COLLEGE OF ENGINEERING

Department of Computer Engineering and Computer Science

I^2C Protocol On FPGA

By

Chanartip Soonthornwan (014353883)

On October 24, 2018

Table of Contents

Table of Appendices

Table of Figures

Table of Tables

I²C Protocol

1 Introduction

I^{2}C or Two-wire interface (TWI) is very popular and power protocol for communication between multiple masters and slaves which only require two wires, SCL and SDA, connected on each device. A master initiates all communication on the wires which means to supply the clock to slaves, and the slave will never initiate a communication. The I^{2}C interface can operate up to 400Kbits/second, but for this project there would be a master and it will operate to the standard of 100Kbits/ second.

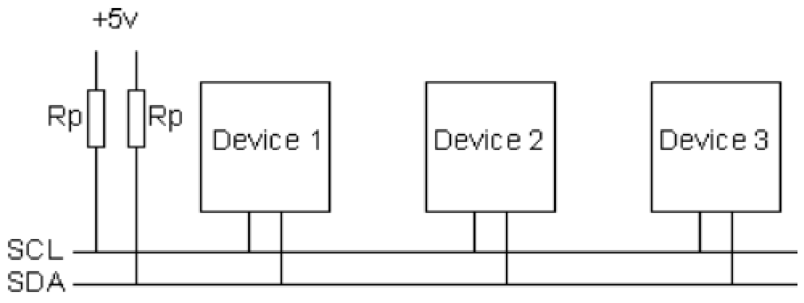


Figure 1: Example of I^{2}C Bus

2 Features

- A master and multiple slave operation
- Requires only two pins to interface I^{2}C bus (SCL and SDA)
- Support standard rate 100 Kbps

3 General I^{2}C Operation

When the I^{2}C master demands to communicate to a slave, the master initiates transmission by pulling the SDA bus LOW while SCL is still HIGH to indicating the START condition (Figure 2). Then, shifting one bit at the time from the most significant bit (MSB) to the least significant bit (LSB) of the address of the slave that the master wants to communicate

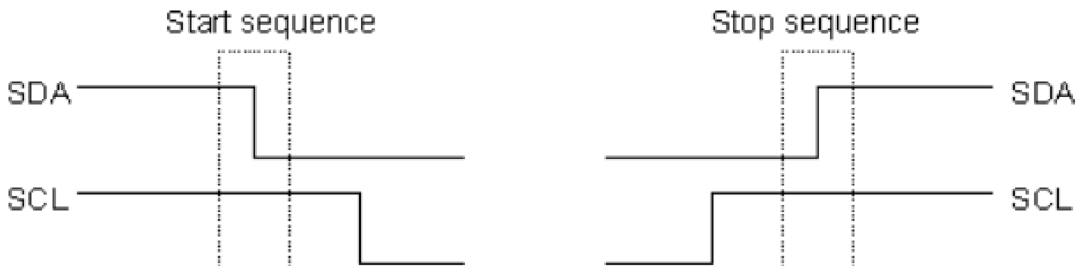


Figure 2: START and STOP condition

following by a read or write bit. For this I^{2}C version, the master will be only writing. After shifting the address and the write bit, the master will wait for an acknowledge from the slave by releasing control of the SDA line. On the other hands, the slave receiving the address and check if it is its address, then the slave will seize the SDA control and output LOW as an acknowledge. Once the master receives the acknowledge, the master transmits an 8-bit data then waits for another acknowledge before ending the communication by sending a STOP condition. The writing communication is shown in Figure 3.

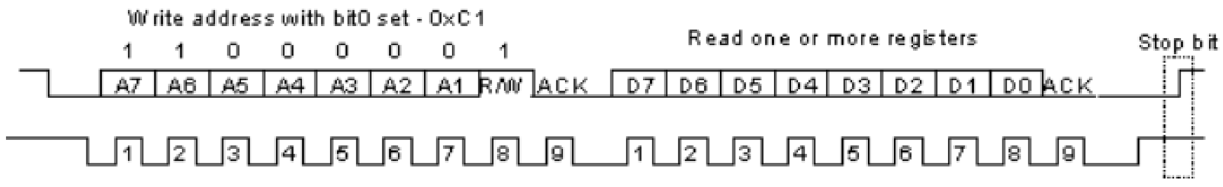

Write address with bit0 set - 0xC1
1 1 0 0 0 0 0 1
A7 A6 A5 A4 A3 A2 A1 R/W ACK
Read one or more registers
D7 D6 D5 D4 D3 D2 D1 D0 ACK
Stop bit
1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9

Figure 3: I^{2}C master to slave communication

4 Data Validity

A bit of data (either address or data) is transmitting during a SCL clock pulse. Therefore, the data to be transmitted must be ready while SCL is LOW and has to be stable during HIGH time (Figure 4). In other words, the SDA line should be set when SCL is LOW because if SDA is LOW when SCL is HIGH, it will trigger a START or STOP condition instead. After sending 8-bit data, an acknowledge should be sent from

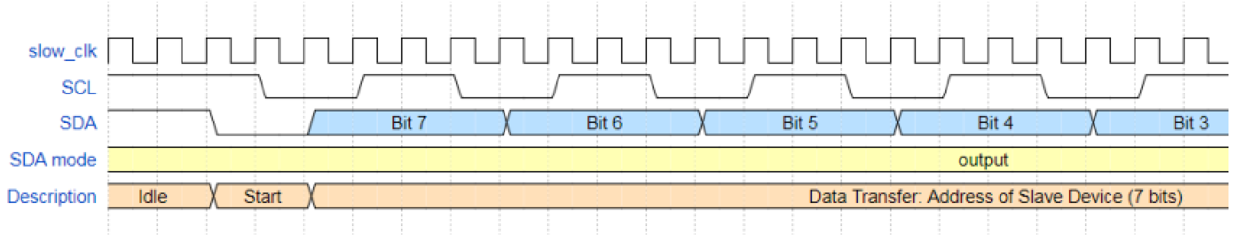


Figure 5: SDA is set while SCL is LOW and remain stable while SCL is HIGH.

a slave under the same condition, the acknowledge should be ready before the rising edge of SCL and stable while SCL is HIGH (Figure 5).

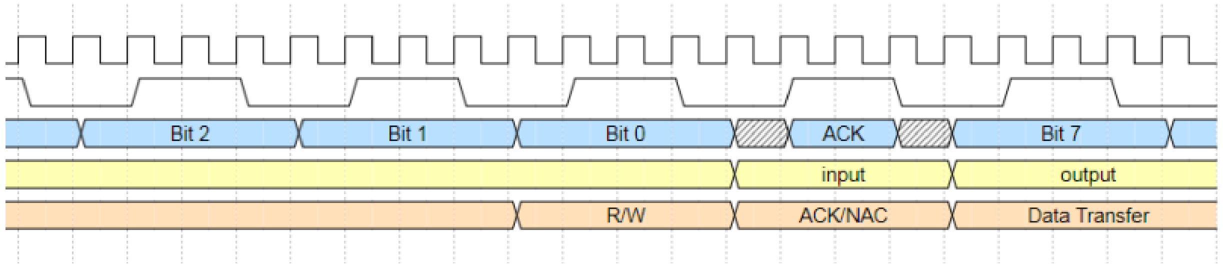


Figure 4: Acknowledge from a slave

5 Design Methodology

I^{2}C module implements a Moore State Machine to control the inputs, outputs, and changes at the right event. The intermediate module interprets and translates the controls or commands into bit-level

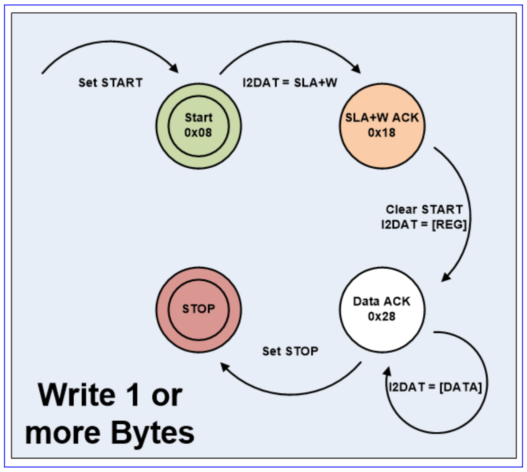


Figure 6: I^{2}C simple state diagram

data for sending SCL and SDA lines. The state diagram in figure 6 below explains an understandable chart while the state is more complex behind the scene.

6 Verilog

I^{2}C wires traditionally are bidirectional pins, but only SCL are commonly used as an output pin since multi-master device case is rare, such that SDA is the only bidirectional pin used for outputting and receiving data. Since SDA is operating serially (one-bit at the time), the FPGA must be able to switch the characteristic of the pin between output and input state. In Verilog, it could be done by utilizing an output enable variable. The output-enable variable acts like a tri-state buffer which takes control of the SDA pin when it is enable or releases the control the pin by setting high impedance to the pin to allow another device to take control over the SDA line.

Sample snippet code	

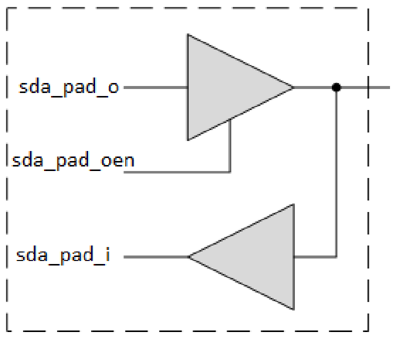
input sda_pad_oen;
inout SDA;
reg   sda_pad_o;
assign SDA = (sda_pad_oen) ? sda_pad_o : 1'bz;
assign sda_pad_i = (~sda_pad_oen) ? SDA : 1'bz;
	

Table 1: Bidirectional Verilog code and Diagram

SDA is an output when sda_pad_oen is set. Meaning the program takes controls the SDA line and transmit data from sda_pad_o out. Otherwise, the program sets SDA on high impedance state. Meanwhile, when sda_pad_oen is disable and the SDA is at high impedance state, the program allows any change on SDA line to store to sda_pad_i, so a user could utilize the value on sda_pad_i (i.e. the value on SDA).

7 I^{2}C for Garbage Collector

The I^{2}C for Garbage Collector is customized specifically for this project as the requirement of the project is to use the I^{2}C master on a FPGA to be a half-duplex protocol which only transmitting data to a destination slave that does not have a special register on it. In other words, whatever data input to the slave will be the data that it uses on their process right away. The SCL clock provides 100KHz or 10 us period times while transmitting the data. Since the FPGA has on-board a 100 MHz oscillator, a clock divider is implemented to create a slow clock which is four times faster than the required SCL clock or 400KHz slowed clock. The reason that slowed clock has to be four times faster than a period of SCL is that a state on the state diagram above could be divided into four states, therefore, the state machine has a full control over data path and control path for the I^{2}C device (Figure 7).

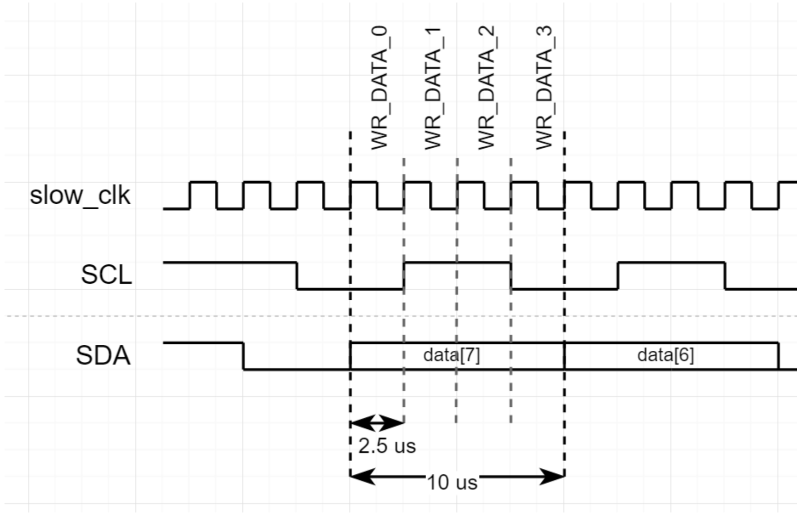


Figure 7: I^{2}C Timing Diagram with States

8 I^{2}C With a TramelBlaze

TramelBlaze is a 16-bit emulator of an 8-bit software microprocessor, PicoBlaze. The TramelBlaze will collect 7-MSB-bit on a FPGA, Nexys4, switches as an address and 8-LSB-bit of the switches as an 8-bit data for the I^{2}C to transmit to a slave device. Since the switches are inputs from the FPGA, the inputs go through the microprocessor before transmitting out to a slave through I^{2}C. The switches will be stored in the microprocessor's scratchpad RAM in order which are the address and then the data, and the microprocessor will output an address, a data, and a write command respectively.

To make the design less complicated, I2C_Interface is implemented to reduce the intermediate connection wires between FPGA's inputs, microprocessor's I/O, I^{2}C's I/O, and other small modules. The interface contains registers that hold immediate values of a slave's address, data, and i^{2}C start bit. Moreover, it sets and reset interrupt request for the microprocessor, and selects an input for the microcontroller to read.

9 Externally Developed Blocks

9.1 TramelBlaze

Description

A 16-bit microcontroller that emulates the 8-bit PicoBlaze utilizing 4Kx16 bit ROM as instruction memory where the processor reads and performs the assembly program. In this application, TramelBlaze utilizes UART engine to communicate with a Serial Terminal as to display and receive an ASCII value according to which port_id the processor preferred.

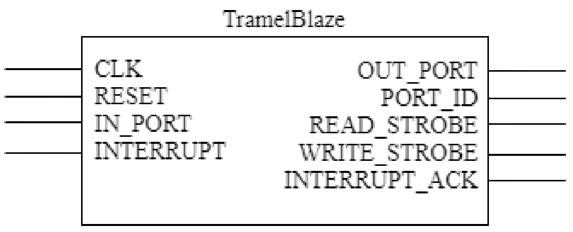


Figure 8: TramelBlaze Block Diagram

I/O

Signal	Size (bit)	I/O	Connected to
CLK	1	I	100MHz Crystal Oscillator
RESET	1	I	AISO_RST
INTERRUPT	1	I	RS_FLOP
IN_PORT	16	I	UART_TOP
OUT_PORT	16	O	UART_TOP
PORT_ID	16	O	Address Decoder
INTERRUPT_ACK	1	O	RS_FLOP
READ_STROBE	1	O	UART_TOP
WRITE_STROBE	1	O	UART_TOP

Table 2: TramelBlaze I/O

10 Internally Developed Blocks

10.1 I2C Top Level Design

Description

The top level demonstrates interconnections of controls and data paths through a I^{2}C core, a microprocessor (TramelBlaze), a I^{2}C interface, and an Asynchronized-In-Synchronized-Out Reset signal module (AISO_RST). The switches input will be sent through the I^{2}C interface before sending to the microprocessor and then sending to the I^{2}C core through the interface.

Block Diagrams

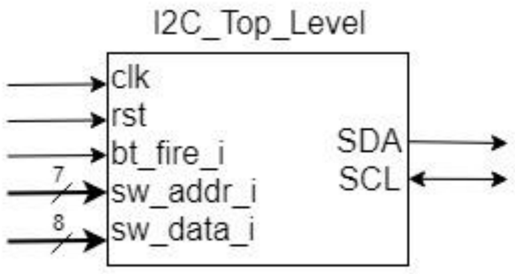


Figure 10: Top Level Block Diagram

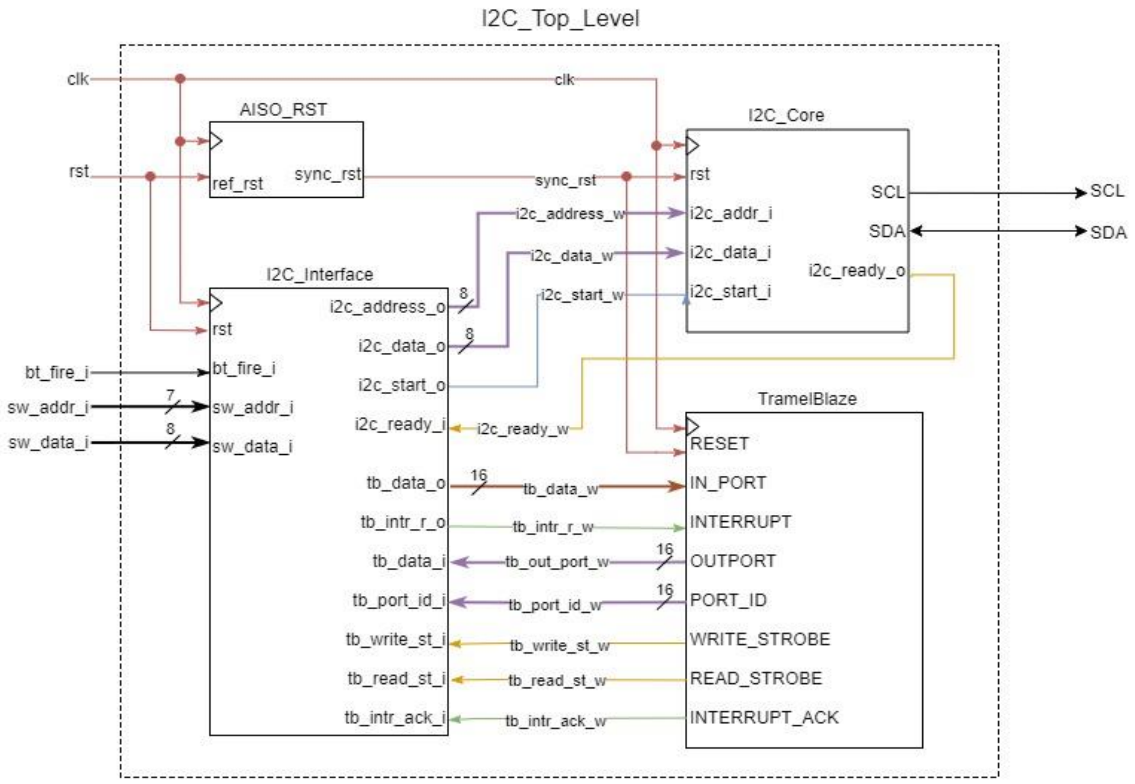


Figure 9: Top Level Detail Block Diagram

I/O

Signals	I/O	Size (bits)	Type	Description	Operation
clk	I	1	Digital	Provide 100 MHz clock frequency to the design	Generate 100 MHz clock
rst	I	1	Digital	Reset signal	Press button up to active
bt_fire_i	I	1	Analog	Start I2C transmission	Press button down to active
sw_addr_i	I	7	Analog	Slave address input	Toggle switches [15:9]
sw_data_i	I	8	Analog	Slave data input	Toggle switches [7:0]
SCL	O	1	Digital	Serial Clock Line	A bus line to slaves
SDA	IO	1	Digital	Serial Data Line	A bus line to slaves

Table 3: I^{2}C Top Level I/O Table

Source Code: Appendix a: I2C Top Level Source Code

Verification

To verify, the top level will be provided different switches input for the master to communicate on multiple slave devices. Then, the top level will receive bt_fire_i signal to simulate a physical push button on a FPGA, and the signal will be stabilized by a debounced module to create a digital pulse signal to operate the I^{2}C Core. The slave addresses are, 7'h01, 7'h72, 7'h55, and 7'h12. The slave devices were programmed by other designer which could successfully verify the communication because the slaves could response to the protocol as expected.

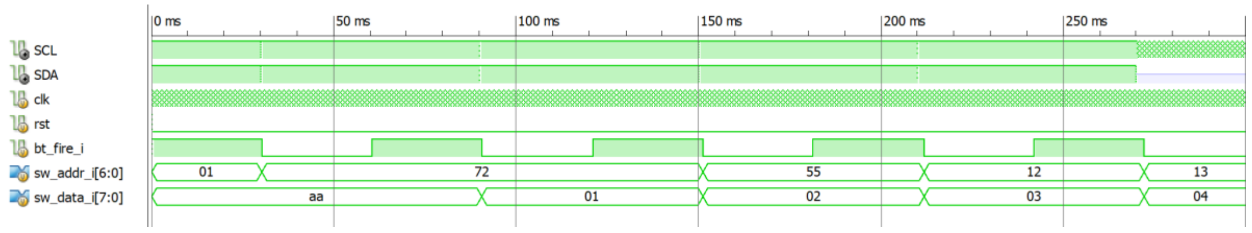


Figure 11: Top Level Verification

The bt_fire_i button is pushed for 30 ms before generating a one-pulse shot signal to The Tramelblaze to start the I^{2}C transmission. Each time a transmission is done, there is a change on switches for address and data to simulate that the I^{2}C master can communicate to multiple slaves, but there is a limit of this module which is the transmission would fall into a state of reading an acknowledge forever if the address that the I^{2}C is transmitted to an unknown slave device around the end of the simulation.

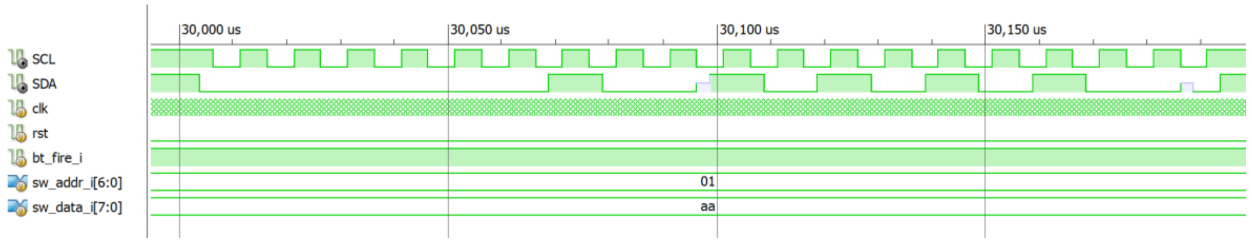


Figure 12: Top Level Complete Transmission

According to the state diagram (Figure 6), a complete transmission begins with a START condition which SDA line is pulled LOW following by SCL line, then transmits 7-bit data with a write bit (LOW) before waiting for an acknowledge from a slave. Since the slave is not created for specifically for this project,

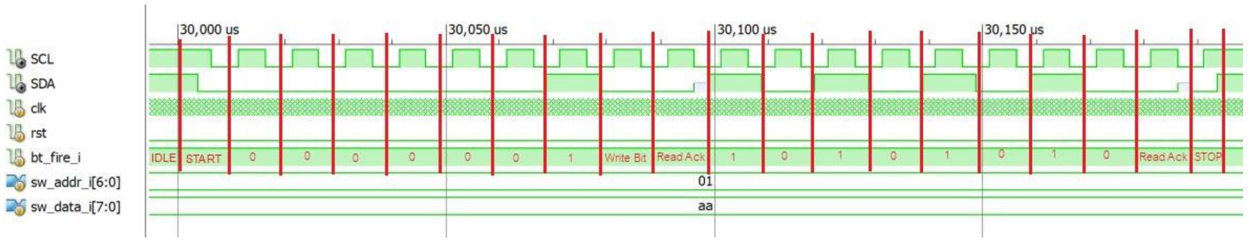


Figure 13: Top Level Complete Transmission Explanation

then there is a little red spike in the diagram, but the acknowledge bit from the slave has been read while the SCL is on HIGH time. Therefore, the Top Level could move to the next steps which are sending 8-bit of data, reading for another acknowledge, send a STOP bit, and come back to idle state as expected (show in Figure 12).

On the other hands, when the master fails by sending an address to a known slave, the master will be waiting for an acknowledge forever which could be fixed by adding a function to scope the reading for acknowledge duration in the next version.

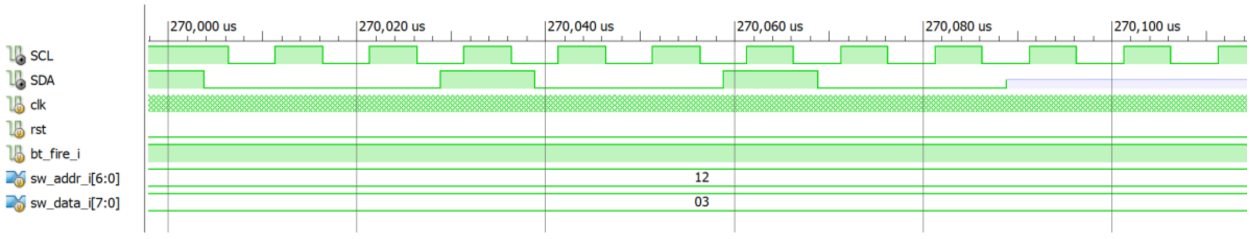


Figure 14: Top Level Incomplete Transmission

Verification Source Code: Appendix d: I2C_Top_Level Testbench

10.2 I2C_Core

Description

A master module of I^{2}C protocol which can initiate a communication between itself to other devices on the Two Wire Interface (TWI) buses. It contains a clock divider which generates a slow clock from on-board oscillator. The slow clock is used for driving states of a state machine that controls the input and output of the I^{2}C Core since the SCL and SDA line must be setup in sequences of each other.

Block Diagrams

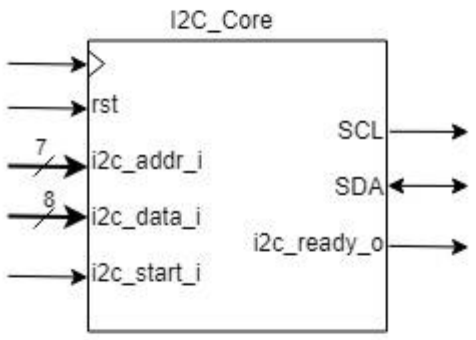


Figure 16: I2C Core Block Diagram

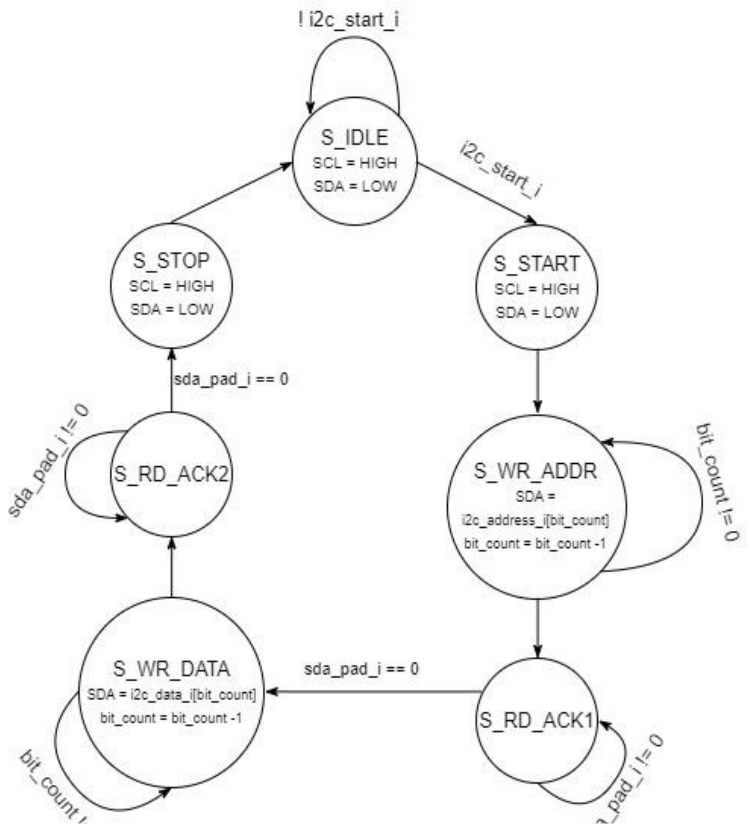


Figure 15: I2C Core State Diagram

I/O

Signals	I/O	Size (bits)	Type	Description	Operation
clk	I	1	Digital	Provide 100 MHz clock frequency to the design	Generate 100 MHz clock
rst	I	1	Digital	Reset signal	Receive synchronous reset signal from AISO_RST
i2c_addr_i	I	7	Digital	Address to transmit	Receive Address from I2C_Interface
i2c_data_i	I	8	Digital	Data to transmit	Receive Data from I2C_Interface
i2c_start_i	I	1	Digital	I2C_Core Start signal	Receive Start signal from I2C_Interface
i2c_ready_o	O	1	Digital	I2C_Core Ready signal	Send Ready to I2C_Interface 1: ready / 0: busy

Table 4: I²C Core I/O table

Source Code: Appendix b: I2C_Core Source Code

Verification

The I2C_Core will be provided an address, data, and a start signal to start the communication, and it will be provided a simulated acknowledgement from a slave at the READ_ACK state, such that the state machine, SCL and SDA signals transition, and the clock divider could be observed and verified.

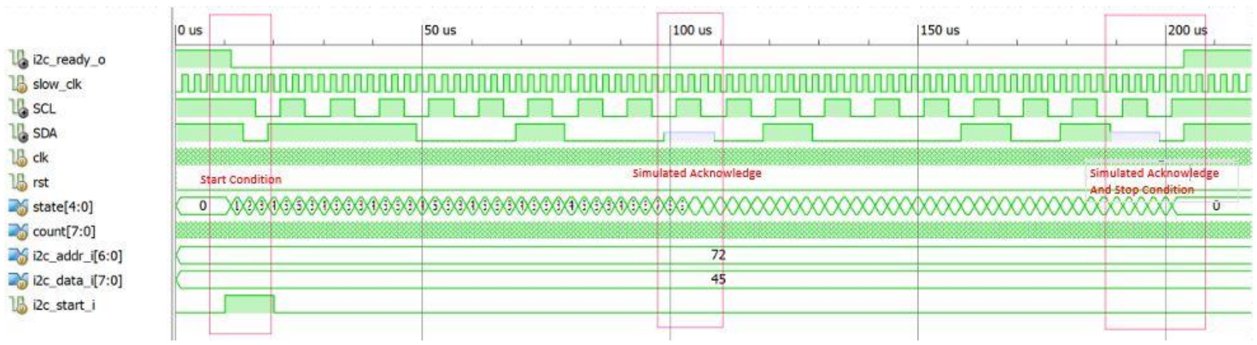


Figure 17: I2C_Core Verification with Simulated Acknowledgement

As shown in Figure 16, the I2C_Core completes a communication to a slave that has the address of 7'h72 by sending a start condition, sending the 7'h72 address, receiving simulated acknowledge, sending data, receiving another simulated acknowledge, and then sending a stop condition. This could verify that I2C_Core's state machine is successfully transitioning from START to STOP states.

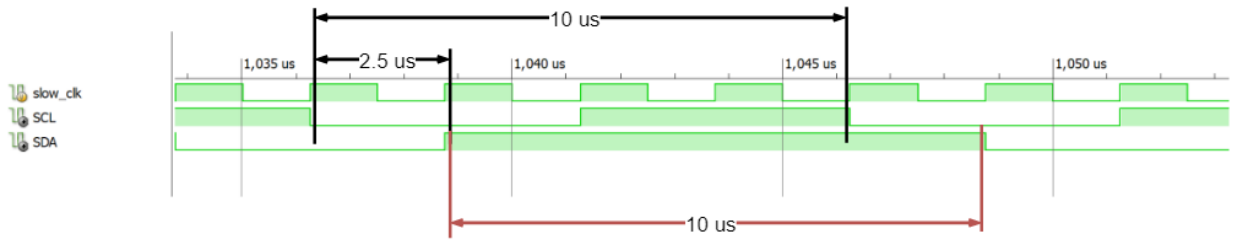


Figure 18: I2C_Core Timing Diagram

According to the timing calculation, slow clock should have 2.5 us period as this equation:

$$Slow\ clock\ period = \frac{Period_{TargetClock}}{Period_{OnBoardClock}} = \frac{\frac{1}{400KHz}}{\frac{1}{100MHz}} = 2.5\ us$$

Also, I^{2}C timing standard requirement is 100Kbit per second or 10 us that can be calculated by this equation:

$$I^2C\ output\ period = \frac{Period_{StandardClock}}{Period_{OnBoardClock}} = \frac{\frac{1}{100KHz}}{\frac{1}{100MHz}} = 10\ us$$

Then, the slow clock, SCL, and SDL timings are verified as they are 2.5 us, 10 us, and 10 us period times respectively according to the simulation.

Verification Source Code: Appendix e: I2C_Core Testbench

10.3 I2C Interface

Description

The module which is an interface of on-board inputs, outputs, and interconnection wires between I2C_Core and TramelBlaze. I2C Interface holds immediate I^{2}C address, data, and start signal for I2C_Core. The signals are reset when I2C_Core is ready for the next operation. The interface also debounce (or stabilize) a physical on-board push button's signal and select which source of input for the TramelBlaze when it reads an input.

Block Diagrams

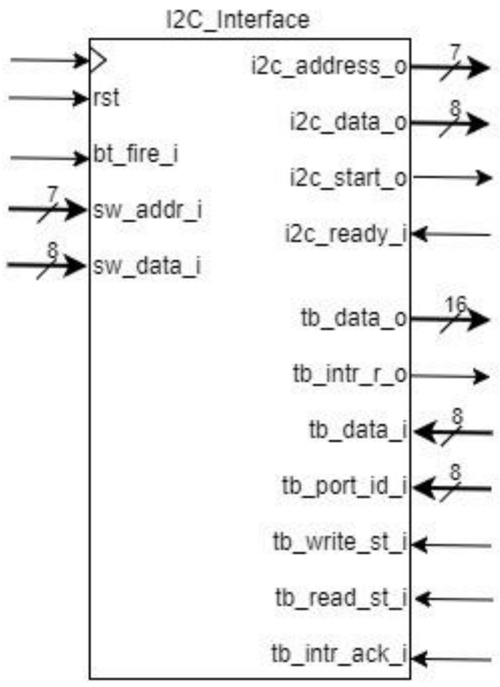


Figure 19: I2C_Interface Block Diagram

I/O

Signals	I/O	Size (bits)	Type	Description	Operation
clk	I	1	Digital	Provide 100 MHz clock frequency to the design	Generate 100 MHz clock
rst	I	1	Digital	Reset signal	Press button up to active
bt_fire_i	I	1	Analog	Start I2C transmission	Press button down to active
sw_addr_i	I	7	Analog	Slave address input	Toggle switches [15:9]
sw_data_i	I	8	Analog	Slave data input	Toggle switches [7:0]
i2c_addr_o	O	8	Digital	Address to transmit	Send Address to I2C_Core
i2c_data_o	O	8	Digital	Data to transmit	Send Data to I2C_Core
i2c_start_o	O	1	Digital	I2C_Core Start signal	Send Start signal to I2C_Core
i2c_ready_i	I	1	Digital	I2C_Core Ready signal	Receive Ready from I2C_Core 1: ready / 0: busy

tb_port_id_i	I	16	Digital	Port ID input	Receive Port ID from TB
tb_data_i	I	16	Digital	Data input	Receive Data from TB
tb_write_st_i	I	1	Digital	Write Strobe input	Receive Write Strobe from TB
tb_read_st_i	I	1	Digital	Read Strobe input	Receive Read Strobe from TB
tb_intr_ack_i	I	1	Digital	Interrupt acknowledge input	Receive interrupt acknowledge from TB
tb_intr_r_o	O	1	Digital	Interrupt request output	Send interrupt request to TB
tb_data_o	O	16	Digital	Data output	Send data to TB

Table 5: I2C_Interface I/O Table

Source Code: Appendix c: I2C_Interface Source Code

Verification

I2C_Interface typically operates by combinational logic from inputs and outputs from TrambelBlaze(TB), I2C_Core, and the FPGA. It is difficult to simulate the output without them, but I2C_Interface could be breakdown to three parts; reading to TB, writing from TB, and clearing registers.

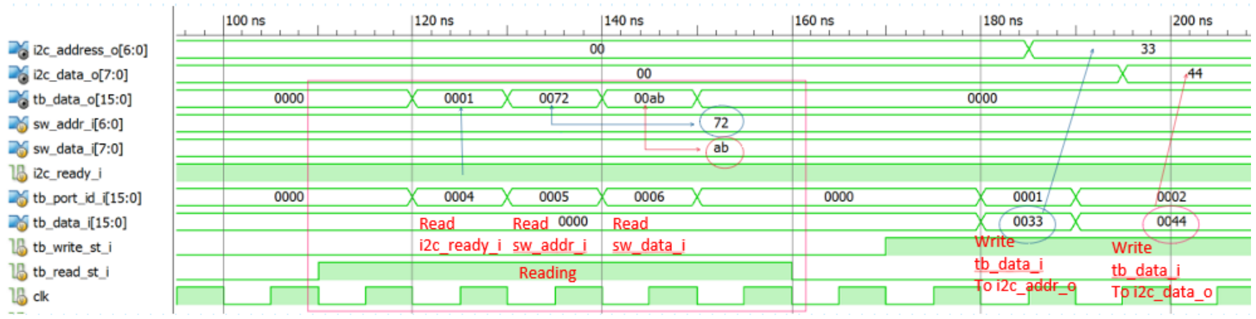


Figure 20: I2C_Interface Reading and Writing phrase

Reading to TB, when TramelBlaze's read strobe (tb_read_st_i) is HIGH, it is indicating that the TB wants to read the address and data from switch inputs or reading the I2C_Core status through the port ID of the input it wants to read. As shown in Figure 20, when Read Strobe is HIGH and the port ID state 16'h0004, it means the TB is reading i2c_ready_i. Since the ready signal is HIGH, then data output to TB (tb_data_o) is HIGH at LSB (16'h0001). Then, port id is 16'h0005 means that TB is reading sw_addr_i, so tb_data_o gets 16'h0072. After that, tb_data_o gets 16'h00ab when port id is 16'h0006.

Writing from TB, similar fashion to the reading part, if port id is 16'h0001, I2C_Interface will pass the input from TB (tb_data_i) to I2C_Core address input (i2c_addr_i). If port id is 16'h0002, it will pass tb_data_i to I2C_Core data input (i2c_data_i). Lastly, if port id is 16'h 0003, it will write a start signal to the I2C_Core (i2c_start_i).

Clearing registers, the data in i2c_addr_i and i2c_data_i will be cleared at the rising edge of the Ready signal. In other word, the data is cleared when I2C_Core finishes a transmission. In order to set the I2C_Core busy, the write button (bt_fire_i) will be HIGH for some amount of time in digital circuit, but it is roughly 30 milliseconds. During the times, there is a debounce module to stabilize the button input and pulse a clock-period long signal indicating that the button is pressed. When the pulse is set, it will trigger an interrupt request (tb_intr_r_o) to TB to call its software interrupt service routine.

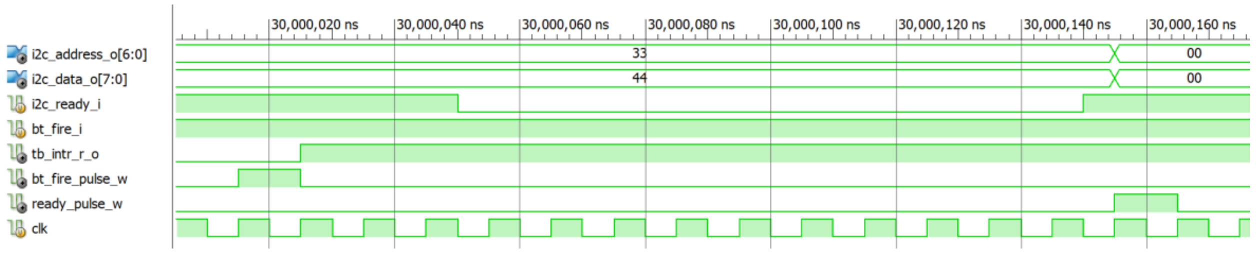


Figure 21: I2C_Interface debouncing button input and clearing registers

Since the switch inputs are sent to TB and registers read data from TB regarding on the port_id successfully, the interrupt request is corresponded to the push button and the debounce module correctly, and the registers are reset when I2C_Core is become ready, then I2C_Interface is verified.

Verification Source Code: Appendix f: I2C_Interface Testbench

Appendix a: I2C Top Level Source Code

```
`timescale 1ns / 1ps
//*************************************************************************//
//    Class:         CECS490B Senior Projects                              //
//    Project name:  Garbage Collector                                     //
//    File name:     I2C_Top_Level.v                                       //
//                                                                         //
//    Created by Chanartip Soonthornwan on October 3, 2018.                //
//    Copyright @ 2018 Chanartip Soonthornwan. All rights reserved.        //
//                                                                         //
//    Abstract:      Display structural of Microprocessor (Tramel Blaze)   //
//                   and Communication Protocol Device (I2C Core)          //
//                   utilizing Interface for complex design.               //
//*************************************************************************//
//  Version 1.1 (October 12, 2018)
//      - Added sw_addr_i and sw_data_i for interacting to a user input
//          to select slave address and data to send
//      - Added tb_port_i to allow input into INPORT of TramelBlaze
//
//  Version 1.0 (October 3, 2018)
//
module I2C_Top_Level(
    input  wire clk,            // System clock (100MHz)
    input  wire rst,            // System reset (from button up)
    output wire SCL,            // I2C Clock wire
    inout  wire SDA,            // I2C Data wire

    input wire       bt_fire_i, // button down to fire a transmission
    input wire [6:0] sw_addr_i, // switches[15:9] for 7-bit address input
    input wire [7:0] sw_data_i  // switches[7:0] for 8-bit data input
);
    wire        sync_rst;       // Synchronized reset signal wire

    wire        tb_intr_r_w;    // Interrupt Request from I2C_Interface to TB
    wire        tb_intr_ack_w;  // Interrupt Acknowledge from TB to I2C_Interface
    wire [15:0] tb_out_port_w;  // Output       from TB to I2C_Interface
    wire [15:0] tb_port_id_w;   // PortID       from TB to I2C_Interface
    wire        tb_wr_st_w;     // Write Strobe from TB to I2C_Interface
    wire        tb_rd_st_w;     // Read Strobe  from TB to I2C_Interface
    wire [15:0] tb_data_w;      // Data output  from I2C_Interface to TB

    wire [ 6:0] i2c_addr_w;     // Address      from I2C_Interface to I2C_Core
    wire [ 7:0] i2c_data_w;     // Data         from I2C_Interface to I2C_Core
    wire        i2c_start_w;    // Start signal from I2C_Interface to I2C_Core
    wire        i2c_ready_w;    // Ready signal from I2C_Core to I2C_Interface

    // Communication Protocol Device
    //  Generating output clock signal on SCL line with frequency of 100KHz
    //  while transmitting 8-bit address and data. Also indicating a ready
    //  signal to microprocessor before initiating the next transmission.
    I2C_Core i2c_core (
    .clk          (clk            ),
    .rst          (sync_rst       ),
    .i2c_addr_i   (i2c_addr_w     ),
    .i2c_data_i   (i2c_data_w     ),
    .i2c_start_i  (i2c_start_w    ),
    .i2c_ready_o  (i2c_ready_w    ),
    .SCL          (SCL            ),
    .SDA          (SDA            )
    );

    // Interface
    //  Wrapping all wires, generating ready pulse signal, setting/resetting
    //  interrupt request, and holding Address and Data in registers
    //  to be ready before initiating an I2C transmission.
    I2C_Interface i2c_interface(
    .clk          (clk          ),
    .rst          (sync_rst     ),
    .bt_fire_i    (bt_fire_i    ),
    .sw_addr_i    (sw_addr_i    ),
    .sw_data_i    (sw_data_i    ),
    // I2C I/O
    .i2c_ready_i  (i2c_ready_w  ),
    .i2c_address_o(i2c_addr_w   ),
    .i2c_data_o   (i2c_data_w   ),
    .i2c_start_o  (i2c_start_w  ),
    // Tramel Blaze I/O
    .tb_port_id_i (tb_port_id_w ),
    .tb_data_i    (tb_out_port_w),
    .tb_write_st_i(tb_wr_st_w   ),
    .tb_read_st_i (tb_rd_st_w   ),
    .tb_intr_ack_i(tb_intr_ack_w),
    .tb_intr_r_o  (tb_intr_r_w  ),
    .tb_data_o    (tb_data_w    )
    );

    // Microprocessor
    //  Created by John Tramel
    //  Utilizing Scratch RAM to load instructions
    //  and execute them before initiates the transmission
    //  *note: In this version, there is only writing to
    //         but not reading from other devices.
    tramelblaze_top tb_top(
    .CLK          (clk          ),
    .RESET        (sync_rst     ),
    .IN_PORT      (tb_data_w    ),
    .INTERRUPT    (tb_intr_r_w  ),
    .OUT_PORT     (tb_out_port_w),
    .PORT_ID      (tb_port_id_w ),
    .READ_STROBE  (tb_rd_st_w   ),
    .WRITE_STROBE (tb_wr_st_w   ),
    .INTERRUPT_ACK(tb_intr_ack_w)
    );

    // Asynchronize-In-Synchronize-Out Reset
    //  In used for initiating synchronous reset signal
    //  to the whole design.
    AISO_RST aiso_rst(
    .clk      (clk     ),
    .ref_rst  (rst     ),
    .sync_rst (sync_rst)
    );

endmodule

//_________________________ ASIO Reset _____________________________________
//  Receives reset signal input from a reset button then generates
//  synchronized output at rising edge to other module in the design.
//  input:  clk - reference clock (on-board clock)
//  input:  ref_rst - reference reset (Asynchronized reset signal input)
//  output: sync_rst - synchronized reset
module AISO_RST(
    input  wire clk, ref_rst,
    output wire sync_rst
);
```

```
    reg  r1,r2;

    assign sync_rst = ~r2; // one-clock impulsive reset signal

    always@(posedge clk, posedge ref_rst) begin
        if(ref_rst) {r1, r2} <= 2'b00;
        else        {r1, r2} <= {1'b1, r1};
    end

endmodule
//_______________________ End ASIO Reset ___________________________________
```

Appendix b: I2C_Core Source Code

```
`timescale 1ns / 1ps
//*****************************************************************************//
//    Class:         CECS490B Senior Projects                                  //
//    Project name:  Garbage Collector                                         //
//    File name:     I2C_Core.v                                                //
//                                                                             //
//    Created by Chanartip Soonthornwan on October 3, 2018.                    //
//    Copyright @ 2018 Chanartip Soonthornwan. All rights reserved.            //
//                                                                             //
//    Abstract:      I2C at bit operation controlling data path and controls   //
//                   via a state machine with eight states.                    //
//*****************************************************************************//
// Version 2.0 (October 21, 2018)
//      - Remodeled Clock Divider to generate 400khz clock
//      - Remodeled State Machine with more states
//
//  Version 1.1 (October 12, 2018)
//      - Added sw_addr_i and sw_data_i for interacting to a user input
//          to select slave address and data to send
//      - Added tb_port_i to allow input into INPORT of TramelBlaze
//
//  Version 1.0 (October 3, 2018)
//
module I2C_Core(
    input wire clk,                   // On-board Clock 100MHz
    input wire rst,                   // Reset signal
    input wire [6:0] i2c_addr_i,      // Incoming Address
    input wire [7:0] i2c_data_i,      // Incoming Data
    input wire i2c_start_i,           // Start signal
    output reg i2c_ready_o,           // Ready Status Register

    output wire SCL,                  // I2C SCL (clock) output
    inout  wire SDA                   // I2C SDA (data) input/output
);

//_____________________________ IO Padder __________________________________
//
//        The middle ground for SDA and SCL bus to stop by since bidirectional
//        bus cannot be directly assigning the value. Therefore, there should be a
//        pad or middle ground taking place to make open-drain-like circuit.
//        The bus will be high impedance when no one takes control of the bus.
//        In order to control the bus, an output enable seize the control of
//        the bus. Once output is done, the output enable will be released, and
//        the bus from this device perspective will be high-impedance.
//        High-impedance is like open-drain which will allow incoming input when
//        there is an input from other devices.
//
    reg     scl_pad_o;                // i2c_clock output
    reg     scl_pad_oen;              // i2c_clock output enable

    wire    sda_pad_i;                // i2c_data input
    reg     sda_pad_o;                // i2c_data output
    reg     sda_pad_oen;              // i2c_data output enable

    assign SCL = (scl_pad_oen) ? scl_pad_o : 1'bz;
    assign SDA = (sda_pad_oen) ? sda_pad_o : 1'bz;
    assign sda_pad_i = (~sda_pad_oen)? SDA : 1'bz;

//____________________________ Clk Divider _________________________________
//
//  Generates slow clock (400khz) by ticking every 2.5 us, and each tick
//  will toggle the slow clock level from low to high and vice versa.
//  Calculation: DELAY = 100Mhz / 400Khz = 250
//               DELAY/2 = 125
    parameter  DELAY = 125;     // Constant for toggleling a clock at 1.25 us
    reg [ 7:0] count;           // Time counter
    reg        slow_clk;        // slow_clock from on_board clk
    wire       tick;            // Toggling the clock indicator

    assign tick = (count == (DELAY -1)); // Tick at 5us

    always@(posedge clk, posedge rst) begin
        if(rst)  begin
                count   <= 0;
                slow_clk <= 0;
        end else
            if(tick) begin                  // got a tick signal
                count   <= 0;               // reset the counter
                slow_clk <= ~slow_clk;      // switch the slow clock edge
        end else begin
                count   <= count + 8'b1;    // increment the counter
                slow_clk <= slow_clk;       // slow clock edge stay the same.
        end
    end

//____________________________ State Machine _________________________________
//
//  Generates controls and outputs according to an events of each state.
//  There are 20 states in total, but the main states are
//      IDLE -> START -> WRITE_ADDR -> READ_ACK
//                    -> WRITE_DATA -> READ_ACK2 -> STOP -> IDLE
//  Explain:
//  IDLE  - Both SCL and SDA are HIGH, and ready is HIGH indicating that the
//          I2C is ready for i2c_start_i signal. When i2c_start_i is HIGH,
//          ready is low as the I2C is busy.
//          slow_clk:  /~~\__/~~\__
//          SCL:       ~~~~~~~~~~~~
//          SDA:       ~~~~~~~~~~~~
//
//  START - First clock will pull SDA LOW while SCL HIGH. Second clock, both
//          SCL and SDA are LOW, and set the data register by concentrating
//          Address and Write bit before shifting on the next clock.
//          slow_clk:  /~~\__/~~\__/~~\__/
//          SCL:       ~~~~~~~~~~~~\______
//          SDA:       ~~~~~~\____________
//
//  WRITE - Shifting the data register from MSB to LSB. Each 4 clocks will
//          decrement the register index by one, and check if it is the LSB.
//          by utilizing command(CMD). If so, move to the read acknowledge.
//          slow_clk:  /~~\__/~~\__/~~\__/~~\__/
//          SCL:       ______/~~~~~~~~~~~\_____/
//          SDA:     ==X=======================X==
//
//  READ  - Wait for acknowledge input from a slave device. The input should be
//          ready before the rising edge of SCL and should not change while
//          the SCL is HIGH.
//          slow_clk:  /~~\__/~~\__/~~\__/~~\__/
//          SCL:       ______/~~~~~~~~~~~\_____/
//          SDA:     ==X=====\___________/======X==
//
//  STOP  - First clock will pull both SCL and SDA LOW, then SCL HIGH while
//          SDA LOW.
//          slow_clk:  /~~\__/~~\__/~~\__/
//          SCL:       ~~~~~~\_____/~~~~~~
```

```
//          SDA:       ~~~~~~\___________/
//
    localparam
        S_IDLE       =  0,
        S_START_0    =  1, S_START_1    =  2,
        S_WR_ADDR_0 =  3, S_WR_ADDR_1 =  4, S_WR_ADDR_2 =  5, S_WR_ADDR_3 =  6,
        S_RD_ACK1_0 =  7, S_RD_ACK1_1 =  8, S_RD_ACK1_2 =  9, S_RD_ACK1_3 = 10,
        S_WR_DATA_0 = 11, S_WR_DATA_1 = 12, S_WR_DATA_2 = 13, S_WR_DATA_3 = 14,
        S_RD_ACK2_0 = 15, S_RD_ACK2_1 = 16, S_RD_ACK2_2 = 17, S_RD_ACK2_3 = 18,
        S_STOP_0     = 19, S_STOP_1     = 20,

        CMD_IDLE     =  1, CMD_START    =  2,
        CMD_WR_ADDR =  3, CMD_RD_ACK   =  4,
        CMD_WR_DATA =  5, CMD_STOP     =  6,

        WRITE_BIT    = 1'b0,
        HIGH         = 1'b1,
        LOW          = 1'b0;

    // Registers
    reg [7:0] data_reg;      // Holding immediate data(address or data)
    reg [2:0] bit_count;     // Index for data_reg
    reg [2:0] cmd;           // Special command on some states
    reg [4:0] state;         // Present State
    reg [4:0] next_state;    // Next State

    // Update Present State on fast clock
    always@(posedge clk, posedge rst) begin
        if(rst) state <= S_IDLE;
        else    state <= next_state;
    end

    // State Machine
    always@(posedge slow_clk, posedge rst) begin
        if(rst) begin
            cmd         <= CMD_IDLE;
            scl_pad_oen <= HIGH;
            sda_pad_oen <= HIGH;
            scl_pad_o   <= HIGH;
            sda_pad_o   <= HIGH;
            data_reg    <= 0;
            bit_count   <= 0;
            next_state  <= S_IDLE;
            i2c_ready_o <= HIGH;
        end
        else begin
            case(state)
                // IDLE STATE
                //  SCL and SDA are high until recieve a start signal
                S_IDLE: begin
                    if(i2c_start_i) begin          // recieve start signal
                        cmd         <= CMD_START;
                        scl_pad_oen <= HIGH;
                        sda_pad_oen <= HIGH;
                        scl_pad_o   <= HIGH;
                        sda_pad_o   <= HIGH;
                        data_reg    <= data_reg;
                        bit_count   <= bit_count;
                        next_state  <= S_START_0;
                        i2c_ready_o <= LOW;         // now i2c is busy
                    end
                    else begin                      // still idleing
                        cmd         <= CMD_IDLE;
```

```
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= HIGH;
        sda_pad_o   <= HIGH;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_IDLE;
        i2c_ready_o <= HIGH;
    end
end

// START STATE
//  SDA is LOW while SCL is HIGH,
//  then SCL is LOW on the next clock
S_START_0: begin
        cmd         <= CMD_START;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= HIGH;
        sda_pad_o   <= LOW;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_START_1;
end
S_START_1: begin
        cmd         <= CMD_WR_ADDR;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= LOW;
        sda_pad_o   <= LOW;
        data_reg    <= {i2c_addr_i, WRITE_BIT}; // Loading data
        bit_count   <= 3'd7;                    // MSB Index
        next_state  <= S_WR_ADDR_0;
end

// WRITE STATE
//  Shifting MSB to LSB,
//  data in data_reg remains the same from clock 0 to 4,
//  decrement bit_count and move to the new state at
//  clock 4.
S_WR_ADDR_0: begin
        cmd         <= cmd;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= LOW;
        sda_pad_o   <= data_reg[bit_count];
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_WR_ADDR_1;
end
S_WR_ADDR_1: begin
        cmd         <= cmd;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= HIGH;
        sda_pad_o   <= sda_pad_o;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_WR_ADDR_2;
end
S_WR_ADDR_2: begin
        cmd         <= (bit_count == 0)? CMD_RD_ACK : cmd;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= HIGH;
        sda_pad_o   <= sda_pad_o;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_WR_ADDR_3;
end
S_WR_ADDR_3: begin
        cmd         <= cmd;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= HIGH;
        scl_pad_o   <= LOW;
        sda_pad_o   <= sda_pad_o;
        data_reg    <= data_reg;
        bit_count   <= bit_count -1;
        next_state  <= (cmd == CMD_RD_ACK)? S_RD_ACK1_0: S_WR_ADDR_0;
end

// READ STATE
//  Waiting for an acknowledge from a slave device.
//  Expected the ack signal to be ready before rising edge
//  of SCL clock. If the signal is valid, set data_reg with
//  i2c_data_i, and move to the next state.
//  Otherwise, come back to wait again.
//  **Note: This version doesn't have Artibute Lost**
S_RD_ACK1_0: begin
        cmd         <= cmd;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= LOW;
        scl_pad_o   <= LOW;
        sda_pad_o   <= sda_pad_o;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_RD_ACK1_1;
end
S_RD_ACK1_1: begin
        cmd         <= (sda_pad_i == LOW)? CMD_WR_DATA: CMD_RD_ACK;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= LOW;
        scl_pad_o   <= HIGH;
        sda_pad_o   <= sda_pad_o;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_RD_ACK1_2;
end
S_RD_ACK1_2: begin
        cmd         <= (sda_pad_i == LOW)? cmd: CMD_RD_ACK;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= LOW;
        scl_pad_o   <= HIGH;
        sda_pad_o   <= sda_pad_o;
        data_reg    <= data_reg;
        bit_count   <= bit_count;
        next_state  <= S_RD_ACK1_3;
end
S_RD_ACK1_3: begin
        cmd         <= cmd;
        scl_pad_oen <= HIGH;
        sda_pad_oen <= LOW;
        scl_pad_o   <= LOW;
        sda_pad_o   <= sda_pad_o;

        if(cmd == CMD_WR_DATA) begin
```

```
                data_reg   <= i2c_data_i;
                bit_count  <= 3'd7;
                next_state <= S_WR_DATA_0;
            end
            else begin
                data_reg   <= data_reg;
                bit_count  <= bit_count;
                next_state <= S_RD_ACK1_0;
            end
    end

    // WRITE STATE
    //  Similar to above state, but outputing data instead of
    //  the concentration of {slave address, write bit}
    S_WR_DATA_0: begin
            cmd         <= cmd;
            scl_pad_oen <= HIGH;
            sda_pad_oen <= HIGH;
            scl_pad_o   <= LOW;
            sda_pad_o   <= data_reg[bit_count];
            data_reg    <= data_reg;
            bit_count   <= bit_count;
            next_state  <= S_WR_DATA_1;
    end
    S_WR_DATA_1: begin
            cmd         <= cmd;
            scl_pad_oen <= HIGH;
            sda_pad_oen <= HIGH;
            scl_pad_o   <= HIGH;
            sda_pad_o   <= sda_pad_o;
            data_reg    <= data_reg;
            bit_count   <= bit_count;
            next_state  <= S_WR_DATA_2;
    end
    S_WR_DATA_2: begin
            cmd         <= (bit_count == 0)? CMD_RD_ACK : cmd;
            scl_pad_oen <= HIGH;
            sda_pad_oen <= HIGH;
            scl_pad_o   <= HIGH;
            sda_pad_o   <= sda_pad_o;
            data_reg    <= data_reg;
            bit_count   <= bit_count;
            next_state  <= S_WR_DATA_3;
    end
    S_WR_DATA_3: begin
            cmd         <= cmd;
            scl_pad_oen <= HIGH;
            sda_pad_oen <= HIGH;//(cmd == CMD_RD_ACK)? LOW: HIGH;
            scl_pad_o   <= LOW;
            sda_pad_o   <= sda_pad_o;
            data_reg    <= data_reg;
            bit_count   <= bit_count -1;
            next_state  <= (cmd == CMD_RD_ACK)? S_RD_ACK2_0: S_WR_DATA_0;
    end

    // READ STATE
    //  Similar to above state, but destination is STOP State
    S_RD_ACK2_0: begin
            cmd         <= cmd;
            scl_pad_oen <= HIGH;
            sda_pad_oen <= LOW;
            scl_pad_o   <= LOW;
            sda_pad_o   <= sda_pad_o;
```

```
                data_reg    <= data_reg;
                bit_count   <= bit_count;
                next_state  <= S_RD_ACK2_1;
        end
        S_RD_ACK2_1: begin
                cmd         <= (sda_pad_i == LOW)? CMD_STOP: CMD_RD_ACK;
                scl_pad_oen <= HIGH;
                sda_pad_oen <= LOW;
                scl_pad_o   <= HIGH;
                sda_pad_o   <= sda_pad_o;
                data_reg    <= data_reg;
                bit_count   <= bit_count;
                next_state  <= S_RD_ACK2_2;
        end
        S_RD_ACK2_2: begin
                cmd         <= (sda_pad_i == LOW)? cmd: CMD_RD_ACK;
                scl_pad_oen <= HIGH;
                sda_pad_oen <= LOW;
                scl_pad_o   <= HIGH;
                sda_pad_o   <= sda_pad_o;
                data_reg    <= data_reg;
                bit_count   <= bit_count;
                next_state  <= S_RD_ACK2_3;
        end
        S_RD_ACK2_3: begin
                cmd         <= cmd;
                scl_pad_oen <= HIGH;
                sda_pad_oen <= LOW;
                scl_pad_o   <= LOW;
                sda_pad_o   <= sda_pad_o;
                data_reg    <= data_reg;
                bit_count   <= bit_count;
                next_state  <= (cmd == CMD_STOP)? S_STOP_0 : S_RD_ACK2_0;
        end

        // STOP STATE
        //  Both SCL and SDA are LOW, then SCL HIGH and SDA LOW
        //  before moving to IDLE STATE
        S_STOP_0: begin
                cmd         <= CMD_STOP;
                scl_pad_oen <= HIGH;
                sda_pad_oen <= HIGH;
                scl_pad_o   <= LOW;
                sda_pad_o   <= LOW;
                data_reg    <= data_reg;
                bit_count   <= bit_count;
                next_state  <= S_STOP_1;
        end
        S_STOP_1: begin
                cmd         <= CMD_IDLE;
                scl_pad_oen <= HIGH;
                sda_pad_oen <= HIGH;
                scl_pad_o   <= HIGH;
                sda_pad_o   <= LOW;
                data_reg    <= data_reg;
                bit_count   <= bit_count;
                next_state  <= S_IDLE;
        end

    endcase
  end
end
endmodule
```

Appendix c: I2C_Interface Source Code

```
`timescale 1ns / 1ps

//*************************************************************************//
//    Class:         CECS490B Senior Projects                              //
//    Project name:  Garbage Collector                                     //
//    File name:     I2C_Interface.v                                       //
//                                                                         //
//    Created by Chanartip Soonthornwan on October 10, 2018.               //
//    Copyright @ 2018 Chanartip Soonthornwan. All rights reserved.        //
//                                                                         //
//    Abstract:      Interconnection of Tramel Blaze and I2C.              //
//                   This module contains register for holding             //
//                   I2C address and I2C data to be ready before           //
//                   the I2C Core initiates the transmission.              //
//*************************************************************************//
module I2C_Interface (
    input wire          clk,            // System Clock Input
    input wire          rst,            // Synchronous Reset input
    input wire          bt_fire_i,      // Button Down for firing next Transmission
    input wire [6:0]    sw_addr_i,      // Switch[15:9] address input
    input wire [7:0]    sw_data_i,      // Switch[7:0] data input
    // From/To I2C
    input wire          i2c_ready_i,    // Ready status from I2C Core
    output reg [6:0]    i2c_address_o,  // Address output to I2C Core
    output reg [7:0]    i2c_data_o,     // Data output to I2C Core
    output reg          i2c_start_o,    // Start signal to I2C Core
    // From/To Tramel Blaze
    input wire [15:0] tb_port_id_i,     // Port ID from Tramel Blaze
    input wire [15:0] tb_data_i,        // Out-port data from Tramel Blaze
    input wire        tb_write_st_i,    // Write Strobe from Tramel Blaze
    input wire        tb_read_st_i,     // Read Strobe from Tramel Blaze
    input wire        tb_intr_ack_i,    // Interrupt Acknowledge from Tramel Blaze
    output reg        tb_intr_r_o,      // Interrupt Request to Tramel Blaze
    output reg [15:0] tb_data_o         // ready status to Tramel Blaze
);
    // Parameters for reserved I2C PORTs
    localparam I2C_ADDR_PORT   = 16'h01;
    localparam I2C_DATA_PORT   = 16'h02;
    localparam I2C_CMD_PORT    = 16'h03;
    localparam I2C_STATUS_PORT = 16'h04;
    localparam SW_ADDR_PORT    = 16'h05;
    localparam SW_DATA_PORT    = 16'h06;

    // Reset signal for all flops
    wire   reset;
    wire   ready_pulse_w;
    assign reset = rst | ready_pulse_w;

    wire bt_fire_db;
    wire bt_fire_pulse_w;

    // Positive Edge Detector
    //  Generate an impulsive signal from
    //  the edge changing from LOW to HIGH.
    PED
        ready_pulse_ped(
    .clk    (clk          ),
    .rst    (rst          ),
    .d_in   (i2c_ready_i  ),
    .pulse  (ready_pulse_w)
    );
        button_ready_pulse_ped(
    .clk   (clk          ),
    .rst   (rst          ),
    .d_in  (bt_fire_db  ),
    .pulse (bt_fire_pulse_w)
    );

    // Debounce Module
    //  to stabilize firing input from the button down
    //  because the button is a mechanical input which might cause
    //  an unstable input in a short period of time,
    //  such that this module will generate a stable signal.
    Debounce db_mod(
    .clk(clk),
    .rst(rst),
    .db_in(bt_fire_i),
    .db_out(bt_fire_db)
    );

    // Tramel Blaze Data_input
    //  Assign a 16-bit TB input either input data or I2C status.
    //  If port_id is I2C_STATUS_PORT and read strobe is HIGH,
    //  TB input is 15'b0 with i2c_ready signal, otherwise 16'b0.
    //  **Note: it will be only I2C status for this project
    always@(*) begin
        if(tb_read_st_i) begin
            case(tb_port_id_i)
                I2C_STATUS_PORT: tb_data_o = {15'b0, i2c_ready_i};
                   SW_ADDR_PORT: tb_data_o = { 9'b0,   sw_addr_i};
                   SW_DATA_PORT: tb_data_o = { 8'b0,   sw_data_i};
                   default     : tb_data_o =  16'b0;
            endcase
        end
        else
            tb_data_o = 16'b0;
    end

    // RS-Flop for resetting the interrupt request
    //  for the Tramel Blaze when I2C is ready
    always@(posedge clk, posedge rst) begin
        if(rst)                   tb_intr_r_o <= 1'b0;
        else if(tb_intr_ack_i)    tb_intr_r_o <= 1'b0;
        else if(bt_fire_pulse_w)  tb_intr_r_o <= 1'b1;
        else                      tb_intr_r_o <= tb_intr_r_o;
    end

    // I2C Address Register
    //  Change output when port id is I2C_ADDR_PORT,
    //  otherwise output stays the same.
    always@(posedge clk, posedge reset) begin
        if(reset)
            i2c_address_o <= 7'b0;
        else if(tb_write_st_i &(tb_port_id_i == I2C_ADDR_PORT))
            i2c_address_o <= tb_data_i[6:0];
        else
            i2c_address_o <= i2c_address_o;
    end

    // I2C Data Register
    //  Change output when port id is I2C_DATA_PORT,
    //  otherwise output stays the same.
    always@(posedge clk, posedge reset) begin
        if(reset)
```

```
            i2c_data_o <= 8'b0;
        else if(tb_write_st_i &(tb_port_id_i == I2C_DATA_PORT))
            i2c_data_o <= tb_data_i[7:0];
        else
            i2c_data_o <= i2c_data_o;
    end

    // I2C Start signal Register
    //  Change output when port id is I2C_CMD_PORT,
    //  data in[0] is high, and write_strobe is high,
    //  otherwise output stays the same.
    wire i2c_write_w;
    assign i2c_write_w = tb_data_i[0] & tb_write_st_i
                                      & (tb_port_id_i == I2C_CMD_PORT);
    always@(posedge clk, posedge reset) begin
        if(reset)             i2c_start_o <= 1'b0;
        else if(!i2c_ready_i) i2c_start_o <= 1'b0;
        else if(i2c_write_w)  i2c_start_o <= tb_data_i[0];
        else                  i2c_start_o <= i2c_start_o;
    end

endmodule

//___________________ Positive Edge Detector_____________________________________
//  A module to detect Positive Edge input then returns one-shot pulse
//  output. If the input is HIGH at the first clock and second clock period,
//  PED would detect this and output HIGH for one clock period.
module PED(clk, rst, d_in, pulse);

   input        clk, rst;                // on-board clock, and AISO reset signal
   input            d_in;                // input signal
   output  wire    pulse;                // one-shot pulse

   reg             q1,q2;                // registers

   always@(posedge clk, posedge rst)
      if(rst)   {q1, q2} <= 2'b00;       // reset
      else      {q1, q2} <= {d_in, q1};  // q2 gets q1, and q1 get new signal

   // output at the moment of input change
   // q1     ____------------_________
   // q2     ________------------_____
   // pulse  ____----_________________
   assign   pulse = q1 & ~q2;

endmodule
//______________________________ End of PED _____________________________________
```

Appendix d: I2C_Top_Level Testbench

```
`timescale 1ns / 1ps
module I2C_Top_Level_tb;

    // Inputs
    reg clk;
    reg rst;
    reg       bt_fire_i;  // button down input
    reg [6:0] sw_addr_i;  // switches[15:9] for 7-bit address input
    reg [7:0] sw_data_i;  // switches[7:0] for 8-bit data input

    // Outputs
    wire SCL;

    // Bidirs
    wire SDA;

    localparam DB_TIME = 6050000*5; // 30 ms debounce time

    // Instantiate the Unit Under Test (UUT)
    I2C_Top_Level master (
        .clk(clk),
        .rst(rst),
        .SCL(SCL),
        .SDA(SDA),
        .bt_fire_i(bt_fire_i),
        .sw_addr_i(sw_addr_i),
        .sw_data_i(sw_data_i)
    );

    // Instantiate a Slave unit with address of 7'h72
    I2CTest #( .slaveaddress(7'h72) )
    slave1(
        .CLCK(clk),
        .SCL(SCL),
        .SDA(SDA)
    );

    // Instantiate a Slave unit with address of 7'h55
    I2CTest #( .slaveaddress(7'h55) )
    slave2(
        .CLCK(clk),
        .SCL(SCL),
        .SDA(SDA)
    );

    // Instantiate a Slave unit with address of 7'h01
    I2CTest #( .slaveaddress(7'h01) )
    slave3(
        .CLCK(clk),
        .SCL(SCL),
        .SDA(SDA)
    );

    always #5 clk = ~clk; // Toggle on-board clock every 5 ns

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;
        bt_fire_i = 0;
        sw_addr_i = 7'h01;
        sw_data_i = 8'hAA;

        // Reset the design to a known state
        @(negedge clk) rst = 0;
        #100  bt_fire_i = 1;
        #DB_TIME bt_fire_i = 0;

        // Setting switches input and press the fire button, then release
        sw_addr_i = 7'h72;
        sw_data_i = 8'hAA;
        #DB_TIME bt_fire_i = 1;
        #DB_TIME bt_fire_i = 0;

        // Setting switches input and press the fire button, then release
        sw_addr_i = 7'h72;
        sw_data_i = 8'h01;
        #DB_TIME bt_fire_i = 1;
        #DB_TIME bt_fire_i = 0;

        // Setting switches input and press the fire button, then release
        sw_addr_i = 7'h55;
        sw_data_i = 8'h02;

        // Setting switches input and press the fire button, then release
        sw_addr_i = 7'h12;
        sw_data_i = 8'h03;
        #DB_TIME bt_fire_i = 1;
        #DB_TIME bt_fire_i = 0;

        // Setting switches input and press the fire button, then release
        sw_addr_i = 7'h13;
        sw_data_i = 8'h04;
        #DB_TIME bt_fire_i = 1;
        #DB_TIME bt_fire_i = 0;
        // Note: this time, the master will wait for slave's acknowledge forever.

        #DB_TIME $finish;
    end

endmodule
```

Appendix e: I2C_Core Testbench

```
`timescale 1ns / 1ps
module I2C_Core_tb;

    // Inputs
    reg clk;
    reg rst;
    reg [6:0] i2c_addr_i;
    reg [7:0] i2c_data_i;
    reg i2c_start_i;

    // Outputs
    wire i2c_ready_o;
    wire SCL;

    // Bidirs
    wire SDA;

localparam
    S_RD_ACK2_0 = 15, S_RD_ACK2_1 = 16, S_RD_ACK2_2 = 17, S_RD_ACK2_3 = 18;

    // Instantiate the Unit Under Test (UUT)
    I2C_Core uut (
        .clk(clk),
        .rst(rst),
        .i2c_addr_i(i2c_addr_i),
        .i2c_data_i(i2c_data_i),
        .i2c_start_i(i2c_start_i),
        .i2c_ready_o(i2c_ready_o),
        .SCL(SCL),
        .SDA(SDA)
    );

    always #5 clk = ~clk;   // Toggle on-board clock every 5 ns

    // Simulate acknowledge bit from a slave at Read Acknowledge State
    assign uut.sda_pad_i = (uut.next_state == S_RD_ACK1_0)? 1 :
                           (uut.next_state == S_RD_ACK1_1)? 0 :
                           (uut.next_state == S_RD_ACK1_2)? 0 :
                           (uut.next_state == S_RD_ACK1_3)? 1 :
                           (uut.next_state == S_RD_ACK2_0)? 1 :
                           (uut.next_state == S_RD_ACK2_1)? 0 :
                           (uut.next_state == S_RD_ACK2_2)? 0 :
                           (uut.next_state == S_RD_ACK2_3)? 1 : 1;

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;
        i2c_addr_i = 0;
        i2c_data_i = 0;
        i2c_start_i = 0;

        // Reset signal brings the design to a known state
        @(negedge clk) rst = 0;

        // Setup inputs from I2C_Interface and then wait for some time
        @(negedge clk) i2c_addr_i = 7'h72;
        @(negedge clk) i2c_data_i = 8'h45;
        #10000
        @(negedge clk) i2c_start_i = 1'b1;
        #10000
        @(negedge clk) i2c_start_i = 1'b0;
        #1000000

        // Setup inputs from I2C_Interface and then wait for some time
        @(negedge clk) i2c_addr_i = 7'h41;
        @(negedge clk) i2c_data_i = 8'h92;
        #10000
        @(negedge clk) i2c_start_i = 1'b1;
        #10000
        @(negedge clk) i2c_start_i = 1'b0;

        #1000000 $stop;

    end

endmodule
```

Appendix f: I2C_Interface Testbench

```
`timescale 1ns / 1ps
module i2c_interface_tb;

    // Inputs
    reg clk;
    reg rst;
    reg bt_fire_i;
    reg [6:0] sw_addr_i;
    reg [7:0] sw_data_i;
    reg i2c_ready_i;
    reg [15:0] tb_port_id_i;
    reg [15:0] tb_data_i;
    reg tb_write_st_i;
    reg tb_read_st_i;
    reg tb_intr_ack_i;

    // Outputs
    wire [6:0] i2c_address_o;
    wire [7:0] i2c_data_o;
    wire i2c_start_o;
    wire tb_intr_r_o;
    wire [15:0] tb_data_o;

    localparam DB_TIME = 6050000*5; // 30 ms debounce time

    // Instantiate the Unit Under Test (UUT)
    I2C_Interface uut (
        .clk(clk),
        .rst(rst),
        .bt_fire_i(bt_fire_i),
        .sw_addr_i(sw_addr_i),
        .sw_data_i(sw_data_i),
        .i2c_ready_i(i2c_ready_i),
        .i2c_address_o(i2c_address_o),
        .i2c_data_o(i2c_data_o),
        .i2c_start_o(i2c_start_o),
        .tb_port_id_i(tb_port_id_i),
        .tb_data_i(tb_data_i),
        .tb_write_st_i(tb_write_st_i),
        .tb_read_st_i(tb_read_st_i),
        .tb_intr_ack_i(tb_intr_ack_i),
        .tb_intr_r_o(tb_intr_r_o),
        .tb_data_o(tb_data_o)
    );

    always #5 clk = ~clk; // Toggle on-board clock every 5 ns

    initial begin
        // Initialize Inputs
        clk = 0;
        rst = 1;
        bt_fire_i = 0;
        sw_addr_i = 7'h72;
        sw_data_i = 8'hAB;
        i2c_ready_i = 1;
        tb_port_id_i = 0;
        tb_data_i = 0;
        tb_write_st_i = 0;
        tb_read_st_i = 0;
        tb_intr_ack_i = 0;
        // Reset the design to a known state
        @(negedge clk) rst = 0;

        // Fire a start signal for I2C_Core,
        //  but it needs DB_TIME (30ms) to stabilize
        //  the push button signal, then bt_db_pulse
        //  will be generated for launching the I2C_Core
        #100 bt_fire_i = 1;

            @(negedge clk) tb_read_st_i = 1;         // start read
            @(negedge clk) tb_port_id_i = 16'h0004; // read status
            @(negedge clk) tb_port_id_i = 16'h0005; // read sw_address
            @(negedge clk) tb_port_id_i = 16'h0006; // read sw_data
            @(negedge clk) tb_port_id_i = 16'h0000; // reset port_id
            @(negedge clk) tb_read_st_i = 0;         // stop read

            @(negedge clk) tb_write_st_i = 1;        // start write
            @(negedge clk)
                tb_data_i = 16'h0033;    // write '33' to i2c_addr_o
                tb_port_id_i = 16'h0001; // write address
            @(negedge clk)
                tb_data_i = 16'h0044;    // write '44' to i2c_data_o
                tb_port_id_i = 16'h0002; // write data
            @(negedge uut.bt_fire_pulse_w) tb_port_id_i = 16'h03; // write start
            @(negedge clk) tb_port_id_i = 16'h0000; // reset port_id
            @(negedge clk) tb_write_st_i = 0;        // stop write

            // Simulating that I2C_Core Receive the start signal
            //  and move to Start condition state and set the
            //  ready signal to low indicating that it is busy.
            @(negedge clk) i2c_ready_i = 0;
            #100 i2c_ready_i = 1;

        #DB_TIME bt_fire_i = 0;


        #500 $finish;

    end

endmodule
```