACK (for successful reception) or NACK (to indicate an error or end of transmission).

3.4.4 Data Transfer

Data transmission happens one byte at a time (8 bits per byte). The master provides the clock pulses on SCL, and data is transferred over the SDA line, most significant bit (MSB) first.

- In a write operation, the master sends data to the slave. The slave acknowledges each byte with an ACK.
- In a read operation, the slave sends data to the master. After each byte, the master sends an ACK if it wants more data or a NACK if it's done.

This handshake ensures data is exchanged reliably.

3.4.5 Repeated Start Condition

Sometimes, the master might want to communicate again without releasing the bus, for example, to switch from writing to reading. In such cases, the master issues a Repeated START condition instead of a STOP. This is identical in waveform to a regular start condition and allows uninterrupted communication on the bus.

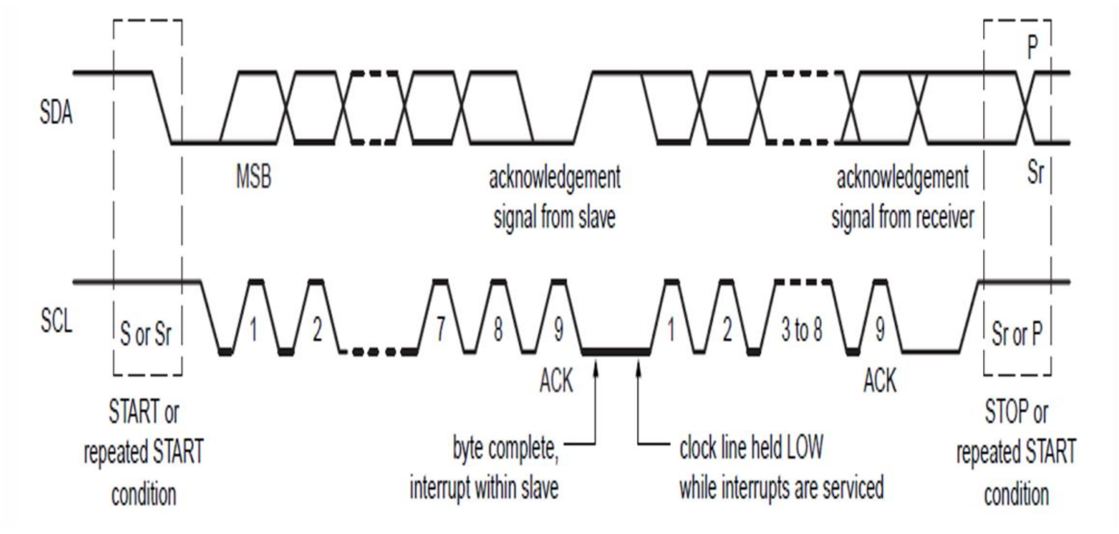

SDA
SCL
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
P
Sr
S or Sr
1
2
7
8
9
ACK
1
2
3 to 8
9
ACK
Sr or P
START or repeated START condition
byte complete, interrupt within slave
clock line held LOW while interrupts are serviced
STOP or repeated START condition

Fig. 3.9 Format of I2C Protocol [7]

3.4.6 Stop Condition

Once all data has been exchanged, the master sends a STOP condition to end the communication. This is generated by releasing the SDA line from low to high while SCL is high. This transition informs all devices that the bus is now free, allowing other masters (if any) to start communication. In the case of a read operation, the master must first send a NACK after the final byte before issuing the stop.'

3.4.7 Arbitration and Clock Stretching

In multi-master systems:

- **Arbitration** ensures only one master controls the bus. If two masters try to communicate simultaneously, the one that sees a mismatch between the SDA value it sent and the one on the line (i.e., it wrote a '1' but sees a '0') backs off, allowing the other to continue[15].
- **Clock stretching** allows a slave to slow down communication by pulling SCL low. The master must wait until SCL is released before continuing. This is useful for slower devices that need more processing time.

CHAPTER 4

SIMULATION RESULTS

4.1 EDA PLAYGROUND SIMULATION OUTPUT

4.1.1 Master's Simulation Results

In the *Fig.4.1,* we observe the I2C master's behavior during simulation. The SCL and SDA lines show standard I2C signaling, with SDA toggling relative to SCL for valid data transmission. The master initiates communication when start is asserted, and drives a valid 7-bit address (0x07, then 0x72) onto the bus. The mode signal determines the direction (read/write), with mode = 0 indicating read in this case. Data is sequentially transmitted over din and visible in the tx_reg, while the count tracks the number of bits sent. The master releases control after asserting the stop signal, indicating the end of transmission. The ack_r signal is asserted once the data is received correctly from the slave. This confirms correct master-side protocol behavior in the simulation.

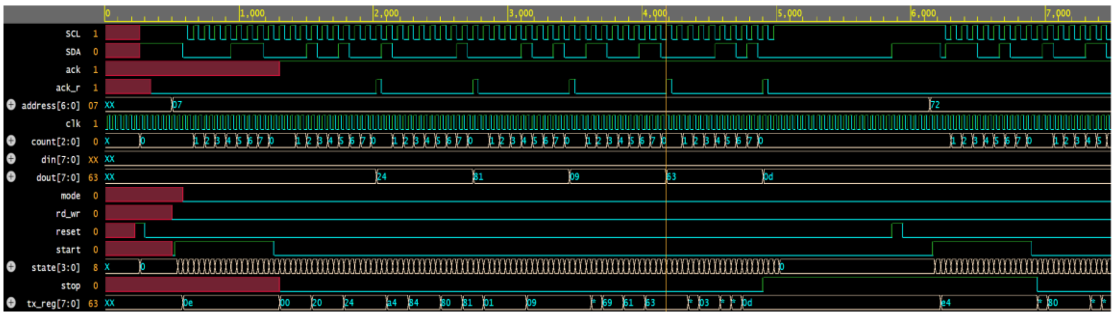


Fig.4.1. Waveform showing the correct working of I2C Master

4.1.2 Slave's Simulation Results

In the *Fig.4.2,* We can see the slave correctly responds to the master's I2C transactions. The slave monitors the SCL and SDA lines and decodes the incoming address (0x07 and later 0x72). When a matching address is detected and mode = 0 (Read operation), the slave captures data on dout (0x24, 0x81, 0x09 etc.) as it's received over the bus. The count signal tracks bit positions, and data is assembled in tx_reg before being driven to the internal logic. Overall, the slave demonstrates proper data reception and protocol compliance throughout the transaction.

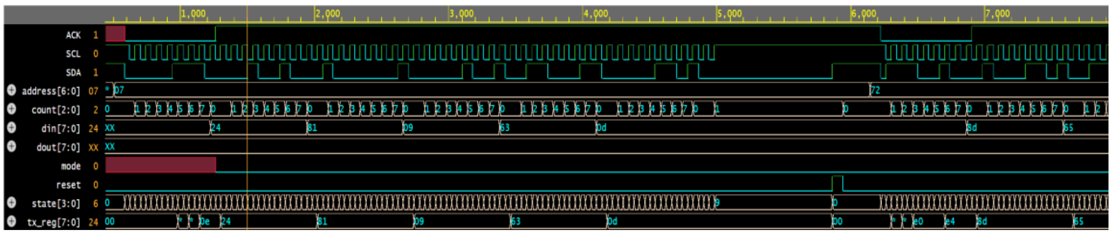


Fig.4.2. Waveform showing the correct working of I2C Slave

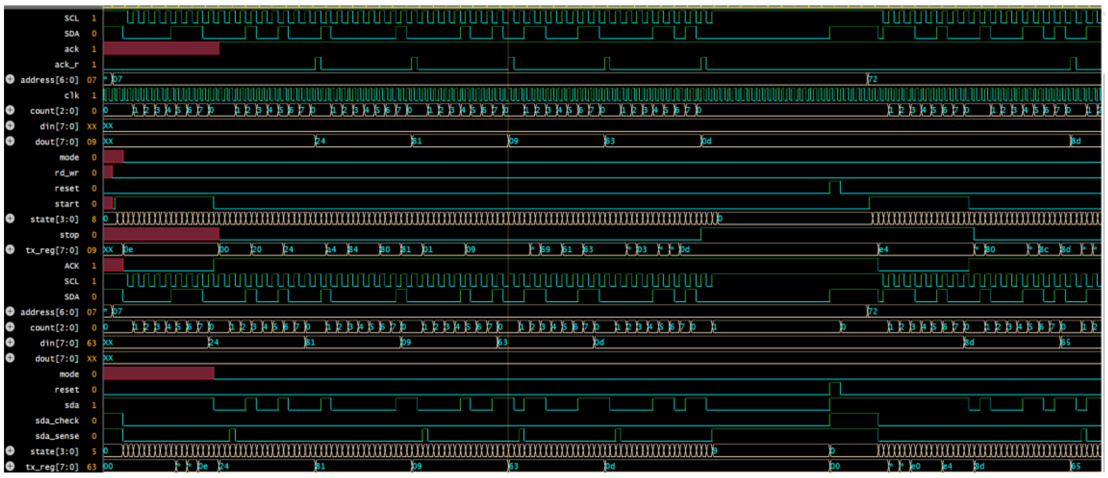


Fig.4.3. Handshake between I2C Master and Slave

4.2 DATA COMPARISON SIMULATION OUTPUT

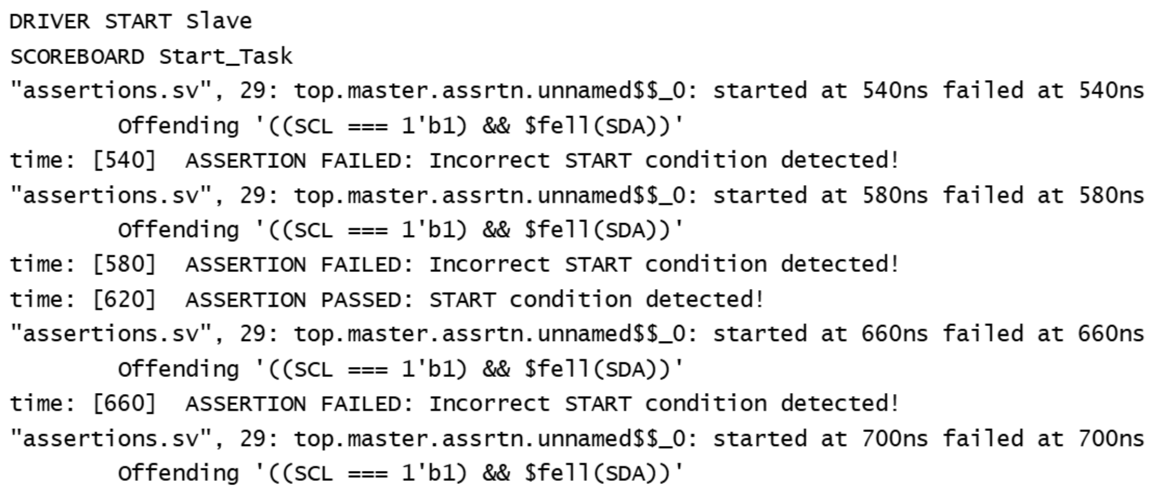
```
Slave sample   data = 24
Master sample  data = 24
COMPARE TASK
PASS [24 , 24]
-----------------------------
Slave sample   data = 81
Master sample  data = 81
COMPARE TASK
PASS [81 , 81]
-----------------------------
Slave sample   data = 09
Master sample  data = 09
COMPARE TASK
PASS [09 , 09]
-----------------------------
Slave sample   data = 63
Master sample  data = 63
COMPARE TASK
PASS [63 , 63]
-----------------------------
Slave sample   data = 0d
Master sample  data = 0d
COMPARE TASK
PASS [0d , 0d]
```

Fig.4.4. Data Comparison Simulation output

The *Fig.4.4* displays the comparison results of data transmitted and received between the master and slave during I2C communication. Each block shows data sampled by the slave and later echoed or matched by the master, along with their respective timestamps. The compare task checks confirm that all data pairs match correctly (e.g., 0x24, 0x81, 0x09, 0x63, 0x0d), indicating proper data exchange. This verifies that the protocol-level data transfer between master and slave is functioning as intended.

4.3 ASSERTION OUTPUT

As mentioned in chapter 3.3.8, there are 9 assertions made for I2C Master and Slave. Assertion checks happen at every clock pulse and prints the display statements stating PASS or FAIL. The *Fig.4.5* shows the results of a System Verilog assertion used to validate the START condition in an I2C protocol. The assertion is checking for a falling edge on SDA while SCL is high, which is the correct definition of an I2C START condition: ((SCL === 1'b1) && $fell(SDA)). Initially, the assertion fails at times 540 ns and 580 ns, indicating that an incorrect START condition was detected—most likely due to a glitch or a protocol timing violation. However, by 620 ns, the condition is met correctly, and the assertion passes, confirming a valid START event was eventually observed. This pattern suggests a timing or signal control issue during the early part of the communication setup. Assertion will hit only once when the START, SCL and SDA satisfies the condition as mentioned in the RTL code. The time when Assertion passes can be verified with *Fig.4.1* also.

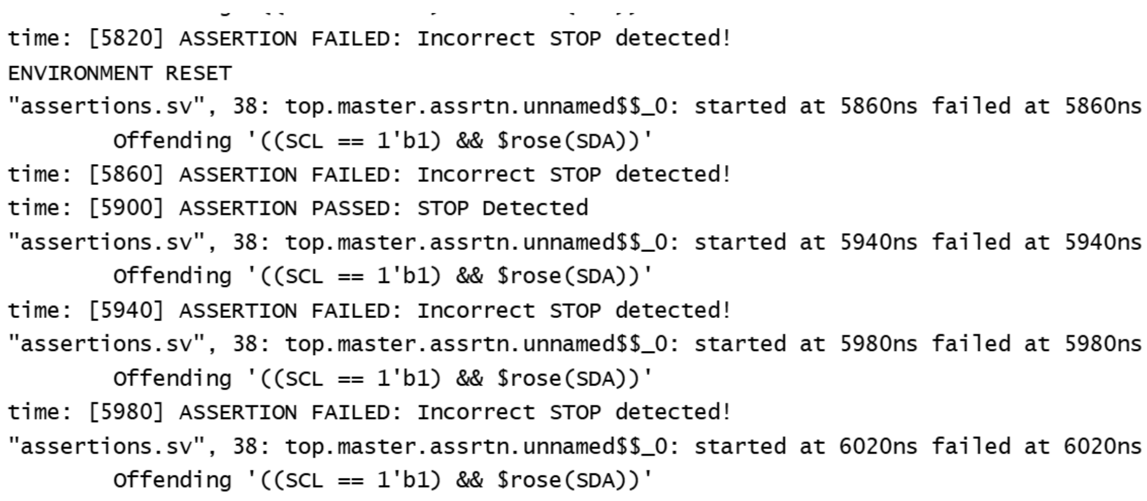


```
DRIVER START Slave
SCOREBOARD Start_Task
"assertions.sv", 29: top.master.assrtn.unnamed$$_0: started at 540ns failed at 540ns
        Offending '((SCL === 1'b1) && $fell(SDA))'
time: [540]  ASSERTION FAILED: Incorrect START condition detected!
"assertions.sv", 29: top.master.assrtn.unnamed$$_0: started at 580ns failed at 580ns
        Offending '((SCL === 1'b1) && $fell(SDA))'
time: [580]  ASSERTION FAILED: Incorrect START condition detected!
time: [620]  ASSERTION PASSED: START condition detected!
"assertions.sv", 29: top.master.assrtn.unnamed$$_0: started at 660ns failed at 660ns
        Offending '((SCL === 1'b1) && $fell(SDA))'
time: [660]  ASSERTION FAILED: Incorrect START condition detected!
"assertions.sv", 29: top.master.assrtn.unnamed$$_0: started at 700ns failed at 700ns
        Offending '((SCL === 1'b1) && $fell(SDA))'
```

Fig.4.5. Assertion START condition result

The *Fig.4.6* shows the assertion simulation output for STOP condition. The output in the figure displays the simulation results of a STOP condition assertion in an I2C protocol. The assertion checks for a valid STOP condition, defined as a rising edge on SDA while SCL is high—expressed as ((SCL == 1'b1) && $rose(SDA)). The logs show that multiple STOP condition checks failed at different time points (e.g., 5820ns, 5860ns, 5940ns, etc.), indicating that the STOP condition was either mistimed or improperly formed during those intervals. However, there is one successful assertion at

5900ns, confirming that a correct STOP condition was detected at that specific moment. This mixed result highlights instability or timing issues in STOP condition handling, suggesting a need to verify the design's control over SCL and SDA during transaction termination.

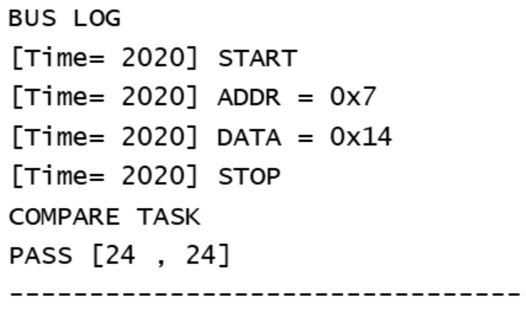


```
time: [5820] ASSERTION FAILED: Incorrect STOP detected!
ENVIRONMENT RESET
"assertions.sv", 38: top.master.assrtn.unnamed$$_0: started at 5860ns failed at 5860ns
        Offending '((SCL == 1'b1) && $rose(SDA))'
time: [5860] ASSERTION FAILED: Incorrect STOP detected!
time: [5900] ASSERTION PASSED: STOP Detected
"assertions.sv", 38: top.master.assrtn.unnamed$$_0: started at 5940ns failed at 5940ns
        Offending '((SCL == 1'b1) && $rose(SDA))'
time: [5940] ASSERTION FAILED: Incorrect STOP detected!
"assertions.sv", 38: top.master.assrtn.unnamed$$_0: started at 5980ns failed at 5980ns
        Offending '((SCL == 1'b1) && $rose(SDA))'
time: [5980] ASSERTION FAILED: Incorrect STOP detected!
"assertions.sv", 38: top.master.assrtn.unnamed$$_0: started at 6020ns failed at 6020ns
        Offending '((SCL == 1'b1) && $rose(SDA))'
```

Fig.4.6. Assertion STOP condition result

4.4 PERL SCRIPT OUTPUT

The displayed log in *Fig.4.7* captures a successful I2C transaction, detailing the sequence of events including START condition, address transmission (0x07), data transfer (0x14), and STOP condition, followed by a successful data comparison (PASS [24, 24]). This structured log format is being used as input to a Perl script developed for automation of verification. The script reads the timestamps, address, and data values, and performs comparisons between master and slave values to detect mismatches, streamlining the validation process and enhancing debugging efficiency.

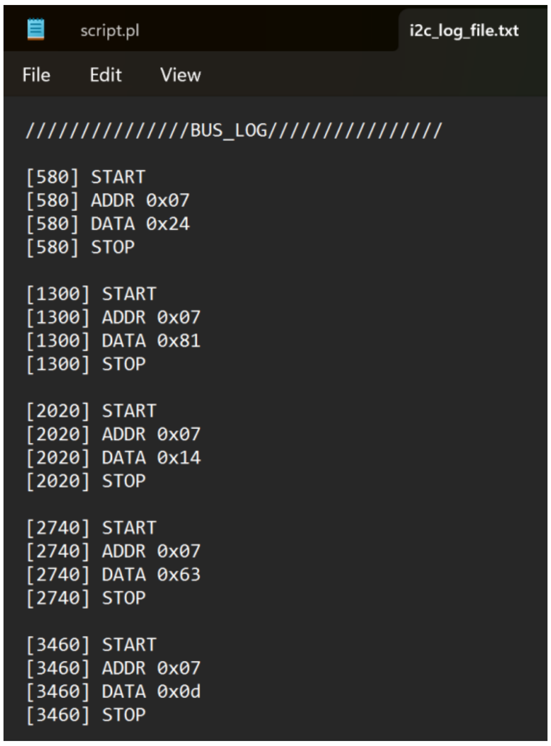


```
BUS LOG
[Time= 2020] START
[Time= 2020] ADDR = 0x7
[Time= 2020] DATA = 0x14
[Time= 2020] STOP
COMPARE TASK
PASS [24 , 24]
--------------------------------
```

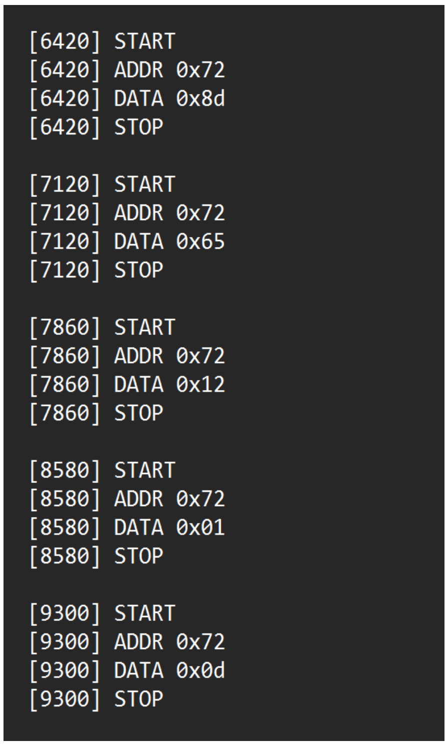
Fig.4.7 Bus Log for one data and

address in simulation time

These data are gathered and used as the input to Perl script created as shown below in the *Fig.4.8.* After running the script in the terminal, it gives the results which includes New Transaction made every time, address, mode, data and the total number of transactions made.



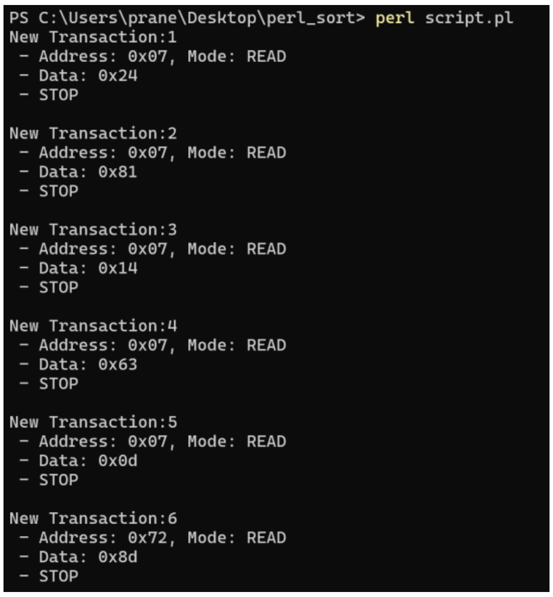
```
//////////////BUS_LOG///////////////

[580] START
[580] ADDR 0x07
[580] DATA 0x24
[580] STOP

[1300] START
[1300] ADDR 0x07
[1300] DATA 0x81
[1300] STOP

[2020] START
[2020] ADDR 0x07
[2020] DATA 0x14
[2020] STOP

[2740] START
[2740] ADDR 0x07
[2740] DATA 0x63
[2740] STOP

[3460] START
[3460] ADDR 0x07
[3460] DATA 0x0d
[3460] STOP
```



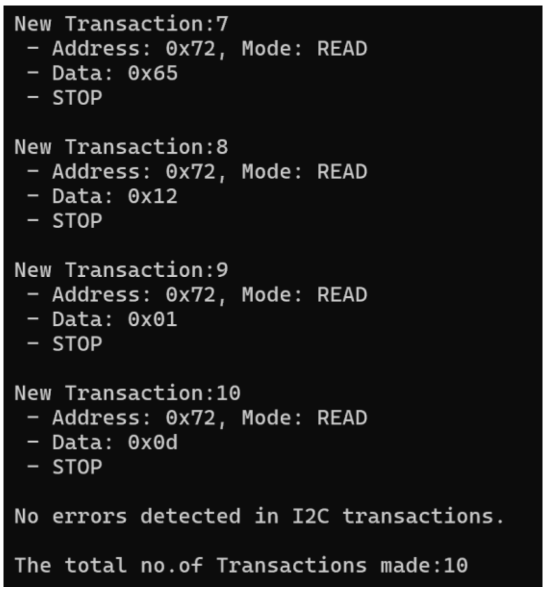
```
[6420] START
[6420] ADDR 0x72
[6420] DATA 0x8d
[6420] STOP

[7120] START
[7120] ADDR 0x72
[7120] DATA 0x65
[7120] STOP

[7860] START
[7860] ADDR 0x72
[7860] DATA 0x12
[7860] STOP

[8580] START
[8580] ADDR 0x72
[8580] DATA 0x01
[8580] STOP

[9300] START
[9300] ADDR 0x72
[9300] DATA 0x0d
[9300] STOP
```

Fig.4.8 I2C Bus log

The displayed terminal output is the result of running a Perl script (script.pl) designed to parse and organize I2C transaction logs. Each transaction from the log is neatly extracted and displayed with essential details such as the transaction number, slave address, communication mode (READ), transmitted data, and the STOP condition. This structured presentation allows for easy inspection and debugging of I2C operations, especially in verifying whether the data received from the master matches the expected protocol behavior. The script effectively groups multiple log entries into meaningful transactions, which is particularly useful during post-simulation analysis in verification environments. This output demonstrates that the Perl script successfully interprets the raw bus data and outputs it in a human-readable, organized format for validation or further automated checks as me.



```
PS C:\Users\prane\Desktop\perl_sort> perl script.pl
New Transaction:1
 - Address: 0x07, Mode: READ
 - Data: 0x24
 - STOP

New Transaction:2
 - Address: 0x07, Mode: READ
 - Data: 0x81
 - STOP

New Transaction:3
 - Address: 0x07, Mode: READ
 - Data: 0x14
 - STOP

New Transaction:4
 - Address: 0x07, Mode: READ
 - Data: 0x63
 - STOP

New Transaction:5
 - Address: 0x07, Mode: READ
 - Data: 0x0d
 - STOP

New Transaction:6
 - Address: 0x72, Mode: READ
 - Data: 0x8d
 - STOP
```



```
New Transaction:7
 - Address: 0x72, Mode: READ
 - Data: 0x65
 - STOP

New Transaction:8
 - Address: 0x72, Mode: READ
 - Data: 0x12
 - STOP

New Transaction:9
 - Address: 0x72, Mode: READ
 - Data: 0x01
 - STOP

New Transaction:10
 - Address: 0x72, Mode: READ
 - Data: 0x0d
 - STOP

No errors detected in I2C transactions.

The total no.of Transactions made:10
```

Fig.4.9 Perl Script Output

CHAPTER 5

CODE COVERAGE

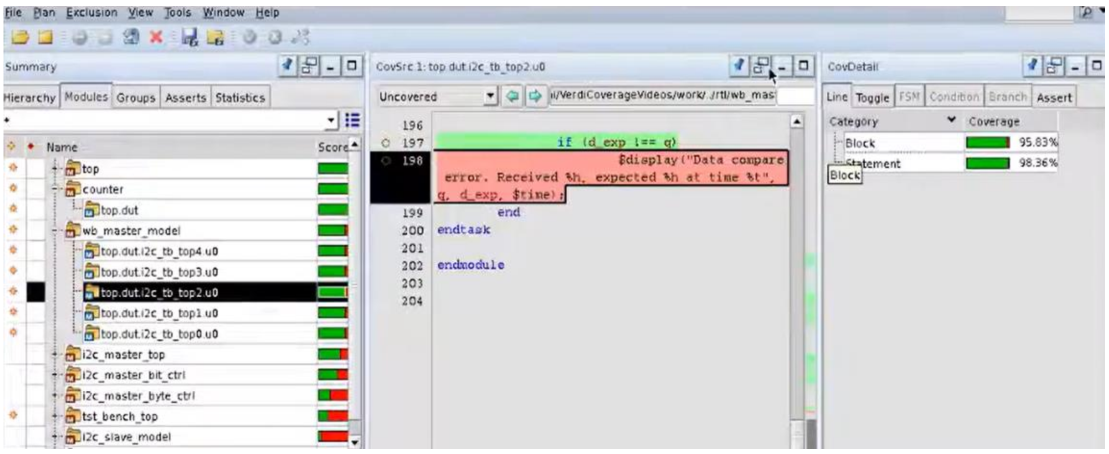
5.1 Work Done in Code Coverage

Code Coverage work was assigned as a task to improve the code coverage of apb_arbiter, apb_sync, paddr_decoder, apb_mux, ctrl_fsm_cm[9] modules. Code coverage is an essential aspect of functional verification in digital design, particularly in complex VLSI projects. It refers to the process of measuring how thoroughly the testbench stimulates the design under test (DUT). The primary goal of code coverage is to ensure that all parts of the design code have been executed and validated by the testbench. Without adequate code coverage, there is a high risk of shipping a design that may behave unpredictably in real-world scenarios because certain logic paths or edge cases were never exercised during simulation.

In industry-grade verification environments, code coverage is used to complement functional coverage. While functional coverage measures whether specific features or scenarios defined by the specification are tested, code coverage focuses on the actual implementation — the lines, conditions, and constructs written in the RTL (usually in Verilog or VHDL). Code coverage analysis allows verification engineers to identify untested areas in the code, improve the quality of testbenches, and ultimately increase confidence in the correctness of the design before tape-out.

5.1.1 Line (Statement) Coverage

This is the most basic form of code coverage. It tracks whether each line of executable code in the RTL was executed at least once during simulation. If certain lines are never executed, it may indicate dead code, unreachable logic, or incomplete test scenarios. Line coverage is a good starting point, but it is not sufficient on its own to guarantee full functional correctness.


File Plan Exclusion View Tools Window Help
Summary
Hierarchy Modules Groups Asserts Statistics
CovSrc 1: top.dut.i2c_tb_top2.u0
Uncovered
CovDetail
Line Toggle FSM Condition Branch Assert

Name	Score
top	🟩
counter	🟩
top.dut	🟩
wb_master_model	🟩
top.dut.i2c_tb_top4.u0	🟩
top.dut.i2c_tb_top3.u0	🟩
top.dut.i2c_tb_top2.u0	🟩
top.dut.i2c_tb_top1.u0	🟩
top.dut.i2c_tb_top0.u0	🟩
i2c_master_top	🟩🟥
i2c_master_bit_ctrl	🟩🟥
i2c_master_byte_ctrl	🟩🟥
tst_bench_top	🟩🟥
i2c_slave_model	🟥

i/VerdiCoverageVideos/work/../rtl/wb_mas

```
                    if (d_exp !== q)
                              $display("Data compare
 error. Received %h, expected %h at time %t",
q, d_exp, $time);
          end
endtask

endmodule
```

Category	Coverage
Block	95.83%
Statement	98.36%

Fig.5.1 Line Coverage in Verdi Tool [9]

5.1.2 Toggle Coverage

Toggle coverage monitors whether individual bits in signals have toggled from 0 → 1 and 1 → 0 during simulation. This is particularly useful for verifying data paths, buses, and registers. For instance, a data bus might always carry zero in the testbench due to an oversight, and toggle coverage would flag this by showing that the bits never changed. Ensuring toggle coverage helps validate that the dynamic behavior of signals is exercised.

5.1.3 Condition Coverage

Condition coverage (also known as sub-expression coverage) ensures that all Boolean conditions in a statement (such as if (a && b)) evaluate to both true and false. It breaks down compound conditions and checks each individual Boolean operand. For example, if a && b is always true because b is never false, the condition coverage will report incomplete coverage. This helps detect hidden logic bugs and incomplete test scenarios.

5.1.4 Branch Coverage

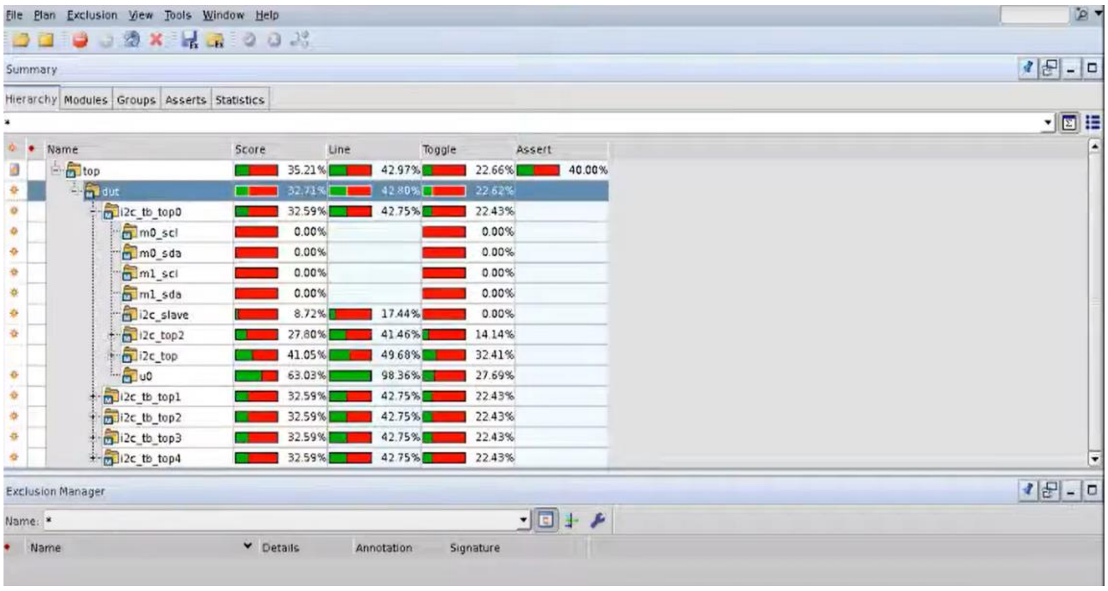
Branch coverage measures whether all branches in the control flow have been exercised — including both the if and else branches, all case items, and loops. This ensures that each decision point in the code has had all possible outcomes tested. Incomplete branch coverage can result in unverified logic paths that could lead to unexpected behaviours.

5.1.5 FSM (Finite State Machine) Coverage

FSM coverage applies to designs that use state machines, which are prevalent in protocol controllers, bus arbiters, and other control-intensive logic. This metric tracks whether all the states and transitions between states have occurred during simulation. Missing state coverage might mean that certain events or edge cases are not being generated in the testbench. This is critical because bugs often hide in rarely used transitions or error states.

5.1.6 Expression Coverage

Expression coverage examines more complex expressions involving multiple variables and operators. It checks all possible combinations of operand values to determine whether the expression has been fully evaluated in every way. For example, in a ternary operator like (sel) ? a : b, expression coverage checks both cases: when sel is true and when it is false. This ensures a more exhaustive validation than condition coverage alone.



File Plan Exclusion View Tools Window Help
Summary
Hierarchy Modules Groups Asserts Statistics
Exclusion Manager
Name: *
Name Details Annotation Signature

Name	Score	Line	Toggle	Assert
top	35.21%	42.97%	22.66%	40.00%
dut	32.71%	42.80%	22.62%	
i2c_tb_top0	32.59%	42.75%	22.43%	
m0_scl	0.00%		0.00%	
m0_sda	0.00%		0.00%	
m1_scl	0.00%		0.00%	
m1_sda	0.00%		0.00%	
i2c_slave	8.72%	17.44%	0.00%	
i2c_top2	27.80%	41.46%	14.14%	
i2c_top	41.05%	49.68%	32.41%	
u0	63.03%	98.36%	27.69%	
i2c_tb_top1	32.59%	42.75%	22.43%	
i2c_tb_top2	32.59%	42.75%	22.43%	
i2c_tb_top3	32.59%	42.75%	22.43%	
i2c_tb_top4	32.59%	42.75%	22.43%	

Fig.5.2 Code Coverage in Verdi tool[9]

5.2 Code coverage flow

In Synopsys Verdi, functional coverage verification entails examining and verifying the testbench stimuli's completeness as well as making sure they exercise the intended functionality of the design under test (DUT)[9]. Metrics for measuring functional coverage keep track of the components of the design that have been evaluated extensively through simulation.

5.2.1 Coverage Database Generation:

• During simulation, Verdi collects coverage data based on the defined coverage points and stores it in a coverage database file (.ucdb)[9]. Before running simulations, you need to define functional coverage points in your SystemVerilog or VHDL testbench code using coverage directives such as covergroup and coverpoint.

5.2.2 Simulation Setup:

• Use Synopsys VCS or another supported simulator to compile and elaborate your design and testbench.

• Enable coverage collection in the simulation by specifying appropriate command line options or settings in your simulation scripts.

5.2.3 Simulation and Coverage Collection:

• Run your simulation with the enabled coverage collection.

• Verdi collects coverage data during simulation execution and updates the coverage database file (.ucdb) with the coverage results.

5.2.4 Integration with Verification Planning Tools:

• Verdi can be integrated with verification planning tools such as Synopsys VC LP (Verification Compiler Test) to link coverage goals defined in the verification plan with actual coverage results obtained during simulation.

• This integration enables you to track coverage progress against verification plan goals and ensure that verification objectives are met.

Verdi Code Coverage is a powerful feature of the Synopsys Verdi tool suite, designed to assist verification engineers in analyzing and optimizing testbench effectiveness by tracking how thoroughly the RTL code has been exercised during simulation. Verdi is widely used in the semiconductor industry not only for waveform debugging and tracing signals but also for in-depth functional and code coverage analysis. When integrated with simulators like VCS[9], Verdi provides a comprehensive and interactive GUI for visualizing coverage metrics.

Additionally, Verdi supports merging coverage data from multiple simulation runs, which is essential when dealing with randomized testing or large designs. Engineers can also set coverage goals, filter out irrelevant sections (using exclude directives), and generate detailed reports to track verification progress. Overall, Verdi’s code coverage features are essential in ensuring high-quality, thoroughly tested digital designs before tape-out.

CHAPTER 6

SUMMARY AND CONCLUSION

6.1 Summary

The Inter-Integrated Circuit (I^2C) protocol, introduced by Philips (now NXP Semiconductors), is a popular two-wire, bidirectional communication protocol widely used for low-speed, short-distance data exchange between integrated circuits, particularly in embedded systems. Its defining features include simplicity, scalability, and support for multi-master and multi-slave communication over just two lines: SDA (Serial Data) and SCL (Serial Clock).

I^2C operates in master-slave architecture, where the master initiates communication and controls the clock, while the slave responds based on the address sent by the master. Data is transferred in 8-bit packets, accompanied by acknowledgment (ACK) bits, and the communication is synchronized using clock edges. It supports multiple speed modes — Standard (100 kbps), Fast (400 kbps), Fast Plus (1 Mbps)[2], and High-Speed Mode (3.4 Mbps).

From a hardware implementation perspective, I^2C is relatively simple, requiring only two pull-up resistors and open-drain or open-collector outputs. It has become an industry standard for interfacing peripheral devices like EEPROMs, RTCs, ADC/DACs, sensors, and display drivers. Its bus arbitration and synchronization mechanisms make it suitable for complex systems with multiple masters.

In recent VLSI and SoC design research, I^2C is being extensively modeled, simulated, and verified using Verilog, VHDL, UVM, and SystemVerilog[1] testbenches. Verification environments often include drivers, monitors, and scoreboards, with synchronization techniques using clocks and assertions. Simulation outputs and logs are used to drive further analysis and automation through scripts, including Perl and Python.

Moreover, design coverage and code coverage analysis using tools like Synopsys Verdi are vital in verifying that all paths, conditions, and state transitions in I^2C RTL designs are exercised. Assertions are employed to ensure protocol compliance, such as start/stop detection, address validity, data integrity, and timing correctness.

6.2 Conclusion:

In conclusion, the design of a high-speed I2C interface protocol and the implementation of rapid data exchange present a compelling narrative of innovation and efficiency in communication technology [1]. Through meticulous planning, optimization, and rigorous testing, the development of a high-speed I2C interface protocol has enabled seamless communication between devices, facilitating faster data transfer rates and enhanced system performance. By prioritizing speed, reliability, and efficiency, these mechanisms ensure swift and efficient exchange of data, enabling real-time processing, analysis, and decision-making in various applications. In both endeavors, collaboration, and interdisciplinary. This collaborative effort led to the creation of high-performance communication solutions that drive innovation and propel technological advancements forward. The future holds exciting possibilities for further optimization and refinement of high-speed communication protocols and data exchange mechanisms [1]. As technology continues to evolve, there will be a growing need for faster, more efficient. The design of high-speed I2C interface protocols and rapid data exchange mechanisms represents a milestone in communication technology, paving the way for enhanced connectivity, improved system performance, and transformative innovations in the digital age.

The I^{2}C protocol implementation and verification performed in this project reflect a structured and disciplined approach typical of modern VLSI design environments. The simulation flow was built using SystemVerilog and included master and slave drivers, interfaces, environment instantiation, and the use of key verification constructs such as mailboxes and assertions. This allowed for the functional simulation of write and read operations across multiple clock cycles, ensuring the protocol behaved according to specification. Assertions, such as those checking for START and STOP conditions, added robustness by automatically flagging any violations, while log files were parsed using Perl scripts to automate result extraction and analysis.

6.3 Future Work

Based on the code and simulation outputs discussed above, it is evident that the I^{2}C protocol was implemented and verified using SystemVerilog constructs, including drivers, interfaces, and assertions to validate protocol functionality such as START, STOP, data transfer, and acknowledgement handling. This detailed verification setup, often part of a VLSI design verification flow, showcases how the protocol is mimicked through master and slave modules with appropriate synchronization on the clock and control lines. The implementation leverages functional coverage and assertion-based verification to ensure correctness. In the context of VLSI, where reliability and correctness are paramount, such verification environments are vital for validating the RTL at an early stage. Tools like Synopsys Verdi play a crucial role here, helping engineers debug waveform issues, monitor assertions, and analyze code coverage. The Perl scripting integration for log parsing further supports automation in the verification process, extracting useful metrics and validation points. As VLSI designs grow complex, this level of detailed simulation and verification will form the foundation for scalable and reusable IP blocks, ensuring protocol compliance and robustness in larger SoC architectures.

REFERENCES

[1] A. Chaithanya and A. Sainath, "Design of High-Speed I²C Interface Protocol Design and Its Verification Using SystemVerilog," *International Journal of Engineering Science and Research*, vol. 7, no. 5, pp. 647–654, May 2017.

[2] R. C. Radha and R. A. Kumar, "Design and Implementation of I²C Communication Protocol on FPGA for EEPROM," *International Journal of Scientific and Engineering Research*, vol. 9, no. 6, pp. 1609–1612, Jun. 2018.

[3] Sajjanar, Sumanth, and Mrs Deepika Prabhakar. "A Review on BIST Embedded I2C and SPI using Verilog HDL." (2020).

[4] D. Thirumurugan and P. Harikrishna, "VLSI Implementation of SPI and I²C Communication Protocols," *International Journal of Advanced Research, Ideas and Innovations in Technology*, vol. 5, no. 4, pp. 403–406, Jul.–Aug. 2019.

[5] F. Hadinata, S. Yuwono, and F. Indrianto, "Design and Implementation of I²C Bus Protocol on Master and Slave Using FPGA," *Makara Journal of Technology*, vol. 26, no. 1, pp. 30–38, Apr. 2022.

[6] A. Tripathi, R. Jha and S. Prakash, "A Comparative Analysis of Communication Protocols in VLSI Design," *International Journal of Research Publication and Reviews*, vol. 5, no. 7, pp. 244–250, Jul. 2024.

[7] S. B. Patel, P. Talati, and S. Gandhi, "Design of I²C Protocol," *International Journal of Technical Innovation in Modern Engineering & Science (IJTIMES)*, vol. 5, no. 3, pp. 741–745, Mar. 2019.

[8] B. Dineshkumar and R. Vignesh, "Design and Implementation of I²C Communication Protocol Using Verilog HDL," *International Journal of Scientific and Engineering Research*, vol. 11, no. 3, pp. 620–625, Mar. 2020.

[9] Synopsys Datasheets and links
https://www.synopsys.com/dw/ipdir.php?ds=proc_periph_i2c

[10] Sagar Patel, Rahul Patel and Jaymin Bhalani, "Performance Analysis & Implementation of different Modulation Techniques in Almouti MIMO Scheme with Rayleigh Channel", International Conference on Recent Trends in computing and Communication Engineering, April 2013

[11] S Patel, VV Dwivedi, YP Kosta, “A Parametric Characterization and Comparative Study of Okumura and Hata Propagation-loss-prediction Models for Wireless Environment”, International Journal of Electronic Engineering Research, Vol. 2, No. 4, 2010

[12] Khelif, Mohamed Amine, Jordane Lorandel, and Olivier Romain. "Non-invasive i2c hardware trojan attack vector." In 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pp. 1-6. IEEE, 2021.

[13]]Nagaraju, C. H., P. L. Mounika, K. Rohini, T. Naga Yaswanth, and A. Maheswar Reddy. "Design and Implementation of Built-In Self-Test (BIST) Master Slave Communication Using I2C Protocol." In International Conference on Communications and Cyber Physical Engineering 2018, pp. 65-73. Singapore: Springer Nature Singapore, 2023.

[14] Ishak, Mohamad Khairi, and Meenal Pradeep Kumar. "Design and Implementation of I2C Bus Protocol on Master and Slave Data Transfer Based on FPGA." Makara Journal of Technology 26, no. 1: 5.

[15] Malviya, Utsav Kumar, and Gopal Kumar. "Tiny I2C protocol for camera command exchange in CSI-2: a review." In 2020 International Conference on Inventive Computation Technologies (ICICT), pp. 149-154. IEEE, 2020.

[16] Shilaskar, Swati, Anup Behare, Ketki Sonawane, and Shripad Bhatlawande. "Post Silicon Validation for I2C (SMBUS) Peripheral." In 2023 36th International Conference on VLSI Design and 2023 22nd International Conference on Embedded Systems (VLSID), pp. 313-318. IEEE, 2023.

[17] S. S. Raut and A. S. Patil, "Design and Implementation of I²C Protocol," *JournalNX – A Multidisciplinary Peer Reviewed Journal*, vol. 7, no. 5, pp. 46–50, May 2021.