ECE 337 Lab 6: I2C Transmitting Slave Mini Design Project

In this lab, you will:

- Design and Test via a Test Bench each of the following functional units for the I^{2}C Slave Transmitter Design: I^{2}C Slave Controller, RX Shift Register, TX Shift Register, Transmit FIFO Wrapper, I^{2}C Decode, I^{2}C SCL Counter and SDA Selector, SCL Edge Detector.
- Combine the aforementioned blocks in order to create the I^{2}C Slave Design
- Generate a Test Bench to test the functionality of the I^{2}C SLAVE.
- Synthesize the I^{2}C SLAVE using Synopsys.
- Test the Synthesized/MAPPED version of the I^{2}C SLAVE.

1. Copying Setup Files

In a UNIX terminal window, issue the following command:

```
setup6
```

This command is actually an alias to a batch/script file in the ece337/Class0.5u directory that will copy the files that are necessary for the completion of this lab. **If you have trouble with this step, please ask for assistance from your TA.**

2. Expectations for Lab 6 (and beyond)

From the first day of class, you have been gathering the knowledge and expertise with the tools to allow you to complete this design. At this point in the course, you should know how to operate all the tools that were introduced to you in Labs 1 through 4. In addition, up to this point, you have been introduced to all the Verilog syntax and constructs that are needed to implement the I^{2}C Slave Device. That is to say that with the knowledge of Verilog that you have you should be able to complete this design.

This lab is structured to mimic what you would encounter should you choose to pursue a career as an ASIC/VLSI designer upon graduation. Essentially, you, the designer, are being provided with a set of specifications, by your boss. You are not being instructed on how to design these blocks, and are only given an architecture that defines a functional segmentation of the design to use. It is up to you to come up with a working solution for each block and then integrate them all to form the I^{2}C Slave block.

3. Grading Policy, Deadlines, and Submit Commands

For Lab 6, there are three required deadlines and one optional deadline.

3.1. Required Preparation Phase ('submit Lab6Prep')

For the preparation phase, you will need to complete State Transition Diagrams for:

- Slave Controller block (controller.sv) (Described in section 6.11)

For the preparation phase, you will need to complete Pseudo-code for:

- Timer block (timer.sv) (Described in section 6.10)

For the preparation phase, you will need to complete functional block diagrams for:

- Timer block (timer.sv) (Described in section 6.10)

For the preparation phase, you will need to complete RTL diagrams for:

- Decode block (decode.sv) (Described in section 6.6)
- Slave Controller block (controller.sv) (Described in section 6.11)

For the preparation phase, you will need to complete schematic diagrams for:

- SCL Edge Detector block (scl_edge.sv) (Described in section 6.4)
- SDA Output Selection block (sda_sel.sv) (Described in section 6.5)

Reminder: All diagrams, both RTL and state transition diagrams, must be done as a digital drawing and saved as either a pdf or an image file. Hand drawn diagrams or non-pdf/image files will receive a grade of zero points.

You must have these diagrams and pseudo-code both signed off and submitted via the "submit Lab6Prep" command no later than the following deadlines

- Tuesday Labs: **End of Monday office hours**
- Wednesday Lab: **End of Tuesday office hours**
- Thursday Lab: **End of Wednesday office hours**

3.2. Phase 1 ('submit Lab6Phase1')

Code and test benches due by late night the day of your lab during the second week of Lab 6.

For Phase 1, you will need to complete code and test benches for:

- SCL Edge Detector block (scl_edge.sv) (Described in section 6.4)
- SDA Output Selection block (sda_sel.sv) (Described in section 6.5)
- Decode block (decode.sv) (Described in section 6.6)
- Transmitting FIFO wrapper (tx_fifo.sv) (Described in section 6.9)

The completed blocks and their respective test benches will be graded based on automated grading test benches and test modules respectively. The coverage reports will be used for grading and so you should strive for full 1/0 toggle, statement, branch, expression coverage for each coded module and finite-state-machine (FSM) for modules using state machines. However, the intent for the tx_fifo test bench is for you to familiarize yourself with the fifo module provided to you, therefore you only need to cover the basic usage cases and do not need to worry about coverage reporting for the tx_fifo code. All blocks and test benches must be submitted via the "**submit Lab6Phase1**" command. Y**ou will only have 3 submission attempts for the phase 1 submission**.

3.3. Pre-Phase 2 Test Runs (Optional) ('submit Lab6PrePhase2')

To encourage early testing of your design, you will have three chances to test the source code of your complete design against the grading script without penalty, via the "**submit Lab6PrePhase2**" command, **up to 48 hours before you lab's Phase 2 deadline given below**.

3.4. Phase 2 ('submit Lab6Phase2')

The mapped version of your complete design must be working and will be graded based on the most recent submission score for your mapped design. This is due by the start of lab during the first week of Lab 7. The command for Phase 2 is '**submit Lab6Phase2**'.

You will get 3 chances to use the automated grading script, and your grade for this phase will be based solely on your mapped score. The test cases are organized into groups and you must completely pass all of the test cases in a more basic group before the grading script will award points and inform you of the results of more advanced test cases. Also, if your runs are earning below 50% please consult with the course staff to clarify any misunderstanding or misinterpretation of the specifications provided.

For your lab to be considered complete (to pass the course outcome) your mapped version must be earning at least 50% of the points in the automated tests.

If at any time you would like to check the last set of graded results, you can use "check LabX" or "submit LabX –c", where "LabX" corresponds to the submission phase/lab (ie. Lab6c).

4. Comments

- **You are to work on this lab on your own. You are not to share your test benches or your code with anyone else.**
- The code for the test bench used by the grading script will not be disclosed to the student nor will the specific test cases be told to the student.
- The majority of the points for the lab total come from successfully passing the synthesized design test bench; therefore it is highly recommended to make sure you have an error free run through Synopsys.
- You will be allowed a total maximum of 6 passes through the Lab 6 TA test bench: 3 for the optional complete source grading, with "submit Lab6b", and 3 for the complete mapped grading in Phase 2, with "submit Lab6c".
- **Make sure that your "makefile" is in your Lab6 folder and is up to date and that you can successfully simulate your mapped version with the "make sim_full_mapped" command after running "make veryclean", or the grading script will not be able to get the information it needs to properly grade your design.**
- **Remember that Verilog is case sensitive and thus all ports must be named with the exact same spelling and case as shown in the port descriptions tables otherwise the grading system will be unable to simulate your design.**

5. The I2C-bus Protocol

5.1. Introduction to I2C

The I^{2}C-Bus is a simple bi-directional synchronous 2-wire bus, one data (SDA) and one bus clock (SCL), which allows for the serial transmission of data to and from hardware devices, as shown in figure 1.

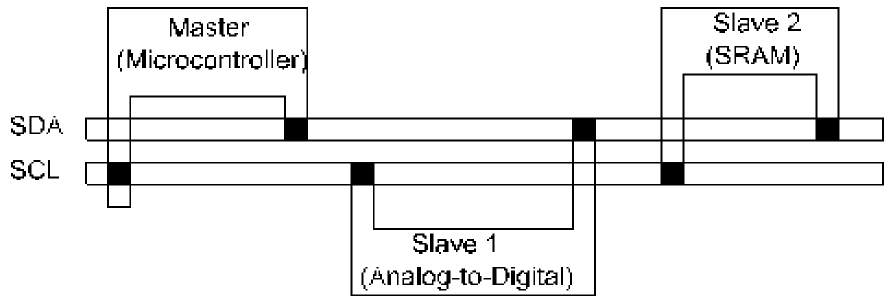

Figure 1: Example I^{2}C Bus Configuration

Philips Semiconductor developed the I^{2}C (Inter IC)-bus to provide a communication link between ICs (Integrated Circuits) in a manner which maximizes the hardware efficiency and simplifies interconnect circuitry. It is a standard that has been widely accepted and is seen in consumer, telecommunications, automotive, and industrial electronics.

Before explaining the I^{2}C-bus protocol, one needs to keep in mind that:

1. Each device connected to the I^{2}C bus can be paged through a unique address
2. Each device may operate as a transmitter along with being a receiver
3. The master controls the bus clock generation (SCL), generation of the START and STOP conditions (the start and stop of a data transfer) and address generation for paging other I^{2}C devices.

Also please note that there are 5 data rate standard ranges in the modern I^{2}C standard:

- Slow mode (up to 10 Kb/s) @ (50% duty cycle)
- Standard mode (up to 100 Kb/s) @ (50 % duty cycle)
- Fast mode (up to 400 Kb/s) @ (50% duty cycle)
- Fast-mode Plus (up to 1 Mb/s) @ (50% duty cycle)
- High-speed mode (up to 3.4 Mb/s) @ (33.3% duty cycle)

For this lab we will be using 3.333 Mb/s (300 ns period) (high-speed mode) as our data rate/SCL clock rate for the design.

5.2. Protocol Overview

The I^{2}C bus protocol is a synchronous protocol and thus all data bits are transmitted in sync with the bus clock. Additionally, since the bus is shared between multiple devices there is a designated master device which controls communication on the bus by generating the bus clock signal as well as the START and STOP conditions, while the other devices on the bus operate as slave devices. All communication to and from slaves is both triggered and controlled by the master device, in a call and response like manner visually described in figure 2 and as follows:

1. A START condition is used to alert all slaves on the bus that communication is going to occur, as they must all listen to determine if they are the one that master wants to communicate with.
2. The master then sends a 7-bit device address followed by a bit that represents the mode of the planned transaction from the master's perspective. Thus:
    a. Read mode = Slave transmits data to the master
    b. Write mode = Slave receives data from the master
3. After the address and communication mode information is received by the slaves, if the transmitted address matches a slave and it supports the requested transfer modes then it should acknowledge (ACK) the master, otherwise a slave should not-acknowledge (NACK) the master.
4. If the master received an ACK, then the requested transaction occurs and is concluded by the master triggering a stop condition following the last acknowledgement phase of the transmission. Additionally, the device receiving the transmitted data, this is the master when the slave is transmitting and the slave when the master is transmitting, must ACK after each byte of data sent or the transmission will be terminated by the master. If the master received a NACK, then the master terminates the transmission.
5. If the master is receiving data from a slave, it will generate a NACK to signal the slave's transmitter that it wants to end the transaction. Following the NACK it will either generate a STOP condition or another START condition. If the master is transmitting data to a slave, it will generate either a START condition or a STOP condition following either the ACK phase of last byte that it wants to transmit or receiving a NACK from the slave. When a STOP condition is issued the communication on the bus ends. If another START condition is generated instead, then the master wants to start a new transaction.

The data byte followed by a NACK must always be considered not received by the intended target and so must be kept by the transmitter for a later retransmission triggered by the master.

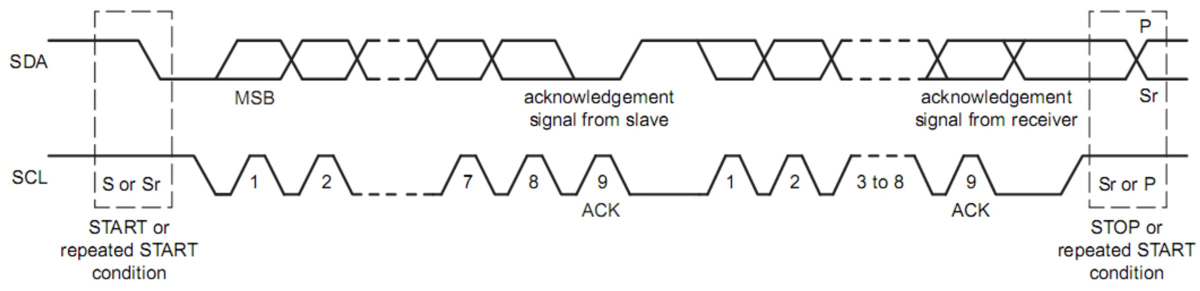
SDA
SCL
MSB
acknowledgement signal from slave
acknowledgement signal from receiver
P
Sr
S or Sr
1
2
7
8
9
ACK
1
2
3 to 8
9
ACK
Sr or P
START or repeated START condition
STOP or repeated START condition

Figure 2: Timing Diagram for an I²C Transmission adapted from Official I²C Specifications

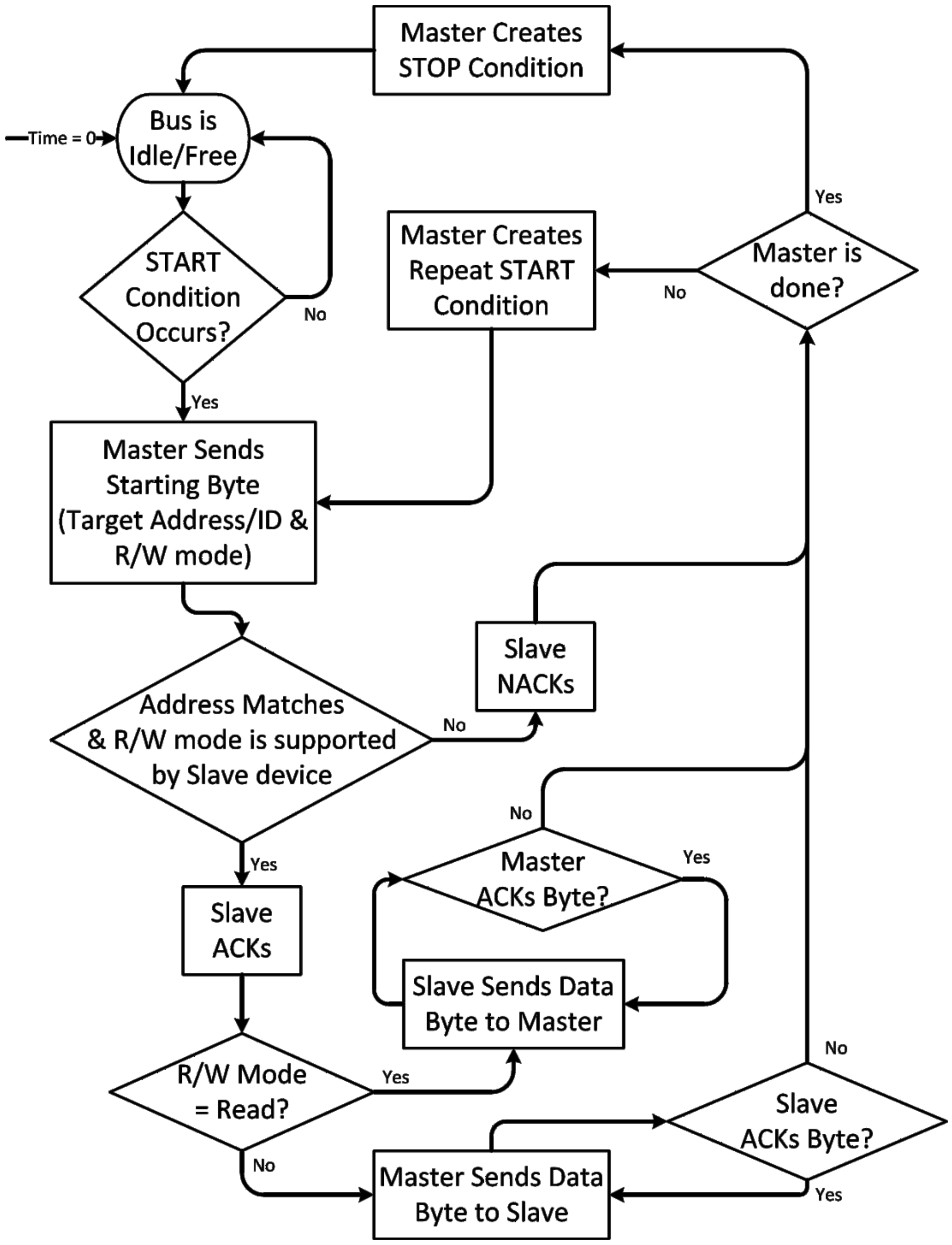
Master Creates STOP Condition
Time = 0
Bus is Idle/Free
START Condition Occurs?
No
Yes
Master Creates Repeat START Condition
No
Yes
Master is done?
Master Sends Starting Byte (Target Address/ID & R/W mode)
Slave NACKs
Address Matches & R/W mode is supported by Slave device
No
Yes
No
Master ACKs Byte?
Yes
Slave ACKs
Slave Sends Data Byte to Master
R/W Mode = Read?
Yes
No
No
Slave ACKs Byte?
Master Sends Data Byte to Slave
Yes

Figure 3: I^{2}C Transmission Flow Diagram

5.3. Physical Bus Characteristics

The I^{2}C-bus Protocol relies on the connecting bus physically consisting of two separate bi-directional buses (one for the SDA line and one for the SCL line) that each cause a logic '0' to override a logic '1' if simultaneously written to the bus. This 0's priority nature is implemented by each device connecting to the bus with open-drain P-channel depletion-mode MOSFETs and the bus being pulled high by a resistor, as depicted in figure 4 below.

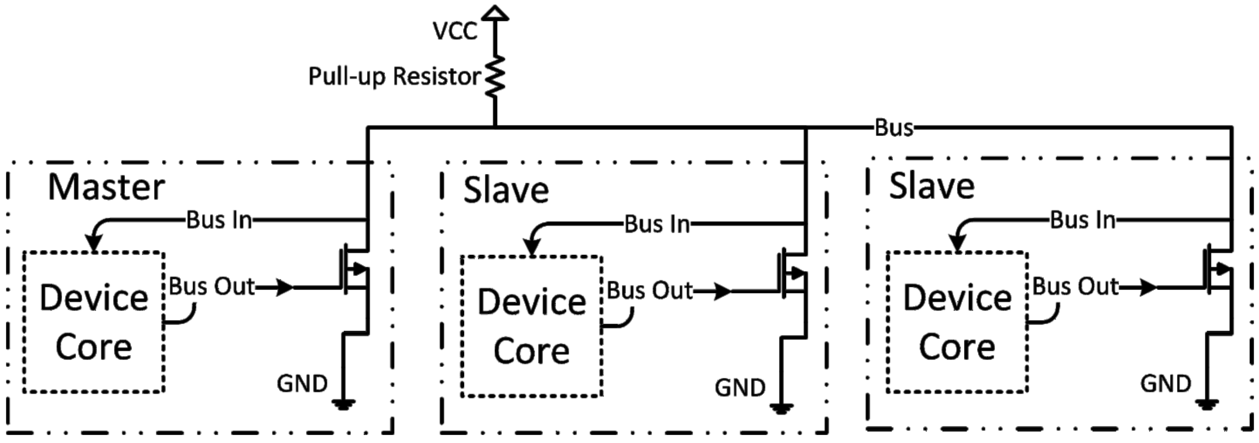

Figure 4: Open-Drain Bus Diagram

5.4. START Condition

For an I^{2}C master to request a data transfer either to or from an I^{2}C slave device, it must generate a START condition. The purpose of the START condition is to alert all the devices connected to the I^{2}C bus that (1) an I^{2}C master is going to perform a data transfer and (2) to be prepared to receive the address of the slave it wants to access, as it might be you. A START condition is recognized by an I^{2}C slave when the slave detects a high-to-low transition on the SDA line while the SCL line continues to stay at a high (logic '1') level, as shown in figure 5.

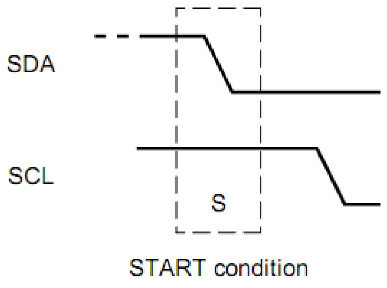

Figure 5: START Condition Diagram adapted from Official I^{2}C Specifications

The minimum time the SDA line must be held high (logic '1'), while the SCL line is held high (logic '1'), before the high-to-low transition may begin is 160 ns. The minimum time the SDA line must be held low (logic '0'), while the SCL line is held high (logic '1'), is 160 ns. Once that time requirement is met, the I^{2}C master will pull the SCL line low (logic '0') and begin to transmit the address of the I^{2}C slave it wishes to communicate with.

5.5. STOP Condition

For an I^{2}C master to request a data transfer either to or from an I^{2}C slave device, it must generate a START condition. The purpose of the STOP condition is to alert all the devices connected to the I^{2}C bus that the current communication session is ending. A STOP condition is recognized by an I^{2}C slave when the slave detects a low-to-high transition on the SDA line while the SCL line continues to stay at a high (logic '1') level, as shown in figure 6.

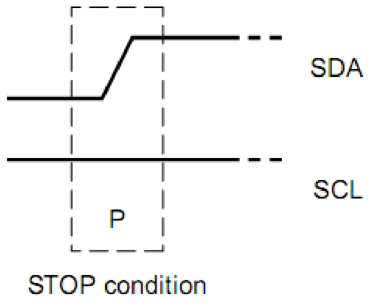

Figure 6: STOP Condition Diagram adapted from Official I^{2}C Specifications

The minimum time the SDA line must be held low (logic '0'), while the SCL line is held high (logic '1'), before the low-to-high transition may begin is 160 ns.

5.6. Byte Format and Bit Transmission Rules

5.6.1. Rules for Byte Format

- Data may only be sent in whole multiples of one byte
- There is no limit on the number of bytes sent during transmissions
- Each byte must be followed by an Acknowledge bit
- All bytes are sent Most Significant Bit (MSB) first

5.6.2. Rules for Sending Bits

- The minimum setup time for the SDA line prior to a rising edge on the SCL line is 10 ns
- The SDA line must remain stable while the SCL line is high (logic '1')
- There is no hold time for the SDA line following a falling edge on the SCL

5.6.3. The Starting Byte (Target Address + Transfer Mode) Format

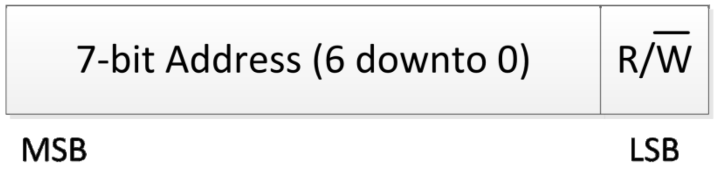

Figure 7: Starting Byte Format Diagram adapted from Official I^{2}C Specifications

5.7. Acknowledge (ACK) and Not Acknowledge (NACK)

Every byte, including the first byte sent by the master containing the target address and transfer mode, must be acknowledged (ACK) or not-acknowledged (NACK) (also referred to as negative acknowledgement). The receiver of a byte **ACK**'s by **pulling the SDA line low** during the bus clock cycle immediately following the last bit in the transmitted byte. This clock cycle is also referred to as the 9th clock cycle or the ACK phase. The receiver of a byte **NACK**'s by **pulling the SDA line high** during the ACK phase. **Any byte that precedes a NACK must be considered to have not been received, or to have been ignored, and so must be retained by the transmitter for a future retransmission triggered by the master.** There are five scenarios where NACK's should occur *(scenarios applicable to your design are bolded)*:

1. **No receiver capable of handling the requested transmission mode is present on the bus with the address transmitted by the master**
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to communicate with the master
3. During the transfer, the receiver gets data or commands it does not understand
4. **During the transfer, the receiver cannot receive any more data bytes, usually due to a full data buffer.**
5. **A master that is receiving data from a slave needs to signal the end of the transfer to the slave. The byte preceding the NACK must be held onto for a later retransmission that will be triggered by the master.**

6. The I2C Transmitting Only Slave Architecture

6.1. A Quick look at a Full I2C Slave Architecture

During this lab you will be designing a version of a I^2C Slave device that is only capable of sending data to a master, instead of also being able to receive data from the master. However, since it will be very similar to a full I^2C Slave design, and you might end up using an I^2C Slave design in your course project, we will first look at an architecture for an I^2C Slave that is capable of both sending and receiving data from a master, as shown in figure 8 below.

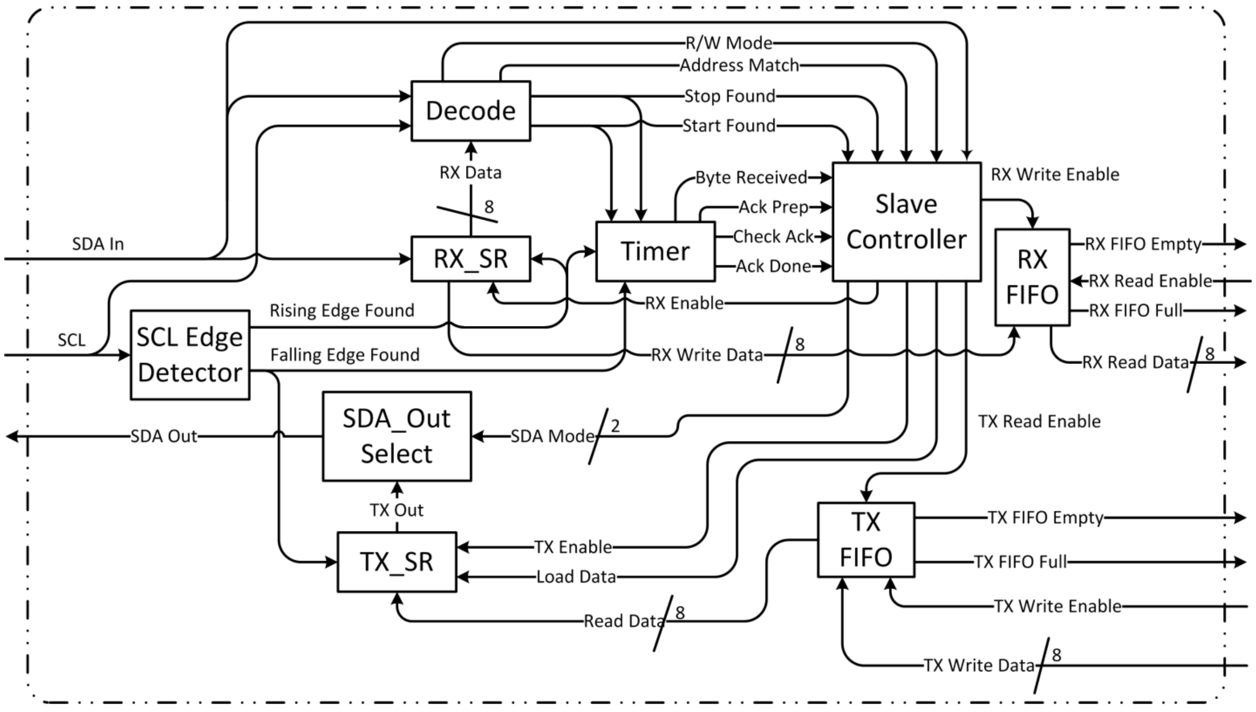

Figure 8: A Simple I2C Slave Architecture

In figure 8, all clock and resent signals are assumed to be sent to the appropriate functional blocks and so are not shown.

6.1.1. List of the Top-Level Ports

- **SDA In**: This is the current value of the SDA bi-directional bus
- **SDA Out**: This is the value the Slave is supplying on the bi-directional bus
- **SCL**: This is the current value of the SCL bi-directional bus
- **RX FIFO Empty**: This is the active-high flag that reports if the RX FIFO is empty
- **RX FIFO Full**: This is the active-high flag that reports if the RX FIFO is full
- **RX Read Enable**: This is the active-high enable signal that tells the RX FIFO that the currently outputted value has been read and the FIFO should advance to the next entry on the next rising-edge of the design's clock
- **RX Read Data**: This is the 8-bit bus that holds the data byte currently available for reading.
- **TX FIFO Empty**: This is the active-high flag that reports if the TX FIFO is empty
- **TX FIFO Full**: This is the active-high flag that reports if the TX FIFO is full
- **TX Write Enable**: This is the active-high enable signal that tells the TX FIFO to write the currently supplied data value into the FIFO storage on the next rising-edge of the design's clock.
- **TX Write Data**: This is the 8-bit bus that holds the data byte to write into the FIFO.
- **CLK**: The system clock for the device
- **N_RST**: The active low chip-wide reset for the design

6.1.2. List of the Functional Units

- **SCL Edge Detector**: This block detects both falling and rising edges on the SCL input and sends the results to the various blocks that depend on those edges.
- **SDA_Out Selector**: This block selects which value to currently output on the SDA Out signal. (Choices are between the current transmit shift register's output, the ACK value, the NACK value, and the idle line value)
- **Decode**: This block decodes/detects START and STOP conditions, check the target address against the device's address, and outputs the requested transmission mode.
- **TX_SR**: This is the 8-bit transmitting shift register.
- **RX_SR**: This is the 8-bit receiving shift register.
- **TX FIFO**: This is the transmission FIFO queue and is used for storing the data bytes to send to the master upon request.
- **RX FIFO**: This is the reception FIFO queue and is used for storing the data bytes sent by the master.
- **Timer**: This block maintains and relays the critical timing information based on the edges in the SCL line and START and STOP conditions.
- **Slave Controller**: This block controls the sequence of operations of the design, including handling when and how to ACK or NACK

6.2. The Architecture for the Transmit-Only I2C Slave Design

This section will describe the required architecture for the Transmit-Only I^{2}C Slave device that you will be designing for your multi-week Lab6. You are required to use the presented architecture (the various functional blocks). However, you will never be told how to directly implement any of the functional blocks, as you are expected to discern what is necessary for the internals of each block and design its implementation based on the described purpose of that block and its input and output signals. As a reminder during Phase 2 your design will be graded only based the required top level functionality and behavior, not based on the specific operation any of the internal blocks. Although, **Phase 1 will grade the components other than the timer and controller according the specifications outlined in this manual**. For this reason, you have to implement those components according to this architecture and specifications, and it is advised that you do so with the timer and controller. Synchronizers may be added using a separate source file for the phase 2 and the Phase 1 components will not be expected to protect themselves from issues that arise with asynchronous inputs. The proposed architecture is illustrated below in figure 9, and the interfaces and purpose of each block is discussed in later sections.

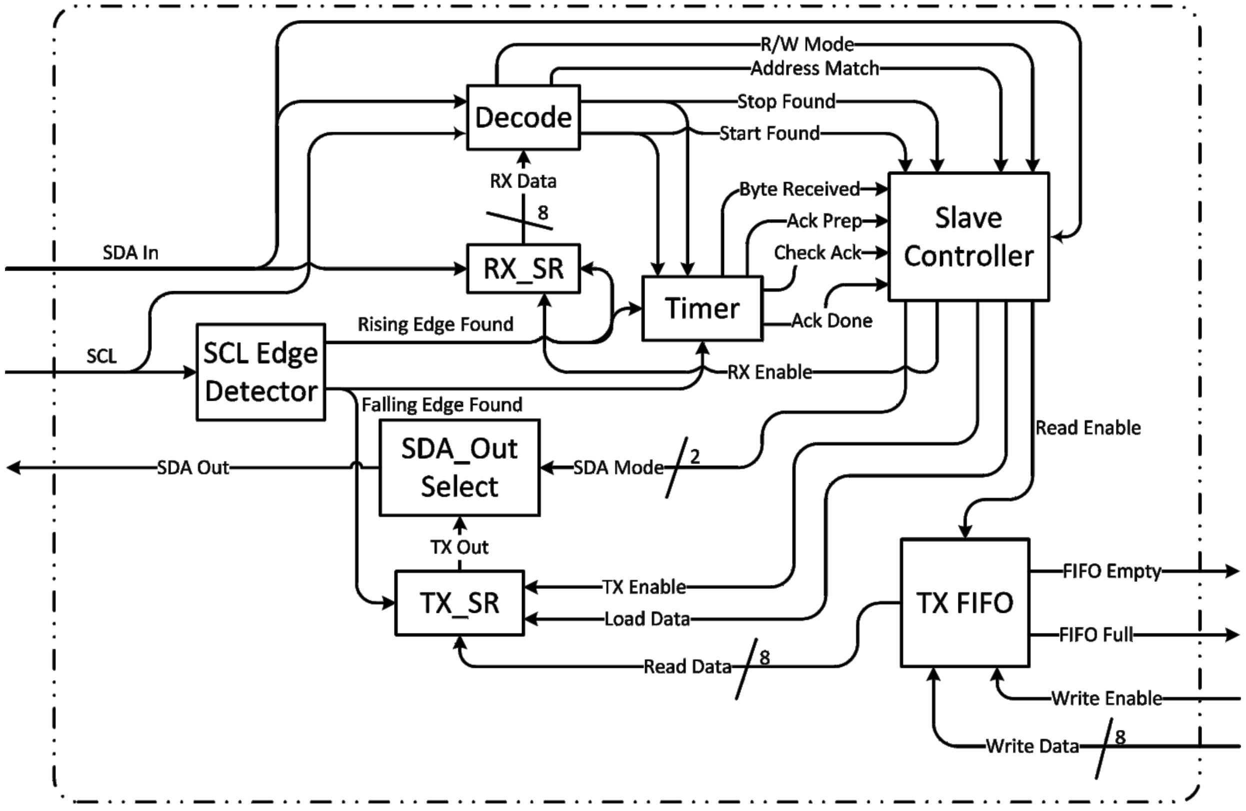

Figure 9: Transmit-Only I^{2}C Slave Architecture Diagram

6.2.1. Differences from a full I2C Slave device

As you can see, what you are going to be designing is very similar to full I^2C Slave device described in section 6.1. The only features discussed in section 5 that you will not be supporting are the ability to suppress the SCL line and the ability to accept data from the master, other than the byte containing the address and transfer mode information. **When your design gets a request to receive data from the master (Write mode), it must NACK the request.** Because of this, the only differences are that the controller logic is somewhat simpler than for a complete I^2C device and there is no RX FIFO block since there is no need to store received data for later sending to the user of the I^2C Slave device.

6.2.2. List of the Functional Units

As a reminder the purposes of each of the blocks in the presented architecture are listed below.

- **SCL Edge Detector**: This block detects both falling and rising edges on the SCL input and sends the results to the various blocks that depend on those edges.
- **SDA_Out Selector**: This block selects which value to currently output on the SDA Out signal. (Choices are between the current transmit shift register's output, the ACK value, the NACK value, and the idle line value)
- **Decode**: This block decodes/detects START and STOP conditions, check the target address against the device's address, and outputs the requested transmission mode.
- **TX_SR**: This is the 8-bit transmitting shift register.
- **RX_SR**: This is the 8-bit receiving shift register.
- **TX FIFO**: This is the transmission FIFO queue and is used for storing the data bytes to send to the master upon request.
- **Timer**: This block maintains and relays the critical timing information based on the edges in the SCL line and START and STOP conditions.
- **Slave Controller**: This block controls the sequence of operations of the design, including handling when and how to ACK or NACK

6.3. Top-level File Description and Requirements

The top-level design file is where you must connect up all of the other functional unit design files. Additionally, you should build all synchronizers for top-level signals into the top-level design file. Alternatively, you may include them in the individual design files that would need the signal, although this is generally a poor design choice. As a reminder only the blocks specified by the COMPONENT_FILES makefile variable will be compiled during the source simulation of your design. Also, the entity (including port names) for the top-level design block must exactly match what is presented below, or your design will fail the grading tests.

6.3.1. Requirements

The required entity name is:	i2c_slave
The required filename is:	i2c_slave.sv

6.3.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic ‘0’), all registers/flip-flops in the device must reset to their initial values
scl	Input	The I^2C Bus Clock. **(300 ns period w/ 33.3% duty cycle)**
sda_in	Input	The current value of the SDA line
sda_out	Output	The value to drive on to the SDA line
write_enable	Input	This is the active high write enable for the transmission FIFO. When this signal is asserted (logic ‘1’) the data currently on the write_data bus is stored into the FIFO on the rising-edge of the system clock
write_data[7:0]	Input	This is the byte of data to be written to the transmission FIFO when the write_enable signal is asserted.
fifo_empty	Output	This is an active high flag that reports if the transmission FIFO is currently empty (logic ‘1’) or not (logic ‘0’).
fifo_full	Output	This is an active high flag that reports if the transmission FIFO is currently full (logic ‘1’) or not (logic ‘0’).

6.4. SCL Edge Detector Block Description

This functional unit is detects both rising and falling edges on the SCL line. Functional units that depend on these edges simply check the outputs of this block instead of doing their own edge detection logic.

6.4.1. Requirements

The required entity name is: scl_edge
The required filename is: scl_edge.sv

6.4.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops in the device must reset to their initial values
scl	Input	The I^2C Bus Clock. **(300 ns period w/ 33.3% duty cycle)**
rising_edge_found	Output	This is used to alert other blocks that a rising-edge occurred, by creating an active-high pulse, for 1 system clock cycle, following the detection of a rising-edge.
falling_edge_found	Output	This is used to alert other blocks that a falling-edge occurred, by creating an active-high pulse, for 1 system clock cycle, following the detection of a falling-edge.

6.5. SDA_Out Select Block Description

This functional unit selects what value to feed to the sda_out port of the design. The possible options are the current output value of the transmission shift register, the ACK value, the NACK value, and the idle line value. This block is controlled by the Slave Controller through a 2-bit bus.

6.5.1. Requirements

The required entity name is: sda_sel
The required filename is: sda_sel.sv

6.5.2. Port Descriptions

Signal	Direction	Description
tx_out	Input	The current output value of the transmission shift register.
sda_mode[1:0]	Input	This is the 2-bit control bus from the Controller. "00" -> Idle Line Value "01" -> ACK value "10" -> NACK value "11" -> tx_out value
sda_out	Output	The chosen value for the sda_out port of the slave device.

6.6. Decode Block Description

This block decodes/observes and reports when START and STOP conditions have occurred and checks the target address against the device's address and decodes the Read/Write mode for the request from the starting byte sent by the master. Your device's address will be "1111000" (this is written with the MSB on the left).

6.6.1. Requirements

The required entity name is: decode
The required filename is: decode.sv

6.6.2. Port Descriptions

Signal	**Direction**	**Description**
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops in the device must reset to their initial values
scl	Input	The current value of the SCL line
sda_in	Input	The current value of the SDA line
starting_byte[7:0]	Input	The byte received from the master with the target address and transfer mode information.
rw_mode	Output	This is used to tell the controller block what transfer mode was requested.
address_match	Output	This is used to tell the controller block that the master's target address matches the device's address.
stop_found	Output	This is used to tell the controller and the timer that a STOP condition has been detected.
start_found	Output	This is used to tell the controller and the timer that a START condition has been detected.

6.7. Receiving Shift Register (RX SR)

This is an 8-bit serial-to-parallel shift register that only shifts when a rising-edge has occurred on the SCL line, the rx_enable is active/asserted, and a rising-edge is occurring on the system clock.

Hint: Since this is simply a special case of the flexible serial-to-parallel shift register you designed in Lab 3, this should simply be a wrapper file that uses the that design, which is why that design file has been copied into your source folder by the setup script.

6.7.1. Requirements

The required entity name is:	rx_sr
The required filename is:	rx_sr.sv

6.7.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops must reset to their initial values
sda_in	Input	The current value of the SDA line
rising_edge_found	Input	An active-high pulse (1 system clock cycle in length) following a rising-edge on the SCL line.
rx_enable	Input	This is an active-high enable signal from the controller that allows the shift register to shift in the current value of the SDA line
rx_data[7:0]	Output	This is the current contents of the shift register.

6.8. Transmitting Shift Register (TX_SR)

This is an 8-bit parallel-to-serial shift register that only shifts when a falling-edge has occurred on the SCL line, the tx_enable is active/asserted, and a rising-edge is occurring on the system clock.

Hint: Since this is simply a special case of the flexible parallel-to-serial shift register you designed in Lab 3, this should simply be a wrapper file that uses the that design, which is why that design file has been copied into your source folder by the setup script.

6.8.1. Requirements

The required entity name is: tx_sr
The required filename is: tx_sr.sv

6.8.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops must reset to their initial values
tx_out	Output	The output value of the shift register
falling_edge_found	Input	An active-high pulse (1 system clock cycle in length) following a falling-edge on the SCL line.
tx_enable	Input	This is an active-high enable signal from the controller that allows the shift register to shift out the next value for the SDA line
tx_data[7:0]	Input	This is the byte that is going to be loaded and shifted out
load_data	Input	This is an active-high pulse (1 system clock cycle long) that tells the shift register to load in the data currently on the tx_data port.

6.9. Transmission FIFO Wrapper (TX_FIFO)

This is the block that will contain the transmission First-In-First-Out (FIFO) queue for storing the data to transmit from the device. However, you will not have to design a FIFO, and instead will just be using a FIFO design that exists in the ECE337_IP library. Thus this file will simply become a "wrapper" file to translate between the interface expected by the architecture and the interface provided by the used FIFO design.

6.9.1. Module Declaration for Course IP module Used

The module declaration for the FIFO module is:

```
module fifo
(
  input  wire r_clk,
  input  wire w_clk,
  input  wire n_rst,
  input  wire r_enable,
  input  wire w_enable,
  input  wire w_data,
  output wire r_data,
  output wire empty,
  output wire full
);
```

6.9.2. Requirements

The required entity name is: tx_fifo

The required filename is: tx_fifo.sv

6.9.3. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)** This should be connected to both the r_clk and w_clk ports of the fifo module.
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops must reset to their initial values
read_enable	Input	This is an active-high enable signal from the controller that tells the FIFO that the currently outputted value is being read and to advance to the next one on the rising-edge of the system clock.
read_data[7:0]	Output	The data of the FIFO entry currently selected for reading.
fifo_empty	Output	This is an active high flag that reports if the transmission FIFO is currently empty (logic '1') or not (logic '0').
fifo_full	Output	This is an active high flag that reports if the transmission FIFO is currently full (logic '1') or not (logic '0').
write_enable	Input	This is the active high write enable for the transmission FIFO. When this signal is asserted (logic '1') the data currently on the write_data bus is stored into the FIFO on the rising-edge of the system clock
write_data[7:0]	Input	This is the byte of data to be written to the transmission FIFO when the write_enable signal is asserted.

6.10. Timer

The purpose of this block is to manage all of the critical timing information that is based on the START and STOP conditions and the SCL edges. It should track the bits either sent from or received by the slave device and reporting the timing information for properly handling the ACK phase following the transmission of a data byte.

Hint: Since the timer is effectively just tracking the number of rising and falling edges on SCL that have happened, you can quickly and easily build this using your flexible counter design from lab 3.

6.10.1. Requirements

The required entity name is: timer
The required filename is: timer.sv

6.10.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops must reset to their initial values
rising_edge_found	Input	An active-high pulse (1 system clock cycle in length) following a rising-edge on the SCL line.
falling_edge_found	Input	An active-high pulse (1 system clock cycle in length) following a falling-edge on the SCL line.
stop_found	Input	This is used to tell the timer that a STOP condition has been detected.
start_found	Input	This is used to tell the timer that a START condition has been detected.
byte_received	Output	This is used to tell the Controller that either a full byte of data has been received from the master or that a full byte has been transmitted to and received by the master.
ack_prep	Output	This is used to tell the Controller that, if it is going to be either ACK'ing or NACK'ing, it needs to start it now.
check_ack	Output	This is used to tell the Slave Controller that value of the current ACK phase should be stable and can be checked.
ack_done	Output	This is used to tell the Slave Controller that the ACK phase is done.

6.11. Slave Controller

The purpose of this block is to manage the sequence of operations and control signals for the device. It decides what values to assign to the various control signals so that currently needed operation will be done, as well as determining what the next operation should be.

6.11.1. Requirements

The required entity name is: controller
The required filename is: controller.sv

6.11.2. Port Descriptions

Signal	Direction	Description
clk	Input	The system clock. **(100 MHz)**
n_rst	Input	This is an asynchronous, active-low system reset. When this line is low (logic '0'), all registers/flip-flops must reset to their initial values
stop_found	Input	Used to tell the controller that a STOP condition has been detected.
start_found	Input	This is used to tell the controller that a START condition has been detected.
byte_received	Input	This is used to tell the Controller that either a full byte of data has been received from the master or that a full byte has been transmitted to and received by the master.
ack_prep	Input	This is used to tell the Controller that, if it is going to be either ACK'ing or NACK'ing, it needs to start it now.
check_ack	Input	This is used to tell the Slave Controller that value of the current ACK phase should be stable and can be checked.
ack_done	Input	This is used to tell the Slave Controller that the ACK phase is done.
rw_mode	Input	This is used to tell the controller block what transfer mode was requested.
address_match	Input	This is used to tell the controller block that the master's target address matches the device's address.
sda_in	Input	The current value of the SDA line
rx_enable	Output	This is an active-high enable signal from the controller that allows the shift register to shift in the next value.
tx_enable	Output	This is an active-high enable signal from the controller that allows the shift register to shift out the next value.
read_enable	Output	This is an active-high enable signal from the controller that tells the FIFO that the currently outputted value is being read and to advance to the next one on the rising-edge of the system clock.
sda_mode[1:0]	Output	This is the 2-bit control bus to the SDA_Out selector. See section 6.5.2.
load_data	Output	This is active-high pulse (1 system clock cycle long) that tells the transmit shift register to load in the data currently on the tx_data port.

6.12. Simulating the I2C Bus in your Test Bench

In order for you have the correct I^{2}C Bus behavior modeled in your test bench we have provided you an I^{2}C bus module in the ECE337_IP library. This module accurately models the true open-drain bus behavior, in addition to the 0's priority nature needed for I^{2}C to work. Also the module is designed to use scalable vectors for its ports, where their size is based on the parameter "NUM_DEVICES". This is so that the module can be naturally expanded during port mapping to connect a variable amount of devices. The assignment in the declaration sets the default number of devices to be two, which is all you should need for your test bench. Thus you do not need to assign this parameter's value during port mapping at all, simply port map it like any other module. Also, you should use the same index value when connecting your signals to module's ports, as shown in figure 10 (i.e. connect all of your test bench's master device signals to bit 0 of their respective ports on the module and connect all of your design's ports to bit 1 of their respective ports on the module).

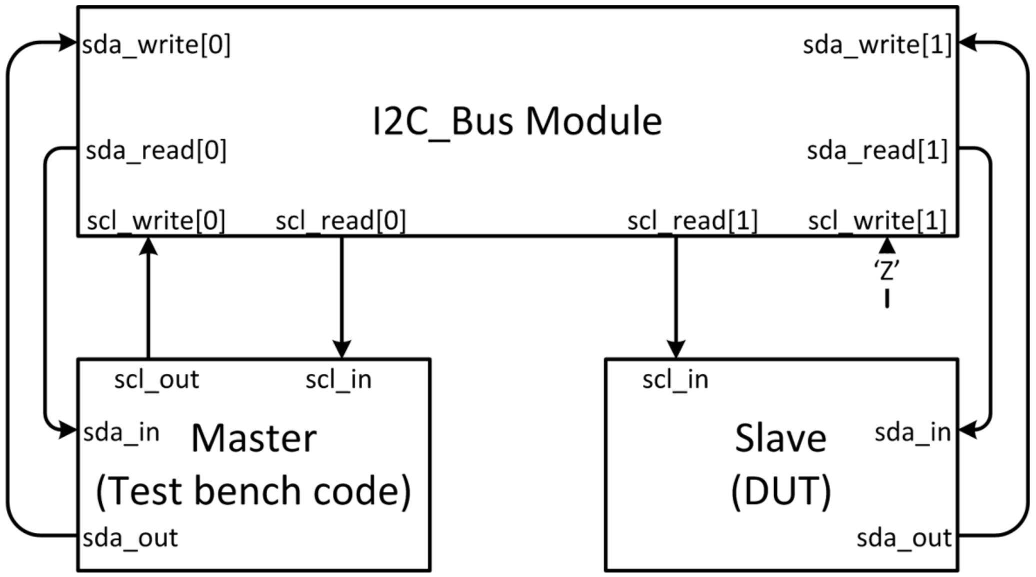

Figure 10: I2C Bus Module Wiring Diagram for use with Two Devices

6.12.1. Port Descriptions

Signal	Direction	Description
scl_read[*:0]	Output	The current value of the SCL bus (should be connected to the scl_in port for a device)
scl_write[*:0]	Input	The value being driven onto the SCL bus by a device (should be connected to the scl_out signal for the master or set to always be floating 'Z' for the slave)
sda_read[*:0]	Output	The current value of the SDA bus (should be connected to the scl_in port for a device)
sda_write[*:0]	Input	The value being driven onto the SDA bus by a device (should be connected to the sda_out port for a device)