



COLLEGE OF ENGINEERING

Department of Computer Engineering and Computer Science

I²C Protocol On FPGA

By

Chanartip Soonthornwan (014353883)

On November 29, 2018

Table of Contents

1	Introduction	5
2	Features	5
3	General I ² C Operation	5
4	Data Validity	6
5	Design Methodology	7
6	Verilog	7
7	I ² C for Garbage Collector	8
8	I ² C With a TramelBlaze.....	9
9	External Document	9
9.1	Nexy4 Datasheet.....	9
9.1.1	I/O Connection.....	10
9.1.2	Seven Segment Display	10
9.1.3	Pmod Connectors.....	11
9.2	PicoBlaze	12
9.2.1	Architecture	12
9.2.2	Instruction Set.....	13
9.3	Artix-7 Libraries Guide for HDL Design	16
10	Requirements.....	16
10.1	Interface Requirements	16
10.2	Physical Requirements.....	16
11	Top Level Design	17
11.1	Description	17
11.2	Block Diagram	17
11.3	Data Flow Description.....	17
12	Externally Developed Blocks	18
12.1	TramelBlaze.....	18
13	Internally Developed Blocks.....	19
13.1	I2C Top Level Design	19
13.2	I2C_Core.....	22
13.3	I2C Interface.....	25
13.4	AISO_RST.....	27

Table of Appendices

Appendix a: TOP_LEVEL Source Code	28
Appendix b: I2C Top Level Source Code.....	30
Appendix c: I2C_Core Source Code.....	33
Appendix d: I2C_Interface Source Code	41
Appendix e: TSI Source Code	44
Appendix f: I2C_Top_Level Testbench.....	46
Appendix g: I2C_Core Testbench	48
Appendix h: I2C_Interface Testbench.....	50

Table of Figures

Figure 1: Example of I ² C Bus	5
Figure 2: START and STOP condition.....	5
Figure 3: I ² C master to slave communication	6
Figure 4: Acknowledge from a slave	6
Figure 5: SDA is set while SCL is LOW and remain stable while SCL is HIGH.	6
Figure 6: I ² C simple state diagram	7
Figure 7: I ² C Timing Diagram with States.....	8
Figure 8: Nexys4 I/O interface	10
Figure 9: Common anode circuit node	11
Figure 10: PMOD connectors and pin assignments	11
Figure 11: PicoBlaze Block Diagram	12
Figure 12: Top Level Block Diagram	17
Figure 13: Top Level Detail Block Diagram.....	17
Figure 14: TramelBlaze Block Diagram	18
Figure 15: I2C Top Level Block Diagram.....	19
Figure 16: I2C Top Level Detail Block Diagram	19
Figure 17: Top Level Verification	20
Figure 18: Top Level Complete Transmission	21
Figure 19: Top Level Complete Transmission Explanation	21
Figure 20: Top Level Incomplete Transmission.....	21
Figure 21: I2C Core Block Diagram.....	22
Figure 22: I2C Core State Diagram	22
Figure 23: I2C_Core Verification with Simulated Acknowledgement.....	23
Figure 24: I2C_Core Timing waveform.....	24
Figure 25: I2C_Interface Block Diagram.....	25
Figure 26: I2C_Interface Reading and Writing phrase	26
Figure 27: I2C_Interface debouncing button input and clearing registers.....	26
Figure 28: AISO_Reset Block Diagram.....	27
Figure 29: AISO verification	27

Table of Tables

Table 1: Bidirectional Verilog code and Diagram.....	8
Table 2: Picoblaze Instruction Set.....	15
Table 3: Buffers used in TSI.....	16
Table 4: Designed buttons interface.....	16
Table 5: Designed switches part 1	16
Table 6: Designed switches part 2	16
Table 7: TramelBlaze I/O.....	18
Table 8: I ² C Top Level I/O Table	20
Table 9: I ² C Core I/O table.....	23
Table 10: I2C_Interface I/O Table	25
Table 11: AISO I/O.....	27

I²C Protocol

1 Introduction

I²C 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 I²C 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.

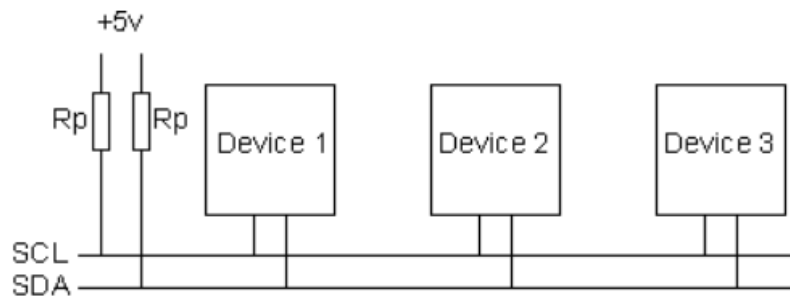


Figure 1: Example of I²C Bus

2 Features

- A master and multiple slave operation
- Requires only two pins to interface I²C bus (SCL and SDA)
- Support standard rate 100 Kbps
- Auto reset to Idle state if no selected slave address is on the bus

3 General I²C Operation

When the I²C 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

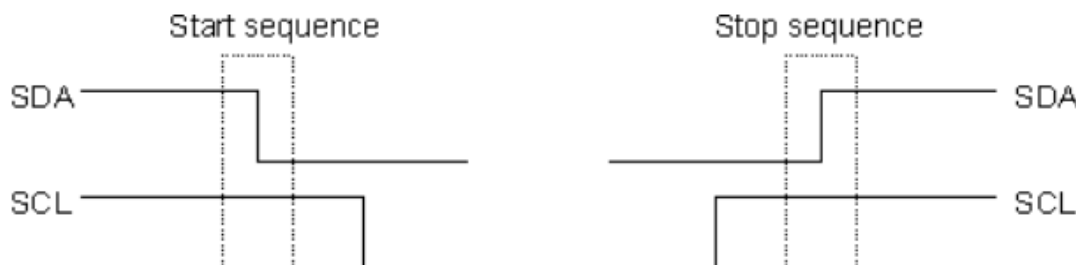


Figure 2: START and STOP condition

least significant bit (LSB) of the address of the slave that the master wants to communicate following by a read or write bit. For this I²C 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.

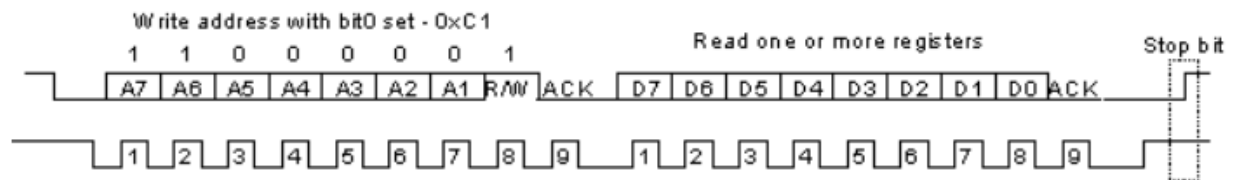


Figure 3: I²C master to slave communication

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 5). 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

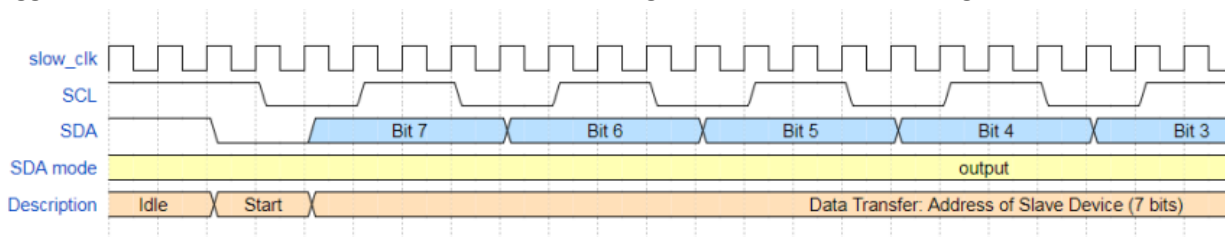


Figure 5: SDA is set while SCL is LOW and remain stable while SCL is HIGH.

a slave under the same condition, the acknowledge should be ready before the rising edge of SCL and stable while SCL is HIGH (Figure 4).

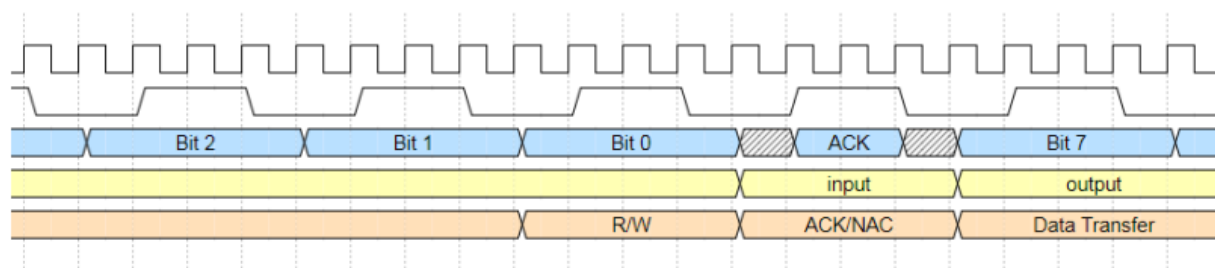


Figure 4: Acknowledge from a slave

5 Design Methodology

I²C 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

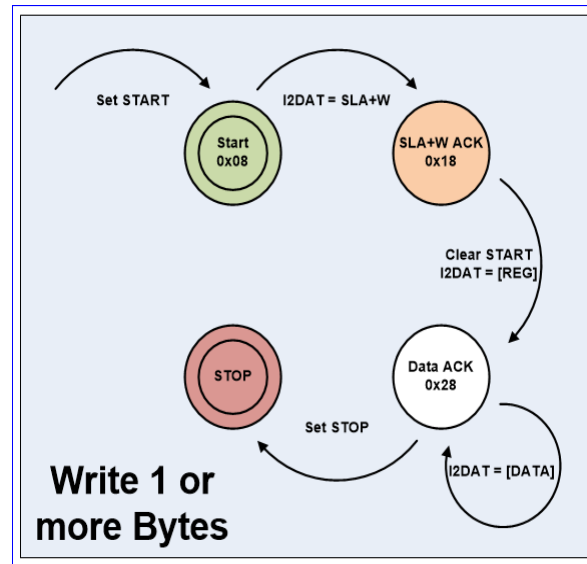


Figure 6: I²C simple state diagram

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

I²C 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.

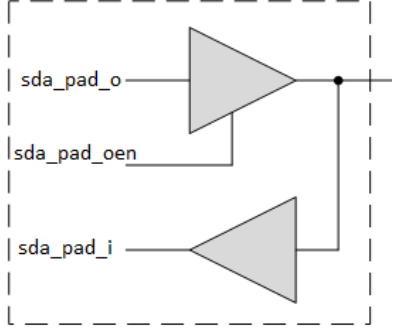
Sample snippet code	
<pre> input sda_pad_oen; inout SDA; reg sda_pad_o; assign SDA = (sda_pad_oen) ? sda_pad_o : 1'bz; assign sda_pad_i = (~sda_pad_oen) ? SDA : 1'bz; </pre>	

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 I²C for Garbage Collector

The I²C for Garbage Collector is customized specifically for this project as the requirement of the project is to use the I²C 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

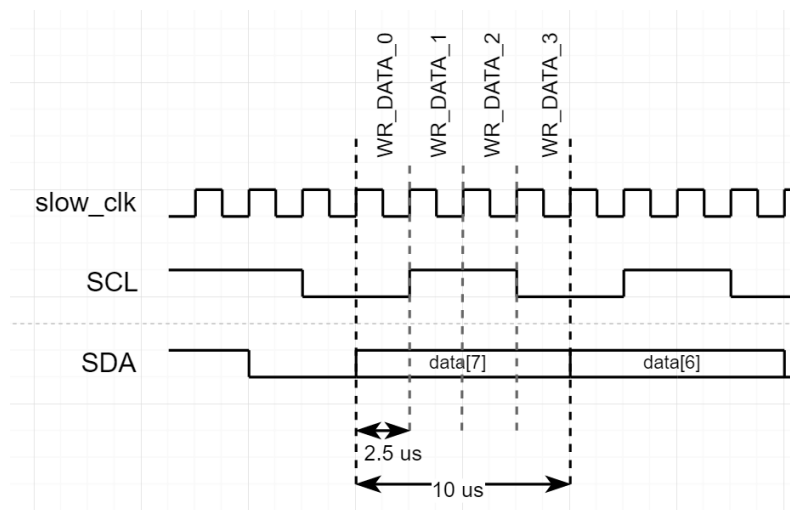


Figure 7: I²C Timing Diagram with States

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 I²C device (Figure 7).

8 I²C 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 I²C 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 I²C. 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, I²C's I/O, and other small modules. The interface contains registers that hold immediate values of a slave's address, data, and I²C start bit. Moreover, it sets and reset interrupt request for the microprocessor, and selects an input for the microcontroller to read.

9 External Document

These documents are used in developing this chip design as to assist the understanding of how to utilize the microcontroller (TramelBlaze) interface, and to create an actual interface on top of it for the later asserted Assembly program on a Programmable Read-Only Memory (PROM).

9.1 Nexy4 Datasheet

The datasheet provides essential information of Nexys4 (Artix-7) including basic I/O interfaces and 100MHz Crystal Oscillator.

9.1.1 I/O Connection

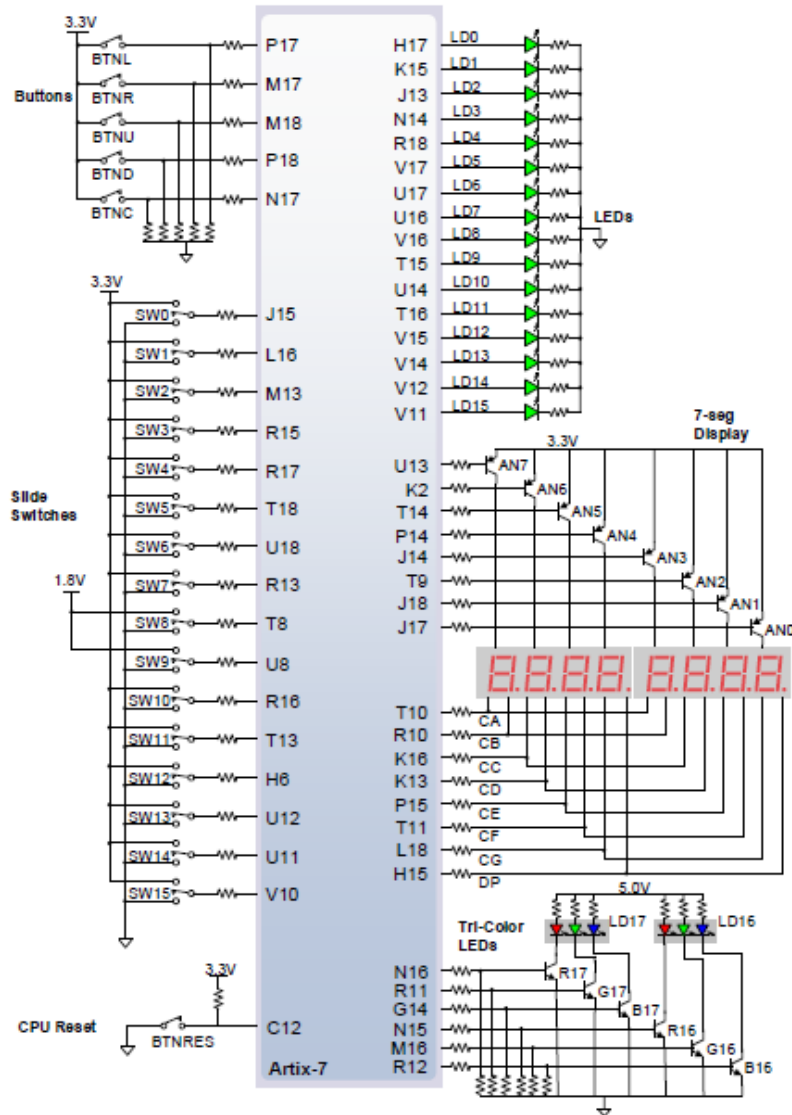


Figure 16. General Purpose I/O devices on the Nexys4 DDR.

Figure 8: Nexys4 I/O interface

The figure above shows the input and output pins of buttons, switches, LEDs, and 7-segment displays, and provides the voltages requirement for logic 1 and 0 for the buttons and switches.

9.1.2 Seven Segment Display

There are two of four-digit common anode of seven-segment LED displays on Nexys4. The segment LED can be individually illuminated in a “figure 8” pattern. In order to illuminate a segment, the segment’s anode should be driven high while individually driving low on the cathode. In other word, the segments are driven low when active.

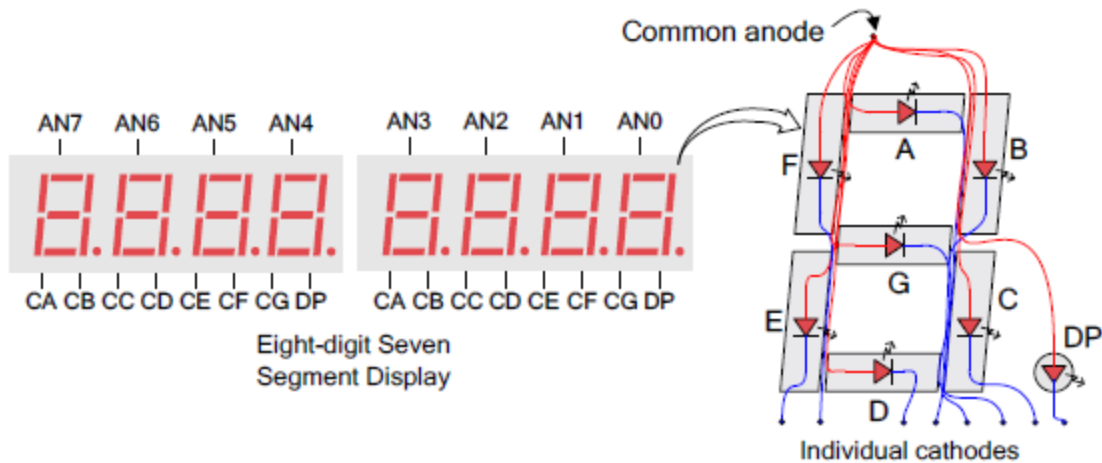
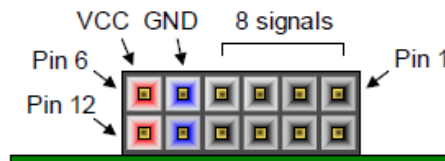


Figure 9: Common anode circuit node

9.1.3 Pmod Connectors

Pmod Connectors are standard 2x6 pin headers which included two pins of VCC, two pins of GND, and eight signal pins as shown in figure below. The VCC and GND pins can be deliver up to 1 ampere. Pmod pins on JA group are used for I²C pins since the pins can be programmed as open-drain pins which are needed for bidirectional SDA bus of the I²C.



Pmod JA	Pmod JB	Pmod JC	Pmod JD	Pmod XDAC
JA1: C17	JB1: D14	JC1: K1	JD1: H4	JXADC1: A13 (AD3P)
JA2: D18	JB2: F16	JC2: F6	JD2: H1	JXADC2: A15 (AD10P)
JA3: E18	JB3: G16	JC3: J2	JD3: G1	JXADC3: B16 (AD2P)
JA4: G17	JB4: H14	JC4: G6	JD4: G3	JXADC4: B18 (AD11P)
JA7: D17	JB7: E16	JC7: E7	JD7: H2	JXADC7: A14 (AD3N)
JA8: E17	JB8: F13	JC8: J3	JD8: G4	JXADC8: A16 (AD10N)
JA9: F18	JB9: G13	JC9: J4	JD9: G2	JXADC9: B17 (AD2N)
JA10: G18	JB10: H16	JC10: E6	JD10: F3	JXADC10: A18 (AD11N)

Figure 10: PMOD connectors and pin assignments

9.2 PicoBlaze

PicoBlaze is an 8-bit RISC microcontroller core optimized for Spartan-3, Virtex-II, and Virtex-II Pro families which is later developed to be a 16-bit RISC microcontroller called TrameBlaze, developed by John Tramel, which is capable to apply on Spartan-6 (Nexys3) and Artix-7 (Nexys 4). PicoBlaze provides the basis of instruction set and architecture, therefore PicoBlaze user's guide clarifies and developing a program on the TrameBlaze's instruction ROM.

9.2.1 Architecture

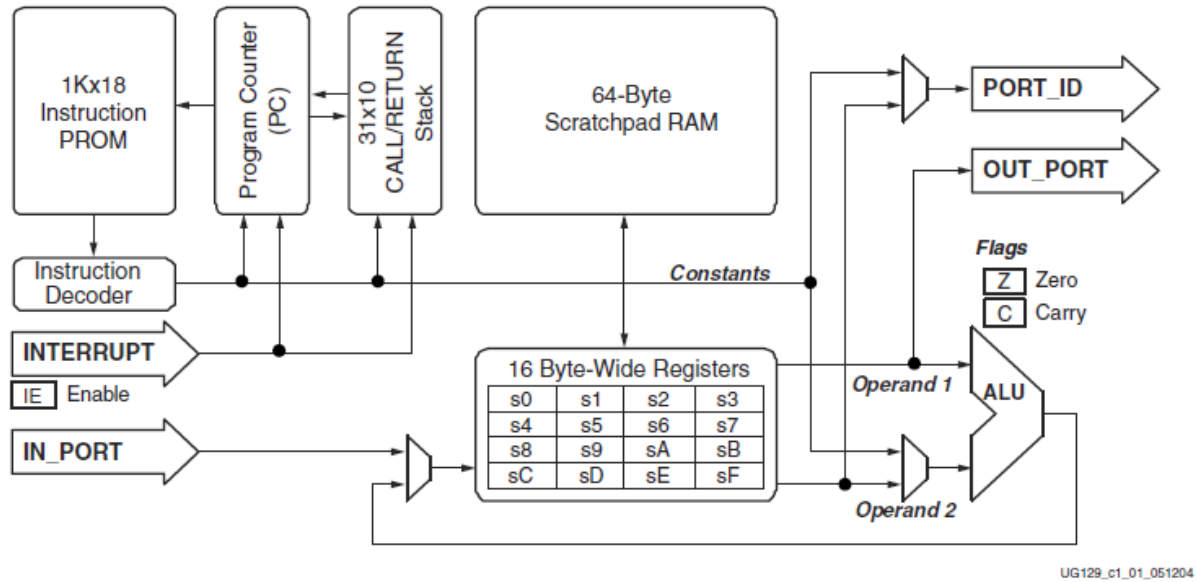


Figure 11: PicoBlaze Block Diagram

PicoBlaze and TrameBlaze share the similar architecture. The differences are that TrameBlaze consists of a 4Kx16 instruction ROM, 128x16 stack RAM, 512x16 Scratchpad RAM, and all bus lines are 16 bits.

9.2.2 Instruction Set

Instruction	Description	Function	ZERO	CARRY
ADD sX, kk	Add register sX with literal kk	$sX \leftarrow sX + kk$?	?
ADD sX, sY	Add register sX with register sY	$sX \leftarrow sX + sY$?	?
ADDCY sX, kk (ADDC)	Add register sX with literal kk with CARRY bit	$sX \leftarrow sX + kk + \text{CARRY}$?	?
ADDCY sX, sY (ADDC)	Add register sX with register sY with CARRY bit	$sX \leftarrow sX + sY + \text{CARRY}$?	?
AND sX, kk	Bitwise AND register sX with literal kk	$sX \leftarrow sX \text{ AND } kk$?	0
AND sX, sY	Bitwise AND register sX with register sY	$sX \leftarrow sX \text{ AND } sY$?	0
CALL aaa	Unconditionally call subroutine at aaa	$\text{TOS} \leftarrow \text{PC}$ $\text{PC} \leftarrow \text{aaa}$	-	-
CALL C, aaa	If CARRY flag set, call subroutine at aaa	If $\text{CARRY}=1$, $\{\text{TOS} \leftarrow \text{PC}, \text{PC} \leftarrow \text{aaa}\}$	-	-
CALL NC, aaa	If CARRY flag not set, call subroutine at aaa	If $\text{CARRY}=0$, $\{\text{TOS} \leftarrow \text{PC}, \text{PC} \leftarrow \text{aaa}\}$	-	-
CALL NZ, aaa	If ZERO flag not set, call subroutine at aaa	If $\text{ZERO}=0$, $\{\text{TOS} \leftarrow \text{PC}, \text{PC} \leftarrow \text{aaa}\}$	-	-
CALL Z, aaa	If ZERO flag set, call subroutine at aaa	If $\text{ZERO}=1$, $\{\text{TOS} \leftarrow \text{PC}, \text{PC} \leftarrow \text{aaa}\}$	-	-
COMPARE sX, kk (COMP)	Compare register sX with literal kk. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If $sX=kk$, $\text{ZERO} \leftarrow 1$ If $sX < kk$, $\text{CARRY} \leftarrow 1$?	?
COMPARE sX, sY (COMP)	Compare register sX with register sY. Set CARRY and ZERO flags as appropriate. Registers are unaffected.	If $sX=sY$, $\text{ZERO} \leftarrow 1$ If $sX < sY$, $\text{CARRY} \leftarrow 1$?	?
DISABLE INTERRUPT (DINT)	Disable interrupt input	$\text{INTERRUPT_ENABLE} \leftarrow 0$	-	-
ENABLE INTERRUPT (EINT)	Enable interrupt input	$\text{INTERRUPT_ENABLE} \leftarrow 1$	-	-
Interrupt Event	Asynchronous interrupt input. Preserve flags and PC. Clear INTERRUPT_ENABLE flag. Jump to interrupt vector at address 3FF.	Preserved $\text{ZERO} \leftarrow \text{ZERO}$ Preserved $\text{CARRY} \leftarrow \text{CARRY}$ $\text{INTERRUPT_ENABLE} \leftarrow 0$ $\text{TOS} \leftarrow \text{PC}$ $\text{PC} \leftarrow 3\text{FF}$	-	-
FETCH sX, (sY) (FETCH sX, sY)	Read scratchpad RAM location pointed to by register sY into register sX	$sX \leftarrow \text{RAM}[(sY)]$	-	-
FETCH sX, ss	Read scratchpad RAM location ss into register sX	$sX \leftarrow \text{RAM}[ss]$	-	-
INPUT sX, (sY) (IN sX, sY)	Read value on input port location pointed to by register sY into register sX	$\text{PORT_ID} \leftarrow sY$ $sX \leftarrow \text{IN_PORT}$	-	-
INPUT sX, pp (IN)	Read value on input port location pp into register sX	$\text{PORT_ID} \leftarrow pp$ $sX \leftarrow \text{IN_PORT}$	-	-

Instruction	Description	Function	ZERO	CARRY
JUMP aaa	Unconditionally jump to aaa	$PC \leftarrow aaa$	-	-
JUMP C, aaa	If CARRY flag set, jump to aaa	If CARRY=1, $PC \leftarrow aaa$	-	-
JUMP NC, aaa	If CARRY flag not set, jump to aaa	If CARRY=0, $PC \leftarrow aaa$	-	-
JUMP NZ, aaa	If ZERO flag not set, jump to aaa	If ZERO=0, $PC \leftarrow aaa$	-	-
JUMP Z, aaa	If ZERO flag set, jump to aaa	If ZERO=1, $PC \leftarrow aaa$	-	-
LOAD sX, kk	Load register sX with literal kk	$sX \leftarrow kk$	-	-
LOAD sX, sY	Load register sX with register sY	$sX \leftarrow sY$	-	-
OR sX, kk	Bitwise OR register sX with literal kk	$sX \leftarrow sX \text{ OR } kk$?	0
OR sX, sY	Bitwise OR register sX with register sY	$sX \leftarrow sX \text{ OR } sY$?	0
OUTPUT sX, (sY) (OUT sX, sY)	Write register sX to output port location pointed to by register sY	$PORT_ID \leftarrow sY$ $OUT_PORT \leftarrow sX$	-	-
OUTPUT sX, pp (OUT sX, pp)	Write register sX to output port location pp	$PORT_ID \leftarrow pp$ $OUT_PORT \leftarrow sX$	-	-
RETURN (RET)	Unconditionally return from subroutine	$PC \leftarrow TOS+1$	-	-
RETURN C (RET C)	If CARRY flag set, return from subroutine	If CARRY=1, $PC \leftarrow TOS+1$	-	-
RETURN NC (RET NC)	If CARRY flag not set, return from subroutine	If CARRY=0, $PC \leftarrow TOS+1$	-	-
RETURN NZ (RET NZ)	If ZERO flag not set, return from subroutine	If ZERO=0, $PC \leftarrow TOS+1$	-	-
RETURN Z (RET Z)	If ZERO flag set, return from subroutine	If ZERO=1, $PC \leftarrow TOS+1$	-	-
RETURNI DISABLE (RETI DISABLE)	Return from interrupt service routine. Interrupt remains disabled.	$PC \leftarrow TOS$ $ZERO \leftarrow \text{Preserved ZERO}$ $CARRY \leftarrow \text{Preserved CARRY}$ $INTERRUPT_ENABLE \leftarrow 0$?	?
RETURNI ENABLE (RETI ENABLE)	Return from interrupt service routine. Re-enable interrupt.	$PC \leftarrow TOS$ $ZERO \leftarrow \text{Preserved ZERO}$ $CARRY \leftarrow \text{Preserved CARRY}$ $INTERRUPT_ENABLE \leftarrow 1$?	?
RL sX	Rotate register sX left	$sX \leftarrow \{sX[6:0], sX[7]\}$ $CARRY \leftarrow sX[7]$?	?
RR sX	Rotate register sX right	$sX \leftarrow \{sX[0], sX[7:1]\}$ $CARRY \leftarrow sX[0]$?	?
SL0 sX	Shift register sX left, zero fill	$sX \leftarrow \{sX[6:0], 0\}$ $CARRY \leftarrow sX[7]$?	?
SL1 sX	Shift register sX left, one fill	$sX \leftarrow \{sX[6:0], 1\}$ $CARRY \leftarrow sX[7]$	0	?

Instruction	Description	Function	ZERO	CARRY
SLA sX	Shift register sX left through all bits, including CARRY	$sX \leftarrow \{sX[6:0], \text{CARRY}\}$ $\text{CARRY} \leftarrow sX[7]$?	?
SLX sX	Shift register sX left. Bit sX[0] is unaffected.	$sX \leftarrow \{sX[6:0], sX[0]\}$ $\text{CARRY} \leftarrow sX[7]$?	?
SR0 sX	Shift register sX right, zero fill	$sX \leftarrow \{0, sX[7:1]\}$ $\text{CARRY} \leftarrow sX[0]$?	?
SR1 sX	Shift register sX right, one fill	$sX \leftarrow \{1, sX[7:1]\}$ $\text{CARRY} \leftarrow sX[0]$	0	?
SRA sX	Shift register sX right through all bits, including CARRY	$sX \leftarrow \{\text{CARRY}, sX[7:1]\}$ $\text{CARRY} \leftarrow sX[0]$?	?
SRX sX	Arithmetic shift register sX right. Sign extend sX. Bit sX[7] Is unaffected.	$sX \leftarrow \{sX[7], sX[7:1]\}$ $\text{CARRY} \leftarrow sX[0]$?	?
STORE sX, (sY) (STORE sX, sY)	Write register sX to scratchpad RAM location pointed to by register sY	$\text{RAM}[(sY)] \leftarrow sX$	-	-
STORE sX, ss	Write register sX to scratchpad RAM location ss	$\text{RAM}[ss] \leftarrow sX$	-	-
SUB sX, kk	Subtract literal kk from register sX	$sX \leftarrow sX - kk$?	?
SUB sX, sY	Subtract register sY from register sX	$sX \leftarrow sX - sY$?	?
SUBCY sX, kk (SUBC)	Subtract literal kk from register sX with CARRY (borrow)	$sX \leftarrow sX - kk - \text{CARRY}$?	?
SUBCY sX, sY (SUBC)	Subtract register sY from register sX with CARRY (borrow)	$sX \leftarrow sX - sY - \text{CARRY}$?	?
TEST sX, kk	Test bits in register sX against literal kk. Update CARRY and ZERO flags. Registers are unaffected.	If $(sX \text{ AND } kk) = 0$, $\text{ZERO} \leftarrow 1$ $\text{CARRY} \leftarrow \text{odd parity of } (sX \text{ AND } kk)$?	?
TEST sX, sY	Test bits in register sX against register sY. Update CARRY and ZERO flags. Registers are unaffected.	If $(sX \text{ AND } sY) = 0$, $\text{ZERO} \leftarrow 1$ $\text{CARRY} \leftarrow \text{odd parity of } (sX \text{ AND } sY)$?	?
XOR sX, kk	Bitwise XOR register sX with literal kk	$sX \leftarrow sX \text{ XOR } kk$?	0
XOR sX, sY	Bitwise XOR register sX with register sY	$sX \leftarrow sX \text{ XOR } sY$?	0

Table 2: Picoblaze Instruction Set

Table 1 lists all PicoBlaze instructions that are applicable to use on TrameBlaze.

9.3 Artix-7 Libraries Guide for HDL Design

The Artix-7 Libraries Guide provides essential primitive information to complete the Technology Specific Instantiations (TSI).

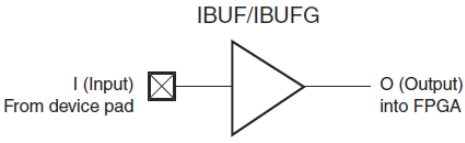
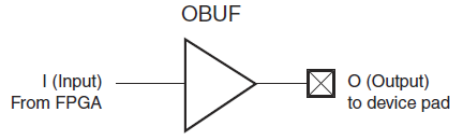
Buffer	Description
	IBUF/IBUFG are an input buffer which IBUFG is an input buffer for a clock signal.
	OBUF is an output buffer driving signals from Artix-7 to external output pads.

Table 3: Buffers used in TSI

10 Requirements

10.1 Interface Requirements

This design has three major components which are I²C Core, I²C Interface, and TramelBlaze altogether created a SOPC core that has inputs and outputs from and to Nexys4 board through a Technology Specific Instantiation (TSI). The design uses 15 on-board switches for 7-bit slave address and 8-bit slave data, two mechanical buttons for reset and fire command to initiate I²C transmission from on-board inputs, and seven-segment display to illuminate switches inputs.

10.2 Physical Requirements

As mention above, a user can use switches and buttons as inputs to interact to slave devices. Switches [15:9] are for slave address, switches [7:0] are for slave data, button up is for reset, and button down is for firing command. Also, seven-segment display are implemented for the user to be able to visualize the inputs.

BTNU	BTND	BTNC	BTNL	BTNR
Global Reset	Firing Command	N/A	N/A	N/A

Table 4: Designed buttons interface

SW15	SW14	SW13	SW12	SW11	SW10	SW9	SW8
Address[6]	Address[5]	Address[4]	Address[3]	Address[2]	Address[1]	Address[0]	N/A

Table 5: Designed switches part 1

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0
Data[7]	Data[6]	Data[5]	Data[4]	Data[3]	Data[2]	Data[1]	Data[0]

Table 6: Designed switches part 2

11 Top Level Design

11.1 Description

Top level demonstrates interconnections between I2C_Top_Level and TSI where the I2C_Top_Level consists of the entire digital design of I²C device and a Microprocessor, while TSI contains references to the Artix-7 libraries. Any I/O for the I2C_Top_Level must pass through the TSI before interacting with the I2C_Top_Level. The I2C utilizes seven-segment on the Nexys4 to display switches input.

11.2 Block Diagram

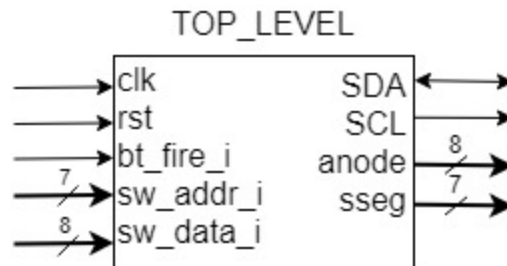


Figure 12: Top Level Block Diagram

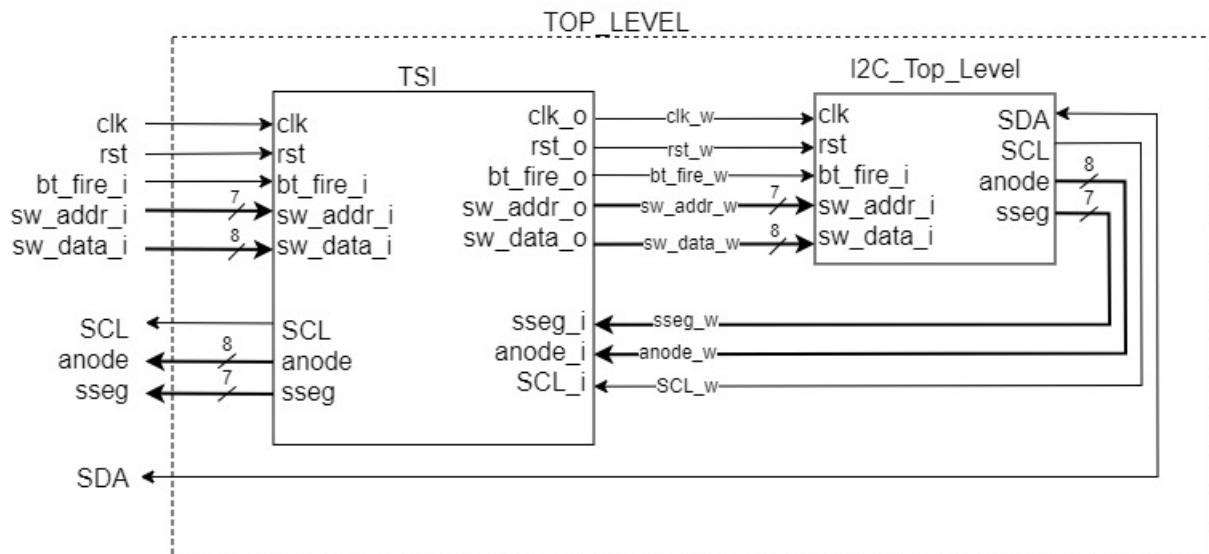


Figure 13: Top Level Detail Block Diagram

Source Code: Appendix a: TOP_LEVEL Source Code

11.3 Data Flow Description

First, a C program will be compiled to be an assembly program and stored in TramelBlaze(TB)'s instruction memory, or Programable Read Only Memory (PROM), on Nexys4. Then TB will fetch each instructions of the program and execute. Whenever there is an INPUT or OUTPUT instruction, it will setup I²C registers. Once OUTPUT instruction is executed with a port id of I2C_CMD_PORT, the I²C device changes from IDLE state to START state and operates the communication.

12 Externally Developed Blocks

12.1 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.

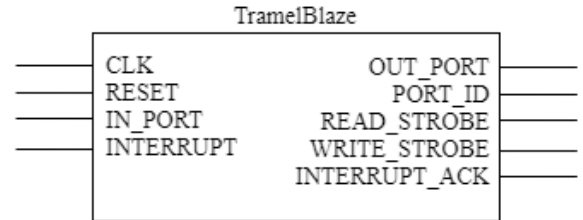


Figure 14: TramelBlaze Block Diagram

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 7: TramelBlaze I/O

13 Internally Developed Blocks

13.1 I2C Top Level Design

Description

The top level demonstrates interconnections of controls and data paths through a I²C core, a microprocessor (TramelBlaze), a I²C interface, and an Asynchronized-In-Synchronized-Out Reset signal module (AISO_RST). The switches input will be sent through the I²C interface before sending to the microprocessor and then sending to the I²C core through the interface.

Block Diagrams

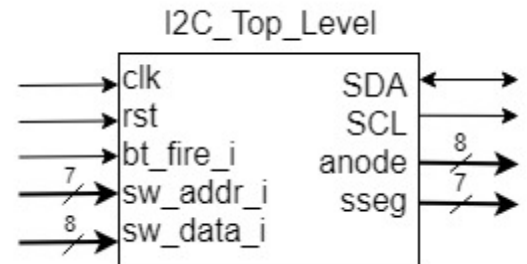


Figure 15: I2C Top Level Block Diagram

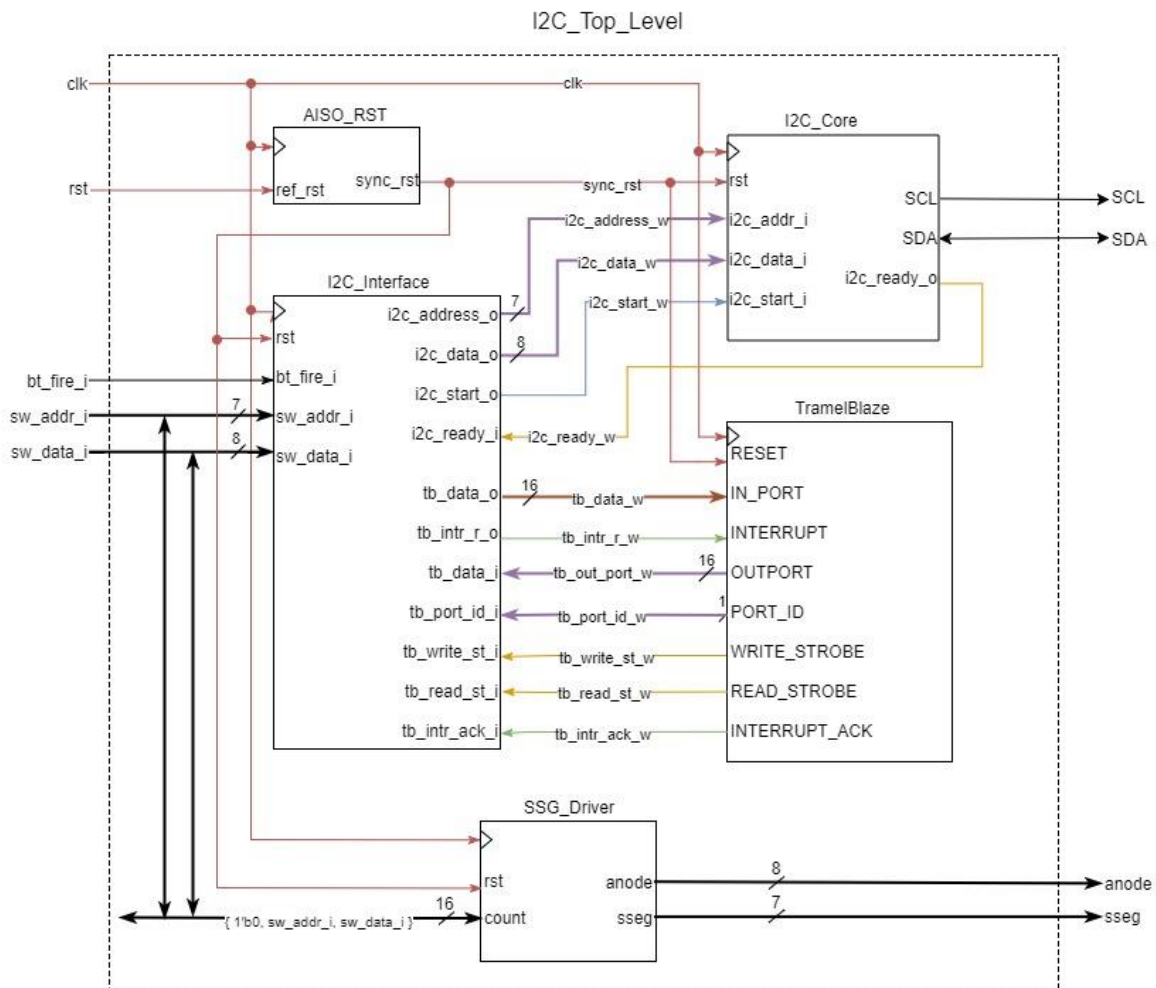


Figure 16: I2C Top Level Detail Block Diagram

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
anode	O	4	Digital	Anodes of 7-segment display	Control anodes on 7-segment display
sseg	O	7	Digital	LED segments	Display a number or a character

Table 8: I²C Top Level I/O Table

Source Code: Appendix b: 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 I²C 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.

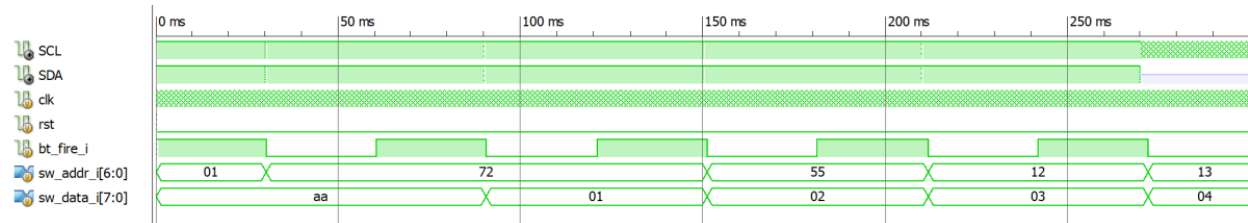


Figure 17: Top Level Verification

The bt_fire_i button is pushed for 30 ms before generating a one-pulse shot signal to The Tramelblaze to start the I²C transmission. Each time a transmission is done, there is a change on switches for address and data to simulate that the I²C 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 I²C is transmitted to an unknown slave device around the end of the simulation.

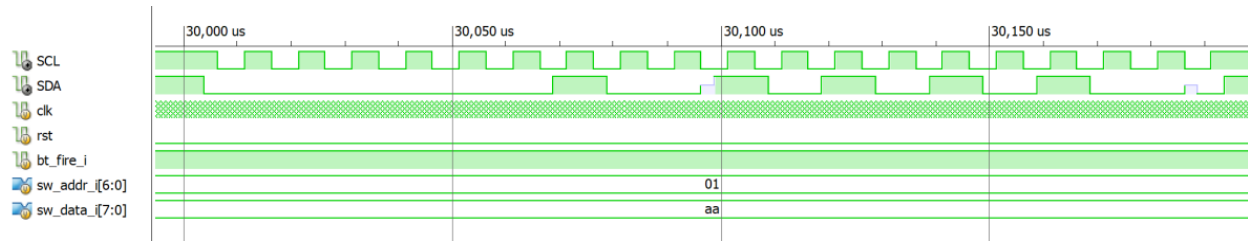


Figure 18: Top Level Complete Transmission

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,

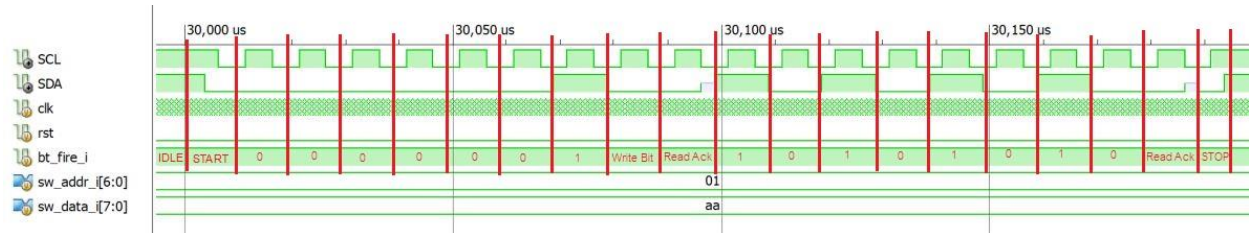


Figure 19: Top Level Complete Transmission Explanation

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 19).

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.

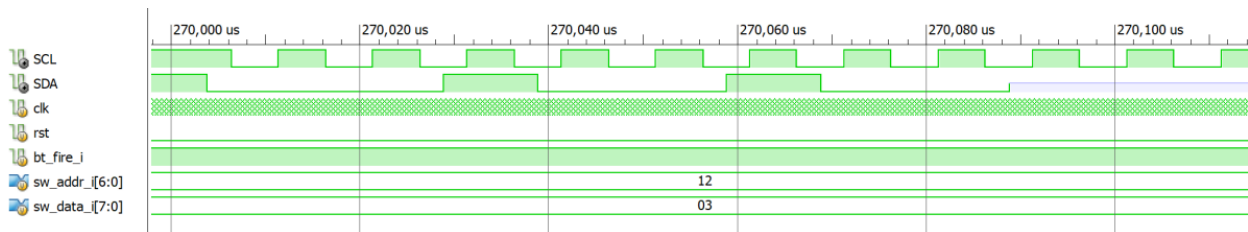


Figure 20: Top Level Incomplete Transmission

Verification Source Code: Appendix f: I2C_Top_Level Testbench

13.2 I2C_Core

Description

A master module of I²C 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 I²C Core since the SCL and SDA line must be setup in sequences of each other.

Block Diagrams

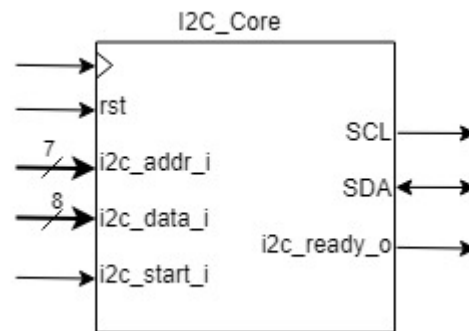


Figure 21: I2C Core Block Diagram

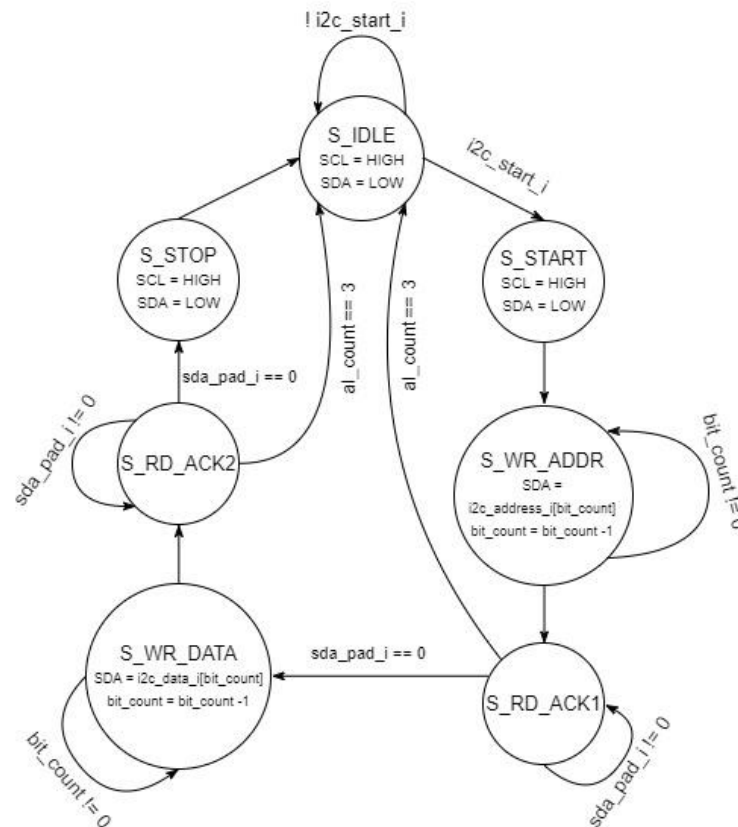


Figure 22: I2C Core State Diagram

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 9: I²C Core I/O table

Source Code: Appendix c: 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.

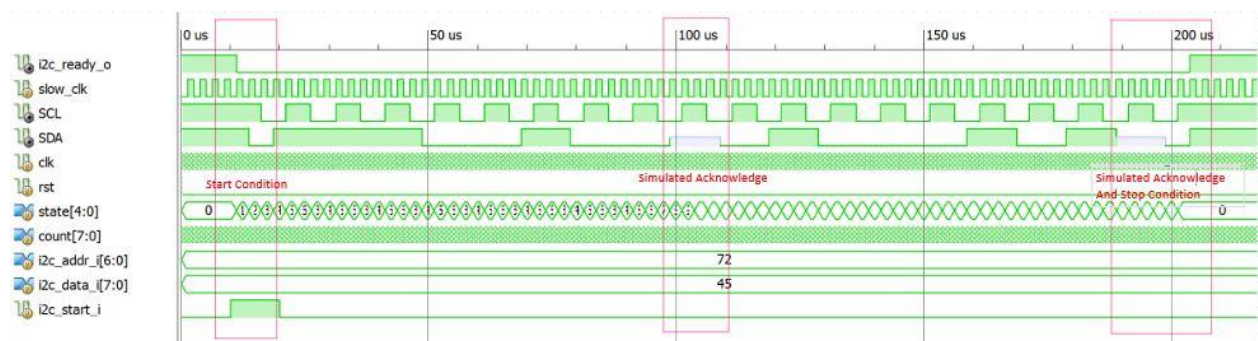


Figure 23: I2C_Core Verification with Simulated Acknowledgement

As shown in Figure 23, 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.

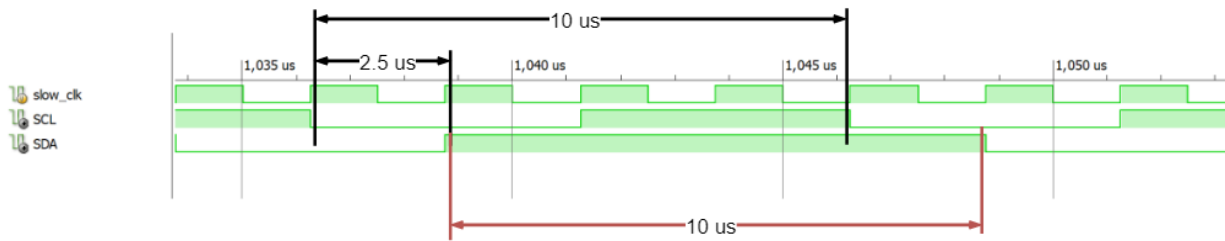


Figure 24: I2C_Core Timing waveform

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

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

Also, I²C timing standard requirement is 100Kbit per second or 10 us that can be calculated by this equation:

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

Then, the slow clock, SCL, and SDA 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 g: I2C_Core Testbench

13.3 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 I²C 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.

I/O

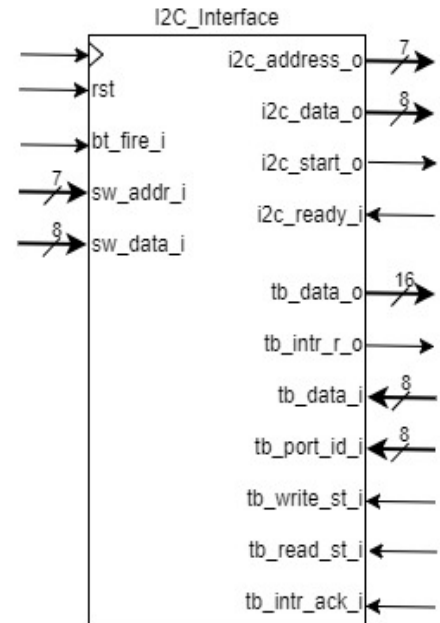


Figure 25: I2C_Interface Block Diagram

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 10: I2C_Interface I/O Table

Source Code: Appendix d: 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.

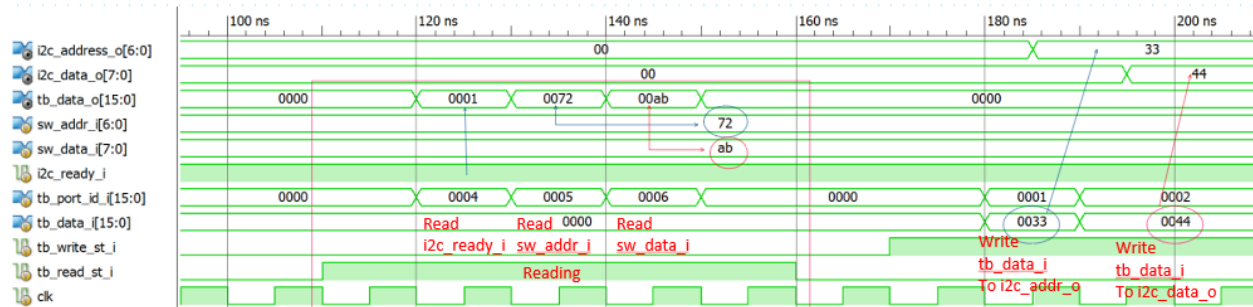


Figure 26: I2C_Interface Reading and Writing phrase

Reading to TB, when TrambelBlaze'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 26, 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'h0003, 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.

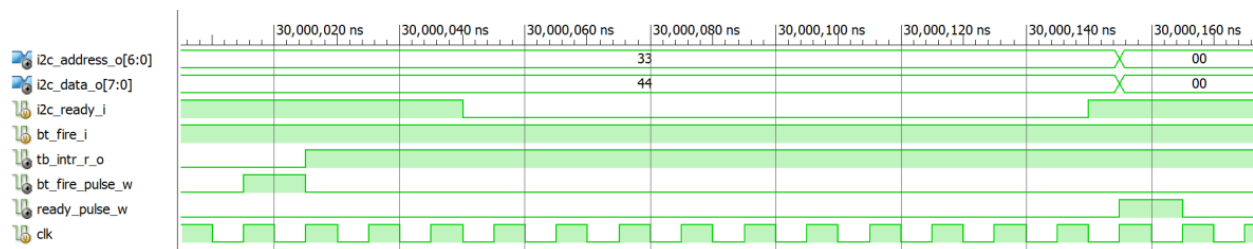


Figure 27: I2C_Interface debouncing button input and clearing registers

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 h: I2C_Interface Testbench

13.4 AISO_RST

Description

A device to synchronise a digital circuit design's system reset signal to occur on all components at the same edge of the clock instead of asynchronous reset signal that could make the system behave differently than expected.

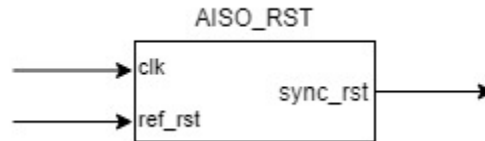


Figure 28: AISO_Reset Block Diagram

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
ref_rst	I	1	Digital	Reset signal	A reset signal from on-board reset signal
sync_rst	O	1	Digital	Synchronized reset signal	Provide system a synchronized reset signal to the system

Table 11: AISO I/O

Source Code: Appendix b: I2C Top Level Source Code

Verification

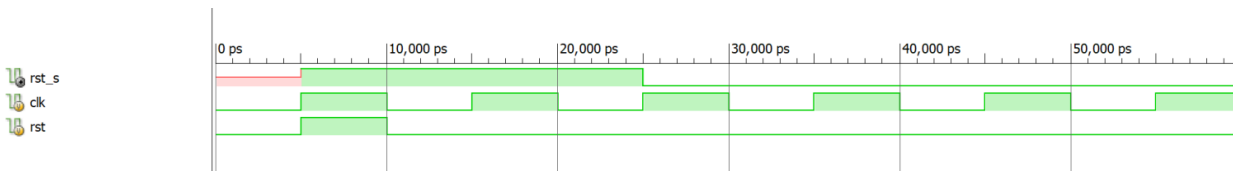


Figure 29: AISO verification

AISO_reset is verified as it generates a synchronous signal reset output when it receives asynchronous input.

Appendix a: TOP_LEVEL Source Code

```

`timescale 1ns / 1ps
//*****
//      Class:          CECS490B Senior Projects          //
//      Project name:    Garbage Collector                //
//      File name:       TOP_LEVEL.v                     //
//                                                              //
//      Created by Chanartip Soonthornwan on November 12, 2018. //
//      Copyright @ 2018 Chanartip Soonthornwan. All rights reserved. //
//                                                              //
//      Abstract:        Overview of the system on chip shows instantiates //
//                      TSI and I2C_Top_Level.            //
//*****
module TOP_LEVEL(
    input wire      clk,          // ON-BOARD clock
    input wire      rst,          // BUTTON UP for reset signal
    input wire      bt_fire_i,    // BUTTON DOWN for fire i2c transmission
    input wire [6:0] sw_addr_i,    // SWITCHES[15:9]
    input wire [7:0] sw_data_i,    // SWITCHES[ 7:0]

    inout wire      SDA,          // Bidirectional on JA2
    output wire     SCL,          // Output on JA1
    output wire [7:0] anode,       // Seven Segment Anodes
    output wire [6:0] sseg         // Seven Segment display
);

    wire      clk_w;
    wire      rst_w;
    wire      bt_fire_w;
    wire [6:0] sw_addr_w;
    wire [7:0] sw_data_w;
    wire [7:0] anode_w;
    wire [6:0] sseg_w;
    wire      SCL_w;

    // I2C Instantiate
    I2C_Top_Level i2c_sopc_inst (
        .clk(clk_w),              // clock from TSI
        .rst(rst_w),              // clock from TSI
        .SCL(SCL_w),              // SCL to TSI
        .SDA(SDA),                // SDA to/from JA2
        .bt_fire_i(bt_fire_w),    // fire input from TSI
        .sw_addr_i(sw_addr_w),    // address input from TSI
        .sw_data_i(sw_data_w),    // data input from TSI
        .anode(anode_w),          // anode output to TSI
        .sseg(sseg_w)             // sseg output to TSI
    );

    // TSI Instantiate
    TSI tsi_inst (
        .clk(clk),                // GLOBAL CLOCK
        .rst(rst),                // GLOBAL RESET
        .bt_fire_i(bt_fire_i),    // ON-BOARD BUTTON DOWN
        .sw_addr_i(sw_addr_i),    // ON-BOARD SWITCH[15:9]
        .sw_data_i(sw_data_i),    // ON-BOARD SWITCH[ 7:0]

        .clk_o(clk_w),            // Clock from TSI to SOPC
        .rst_o(rst_w),            // Reset from TSI to SOPC

```

```

.bt_fire_o(bt_fire_w), // Fire from TSI to SOPC
.sw_addr_o(sw_addr_w), // Address from TSI to SOPC
.sw_data_o(sw_data_w), // Data from TSI to SOPC

.SCL    (SCL),          // SCL from TSI to OUTSIDE
.anode   (anode),        // anode from TSI to OUTSIDE
.sseg    (sseg),         // sseg from TSI to OUTSIDE

.SCL_i   (SCL_w),        // SCL from SOPC to TSI
.anode_i (anode_w),       // anode from SOPC to TSI
.sseg_i  (sseg_w)         // sseg from SOPC to TSI
);

```

```
endmodule
```

Appendix b: 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 1.2 (November 11, 2018)
//      - Added anode and sseg for displaying switches current value
//
// 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

    output wire [7:0] anode,  // Anodes of 7-segment display
    output wire [6:0] sseg    // 7 segments
);
    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)
);

// Seven Segment Display
// Display Address on switches and data
// to send out over I2C as for observation
SSG_Driver ssg_driver(
.clk(clk),
.rst(sync_rst),
.count({1'b0, sw_addr_i, sw_data_i}),
.anode(anode[3:0]),
.sseg(sseg)
);
assign anode[7:4] = 4'b1111;

// 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 (Asynchronous 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 _____
    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 c: 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 2.1 (November 10, 2018)
//      - Added al_count for to check if there is an arbitrary lost from
//        the master waiting from a slave's signal. If there is no response
//        from a slave, then the master will return to idle state.
//
// 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 toggeling 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
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, CMD_AR_LOST    = 7,

    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] al_count;    // Arbitrary lost count
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;
        al_count      <= 0;
        next_state    <= S_IDLE;
        i2c_ready_o   <= HIGH;
    end

```

```

end
else begin
  case(state)
    // IDLE STATE
    // SCL and SDA are high until receive a start signal
    S_IDLE: begin
      if(i2c_start_i) begin          // receive 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;
        al_count     <= al_count;
        next_state   <= S_START_0;
        i2c_ready_o  <= LOW;          // now i2c is busy
      end
      else begin                      // still idling
        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;
        al_count     <= al_count;
        next_state   <= S_IDLE;
        i2c_ready_o  <= HIGH;
      end
    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;
    al_count     <= al_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
    al_count     <= 0;
    next_state   <= S_WR_ADDR_0;
  end
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;
    al_count     <= al_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;
    al_count     <= al_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;
    al_count     <= al_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;
    al_count     <= al_count;
    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.
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;
        al_count   <= al_count;
        next_state <= (al_count == 3)? S_IDLE : S_RD_ACK1_1;
    end

S_RD_ACK1_1: begin
    cmd          <= (sda_pad_i == LOW)? CMD_WR_DATA: cmd;
    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;
    al_count     <= al_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;
    al_count     <= al_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;
        al_count   <= 0;
        next_state <= S_WR_DATA_0;
    end
    else begin
        data_reg   <= data_reg;
        bit_count  <= bit_count;
        al_count   <= al_count + 1;
        next_state <= S_RD_ACK1_0;
    end
end

end

// WRITE STATE
// Similar to above state, but outputting 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;
        al_count   <= al_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;
    al_count     <= al_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;
    al_count     <= al_count;
    next_state   <= S_WR_DATA_3;
end
S_WR_DATA_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;
    al_count     <= al_count;
    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;
    al_count     <= al_count;
    next_state   <= (al_count == 3)? S_IDLE : 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;
        al_count    <= al_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;
    al_count     <= al_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;
    al_count     <= al_count + 1;
    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 d: 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 e: TSI Source Code

```

`timescale 1ns / 1ps
//*****
//      Class:          CECS490B Senior Projects          //
//      Project name:    Garbage Collector                //
//      File name:       TSI.v                            //
//                                                           //
//      Created by Chanartip Soonthornwan on November 12, 2018. //
//      Copyright @ 2018 Chanartip Soonthornwan. All rights reserved. //
//                                                           //
//      Abstract:       Technology Specific Interface (TSI) is a module //
//                      includes buffers for inputs from other devices //
//                      to the SOPC, and the buffers for outputs from //
//                      the SOPC to outside devices.          //
//*****
module TSI(
    input wire      clk,          // on-board 100MHz Oscilloscope
    input wire      rst,          // on-board button up reset signal
    input wire      bt_fire_i,    // button down I2C firing command
    input wire [6:0] sw_addr_i,    // switches[15:9] I2C address
    input wire [7:0] sw_data_i,    // switches[7:0] I2C data

    output wire      clk_o,
    output wire      rst_o,
    output wire      bt_fire_o,
    output wire [6:0] sw_addr_o,
    output wire [7:0] sw_data_o,

    output wire      SCL,          // I2C SCL bus
    output wire [7:0] anode,        // Seven-segment display's anodes
    output wire [6:0] sseg,        // seven segments output

    input wire      SCL_i,
    input wire [7:0] anode_i,
    input wire [6:0] sseg_i
);

    // Global Clock Buffer
    BUFG BUFG_inst (
        .O(clk_o),
        .I(clk)
    );

    // Input
    IBUF #(.IOSTANDARD("DEFAULT"))
    rst_inst(
        .O(rst_o),
        .I(rst)
    ),
    bt_fire_inst(
        .O(bt_fire_o),
        .I(bt_fire_i)
    ),
    sw_addr_inst[6:0](
        .O(sw_addr_o[6:0]),
        .I(sw_addr_i[6:0])
    ),
    sw_data_inst[7:0](

```

```
        .O(sw_data_o[7:0]),  
        .I(sw_data_i[7:0])  
    );  
  
    OBUF #(.IOSTANDARD("DEFAULT"))  
    SCL_inst(  
        .O(SCL),  
        .I(SCL_i)  
    ),  
    anode_inst[7:0](  
        .O(anode[7:0]),  
        .I(anode_i[7:0])  
    ),  
    sseg_inst[6:0](  
        .O(sseg[6:0]),  
        .I(sseg_i[6:0])  
    );  
  
endmodule
```

Appendix f: 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(
        .CLK(clk),
        .SCL(SCL),
        .SDA(SDA)
    );

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

    // Instantiate a Slave unit with address of 7'h01
    I2CTest #( .slaveaddress(7'h01) )
    slave3(
        .CLK(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;
    end

```

```

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 g: 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_ACK1_0 = 7, S_RD_ACK1_1 = 8, S_RD_ACK1_2 = 9, S_RD_ACK1_3 = 10,
    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;
    end

```

```
#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 h: 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;
    end

```

```

// 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

```

Intentionally
left blank.