**DESIGN UNDER TEST IN VERILOG, CREATE A 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 2024**

**DESIGN UNDER TEST IN VERILOG, CREATE A 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 2024**

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.

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

**TABLE OF CONTENTS**

**Title** Page No.

**ABSTRACT** ........................................................................................................... i

**ACKNOWLEDGEMENTS** ................................................................................... ii

**TABLE** **OF CONTENTS** ....................................................................................... iii

**LIST OF TABLES**................................................................................................... v

**LIST OF FIGURES**................................................................................................. vi

**CHAPTER 1 INTRODUCTION**

1.1 General Motivation ……………………………………………………….. 1

1.2 Our Contribution ………………………………………………………….. 2

**CHAPTER 2 LITERATURE REVIEW**

2.1 Related works …………………………………………………………………. 3

**CHAPTER 3 METHODOLOGY**

**CHAPTER 4 RESULTS AND DISCUSSION**

**CHAPTER 5 SUMMARY AND CONCLUSION**

**REFERENCES**

**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. 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.

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.

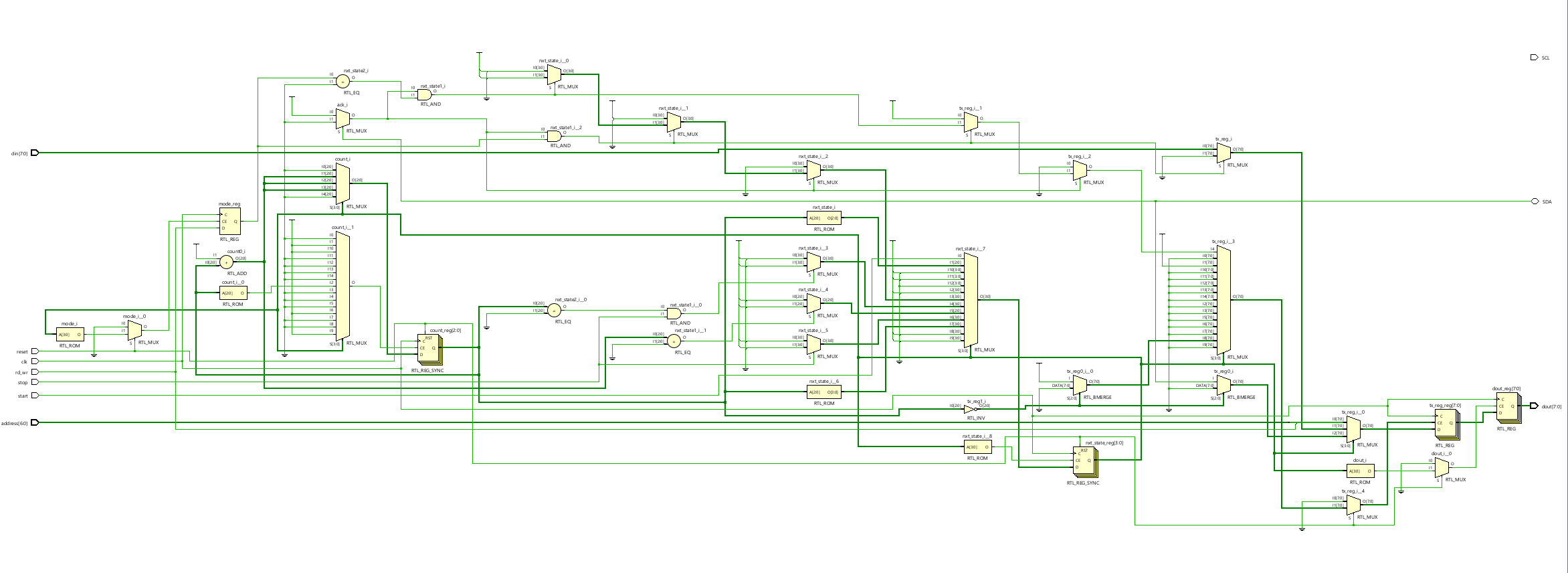
Furthermore, in advanced chip designs, I²C signals might be routed over SERDES (Serializer/Deserializer) lanes in compact, high-speed environments. This makes protocol verification more complex, as control signals may be serialized along with other traffic. Nevertheless, fundamental I²C compliance must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

**1.2 Our 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

**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. The bidirectional nature of SDA is handled using conditional assignments to support reading acknowledgments and data from the slave.

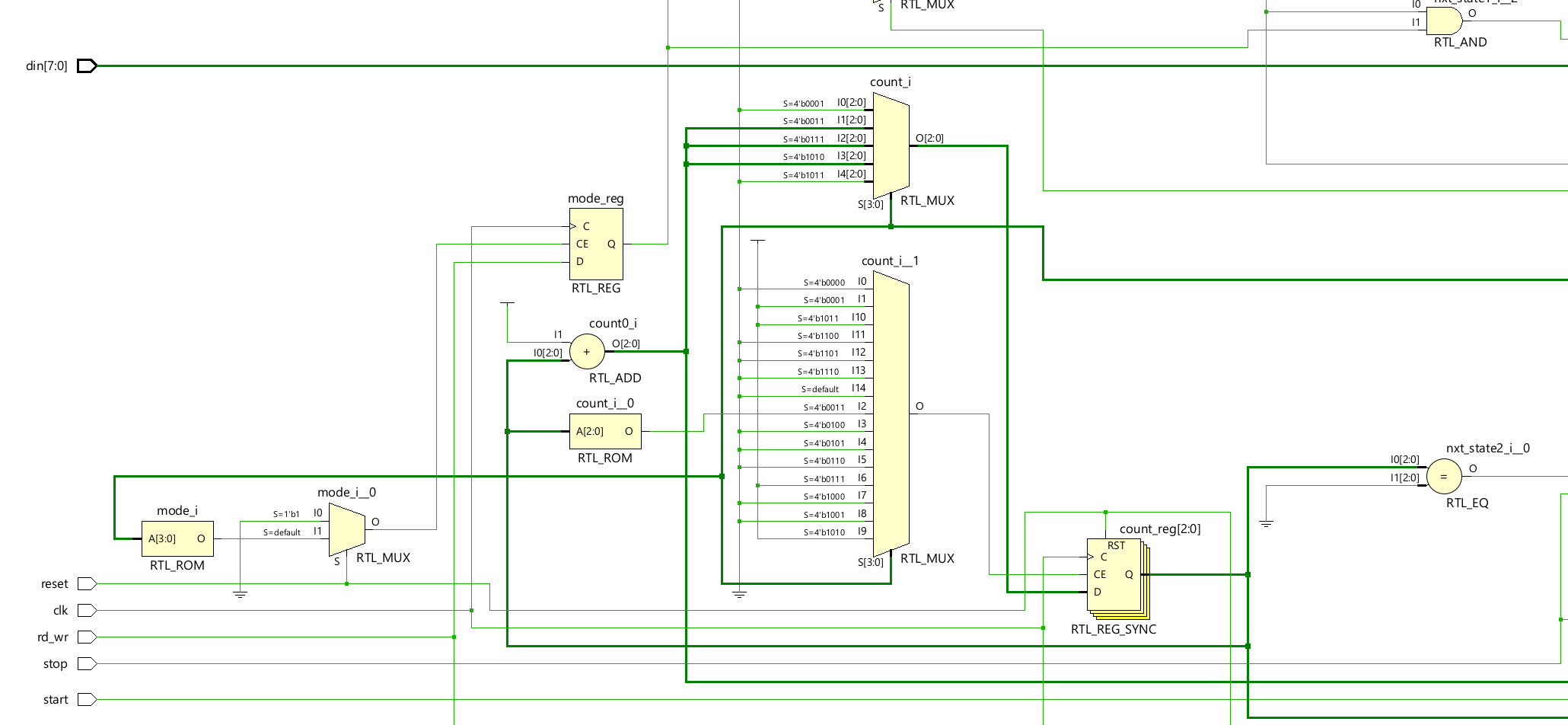


**3.1.1 State Machine Design**

The I²C master logic is governed by a finite state machine (FSM) with 15 defined states, from S0 (idle) to S14 (stop). 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. 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.



**3.1.3 Acknowledge Phase**

Once the address and R/W bit are transmitted, the FSM moves into the acknowledgment phase (S4 and S5). 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**

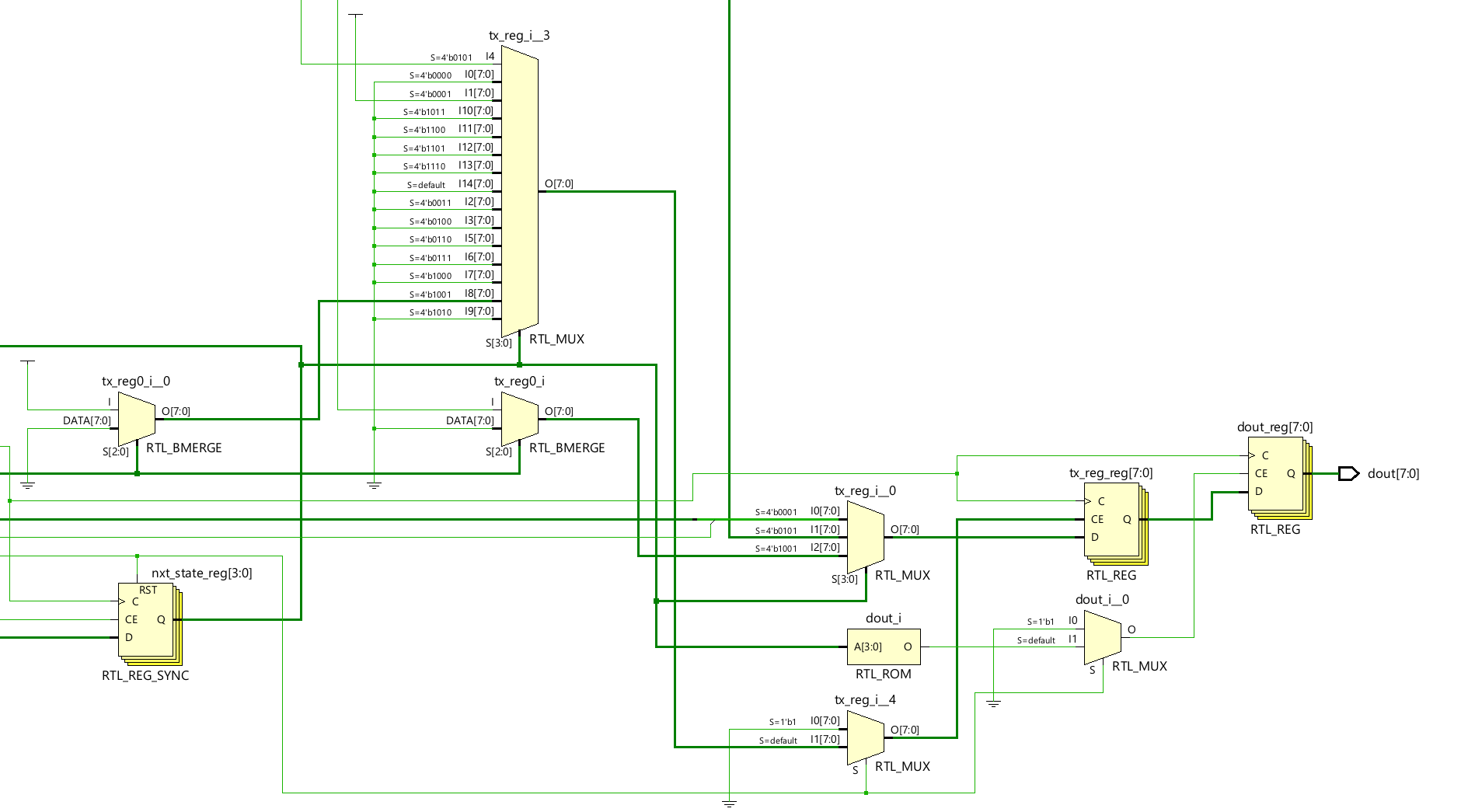
In write mode (rd\_wr = 0), the FSM enters states S6 and S7, where the 8-bit data from the din input is loaded into tx\_reg and sent to the slave, bit by bit, on the SDA line. As each bit is placed on SDA during the falling edge of SCL, the slave samples it on the rising edge. After transmitting 8 bits, 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.

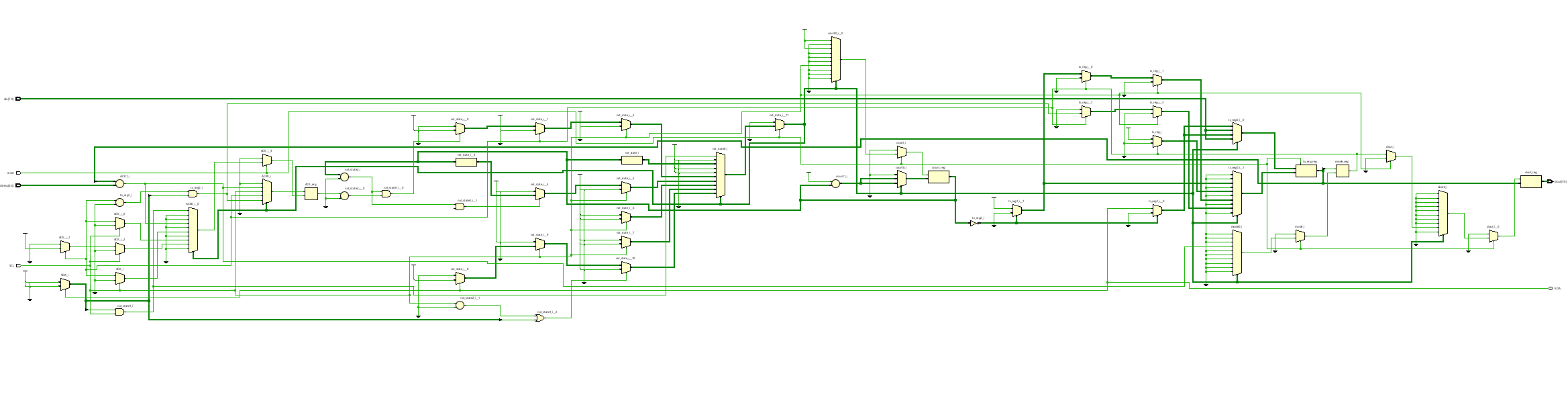


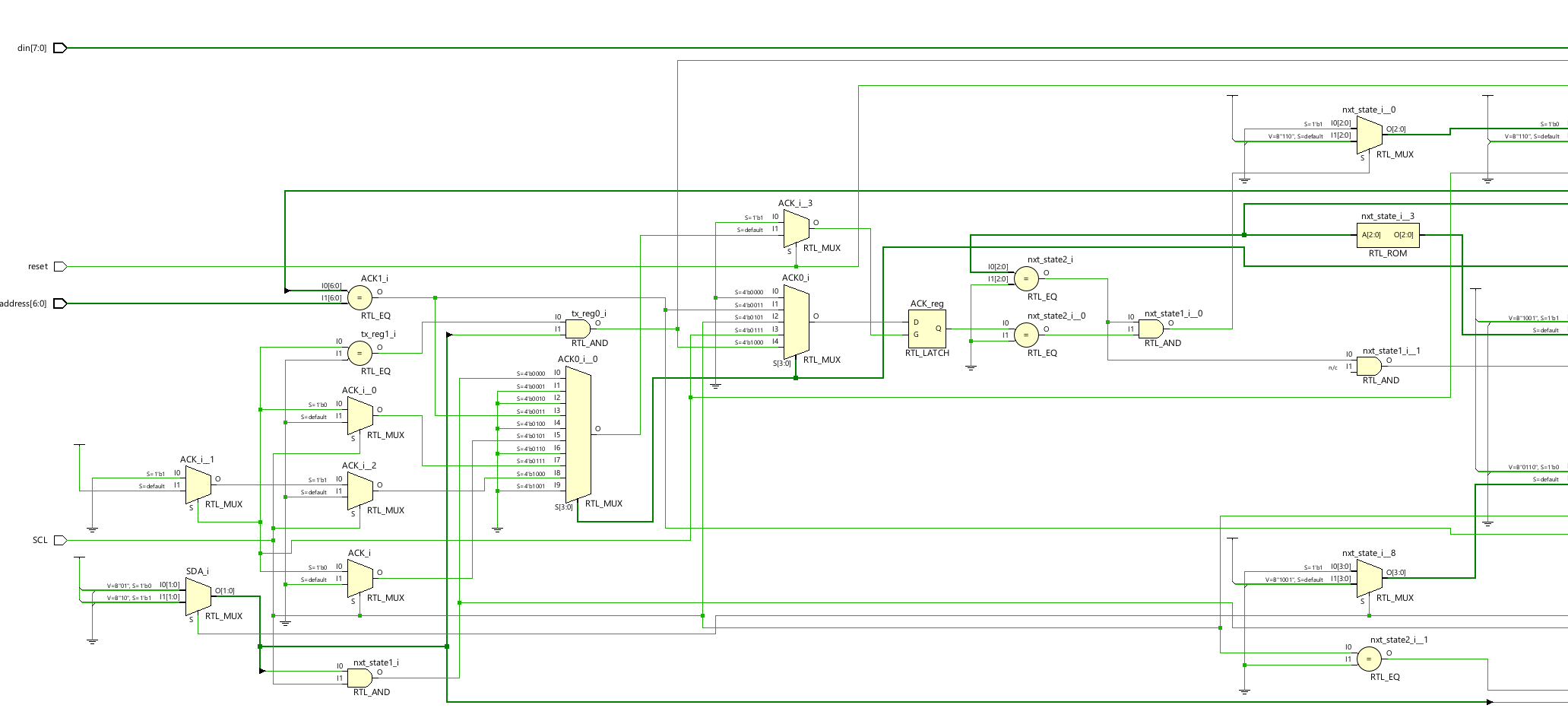
**3.1.6 Open-Drain Handling for SDA and SCL**

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) 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. Internally, it uses a finite state machine (FSM), a data register, and control signals such as ACK, mode, and count to manage protocol operations.





**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, 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**

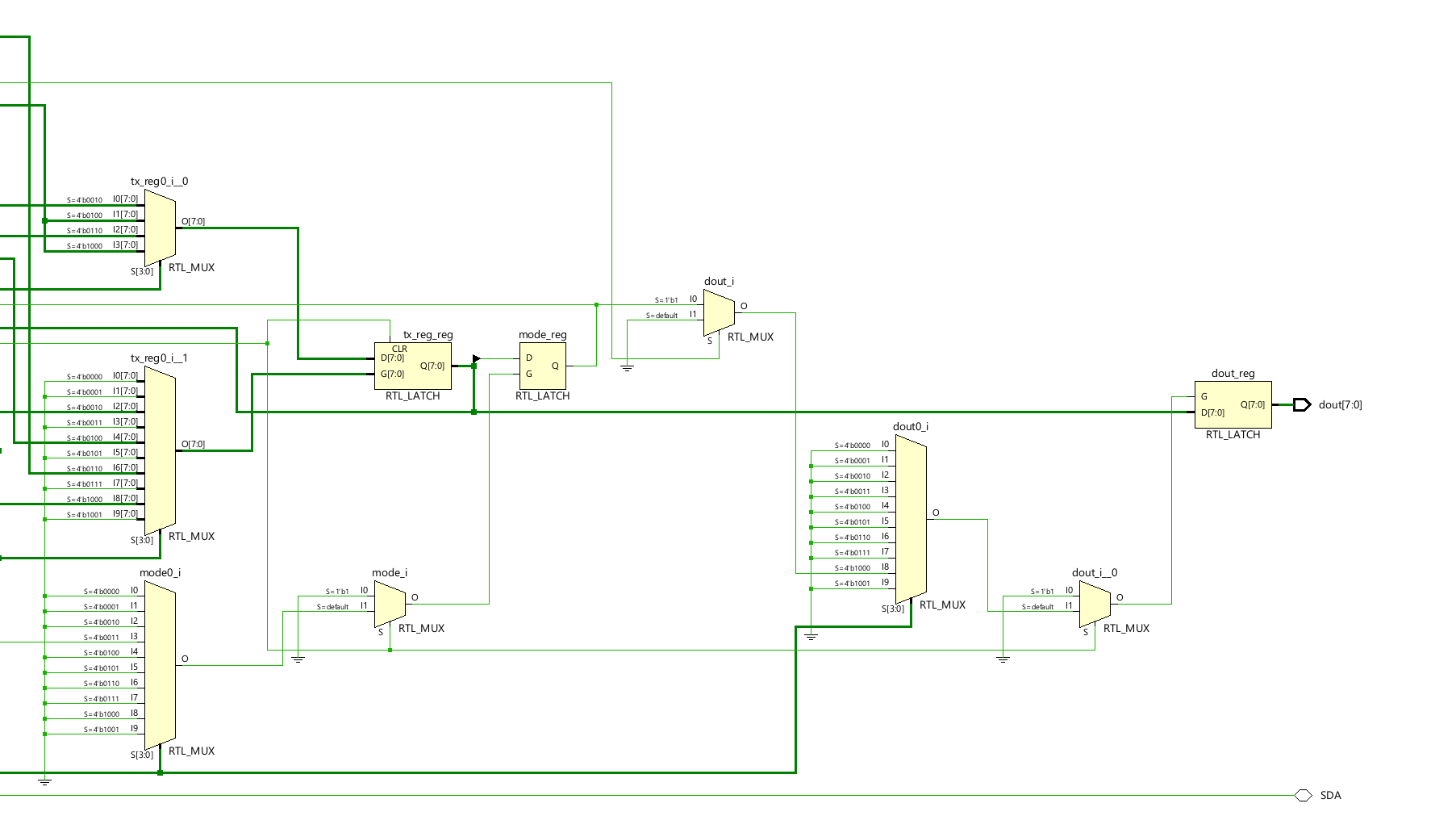
In states S1 and S2, the slave shifts in 8 bits of data from the SDA line, storing them in tx\_reg. These bits represent the 7-bit address and 1-bit R/W command sent by the master. In S3, the module compares the received address (top 7 bits of tx\_reg) with its own. If matched, the slave acknowledges 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.

**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.



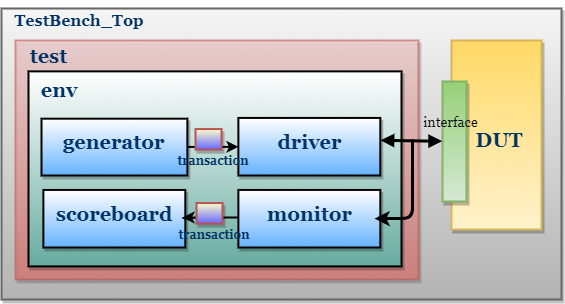
**3.2.5 Stop Condition and Recovery**

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) depending on the mode. The FSM is also designed to handle repeated starts.

**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;. 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**



These are the main Components of System Verilog

**3.3.1 packet (transaction)**

The packet class in System Verilog models a data transaction for I²C 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 Interface**

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

**3.3.3 Driver**

This SystemVerilog code defines two classes—master\_driver and slave\_driver—that model the behavior of I²C 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 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. 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.

**3.3.5 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.

**3.3.6 Assertion**