Design of I^2C Interface for Custom ASICS Used in the Detection of Ionizing Radiation

by Nam Nguyen, Bachelor of Science

A Thesis Submitted in Partial
Fulfillment of the Requirements
for the Master of Science Degree

Department of Electrical and Computer Engineering
in the Graduate School
Southern Illinois University Edwardsville
Edwardsville, Illinois

December, 2008

ABSTRACT

DESIGN OF I²C INTERFACE FOR CUSTOM ASICS USED IN THE DETECTION OF IONIZING RADIATION

by

Nam Nguyen

Advisor: Dr. George L Engel

This thesis presents the design and simulation of an I²C (Inter-Integrated Circuit) serial interface. The design was described using the Verilog® hardware description language. The I²C Interface is intended for use in a family of integrated circuits (ICs) used in the detection of ionizing radiation. These custom ICs aid scientists in carrying out a wide variety of nuclear physics experiments. The ICs are being developed by the Integrated Circuit Design Research Laboratory at Southern Illinois University Edwardsville (SIUE) and were fabricated in a 5-Volt AMIS 0.5 μm, double-poly, tri-metal CMOS process (C5N).

The current ICs only provide for analog outputs. Storing data in a digital format on chip before transmittal to a host computer via an I²C or "I²C-like" interface should result in improved system performance since the transmission of digital data is much less susceptible to interference form environmental noise. A large number of ICs are required in these types of instrumentation systems and therefore, at some point in the future, the I²C interface described herein will likely be incorporated along with an on-chip ADC and buffer RAM into second-generation versions of our current chips.

The design of the proposed I²C interface was prototyped using inexpensive, readily-available Xilinx Spartan3E Starter development boards manufactured by Diligent. The prototype system consisted of two I²C "slaves" (emulating a pair of

custom ICs) and an I^2C “master”. The “master” gathered results from the “slaves” and transmitted the data to a personal computer where it was displayed. The prototype system worked flawlessly at 100 KBits/sec but can accommodate up to 128 slaves.

This work was initiated by the heavy-ion nuclear chemistry and physics group at Washington University in Saint Louis and is currently funded by NSF Grant #06118996.

ACKNOWLEDGEMENTS

I would like to thank my mentor Dr. George L. Engel for his constant support, encouragement, and guidance throughout this project. I would also like to thank my fellow researchers in the IC Design Lab, especially Nagachaitanya Yelchuri. I would also like to acknowledge the contributions of fellow gradauate student Krum Valkov whose Masters Project provided the impetus for the work described in this thesis.

I wish to thank Mr. Steve Muren who was instrumental in providing the necessary equipment for the lab. Finally, I would like to extend my appreciation to my parents, my sister, my friends and especially my wife who are always beside me and help me in the most difficult times.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

Background

The IC Design Research Laboratory at Southern Illinois University Edwardsville (SIUE) is part of an interuniversity collaboration which has as its long-term aim the development of a suite of multi-channel custom integrated circuits (ICs) suitable for use in a wide variety of nuclear physics experiments where the detection of ionizing radiation is needed. The greater collaboration includes researchers at Washington University in Saint Louis, Western Michigan University, Indiana University, and Michigan State University.

Frequently in nuclear physics experiments, the type of radiation incident on the detector must be classified, the energy of the particle must be determined, and the position of interaction within the detector must be accurately estimated. A typical array of silicon strip detectors used in nuclear physics experiments is pictured in Figure 1.1.

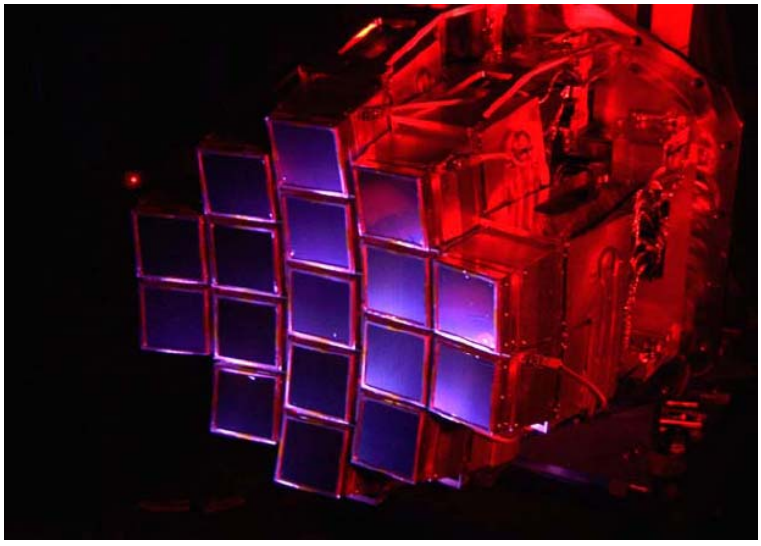


Figure 1.1: An array of silicon strip detectors [Das:08].

The ICs that have already been designed and fabricated, as well as those currently under development, contain analog electronics which make the accomplishment of the scientific objectives enumerated above possible. It should be note that in Figure 1.1 each of the 17 (blue-colored) panels actually contains 32 detectors (16 in the horizontal and 16 in the vertical direction). Therefore, the electronics servicing the array in Figure 1.1 must be able to support 544 independent channels. A Si strip detector is a reversed-biased p-n junction. Incident radiation impinging on the detector releases a packet of electron-hole pairs. The size of the packet is proportional to the energy of the incident radiation.

Preliminary work on the ICs began in the year 2000 [Gan:00]. Initially the development was funded through support from the nuclear reactions group in the Department of Chemistry at Washington University in Saint Louis; however, because of the success of the project over the years, the work is now funded by the National Science Foundation (NSF Grant #06118996).

The reasons why the collaboration became interested in developing a family of custom multi-channel integrated circuits include: (1) the need for high density (hundreds if not thousands of independent channels) signal processing in low- and intermediate-energy nuclear physics community is widespread, (2) no commercial chip was found to do exactly what the researchers wanted, and (3) the scientists deemed it necessary for the "experimenter" to be in the "designer's seat".

In the following section, we briefly discuss in more detail an integrated circuit which was recently developed by our research lab. The chip can support 8 detectors and hence was christened "PSD8C" [Eng:07b, Hal:07, Pro:07]. The work described in this thesis was explicitly undertaken in order to potentially improve the performance of instrumentation systems which will use the PSD8C chip.

PSD8C IC

PSD8C is to be used in nuclear physics experiments where particle identification, total pulse height, and relative timing information are needed. The design employs a technique known as pulse shape discrimination (PSD) to classify the incident radiation. Each of the eight channels is composed of a time-to-voltage converter (TVC) with two time ranges (0.5 μsec, 2 μsec) and three sub-channels. Each of the sub-channels consists of a gated integrator with eight programmable charging rates and an externally programmable gate generator that defines the start (with four time ranges) and width (with four time ranges) of the gate relative to an external discriminator signal. The chip also supports 3 triggering modes.

The IC produces four sparsified ***analog*** pulse trains (3 integrator outputs and 1 TVC output) with synchronized addresses for off-chip digitization with a pipelined ADC. PSD8C, with two bias modes occupies an area of approximately 2.8 mm x 5.7 mm and has an estimated power dissipation of 135 mW in the high-bias mode. The PSD8C chip, presented in Figure 1.2, was fabricated in a 0.5 μm technology (AMIS C5N) available through MOSIS (MOS Implementation Services). The chip returned from fabrication in May 2008. It is been tested and its performance is as predicted. It is expected to be used in experiments later in 2009.

Our research lab is in the process of designing an on-chip ADC for the PSD8C chip and has already designed a small on-chip RAM to store the digitized data before being transmitted back to a host computer using an **Interconnect Integrated Circuit (I²C) Interface**. It is the design of the I²C Interface which is the focus of this thesis.

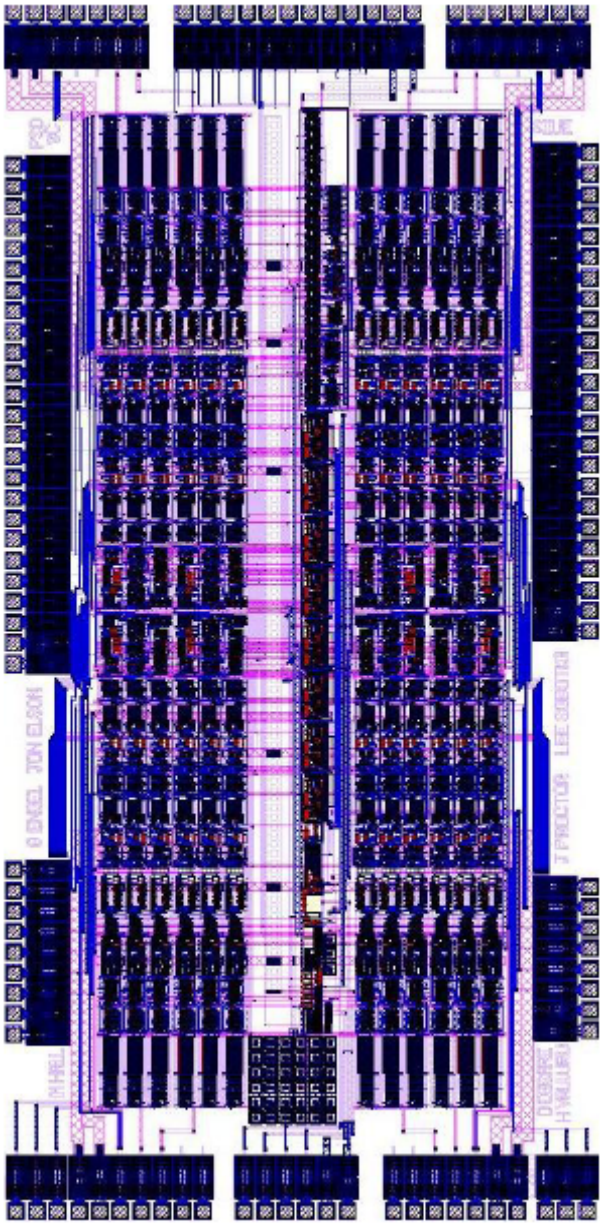


Figure 1.2: PSD8C layout [Das:08].

Need for I²C Interface

The on-chip ADC, RAM and I²C Interface will be integrated for the first time in a second generation PSD8C chip. The presence of an on-chip ADC and an on-chip RAM introduces an immediate need for a data communication between the

chip and a host computer to send data on the RAM. To minimize the system-level interconnect, we propose to transmit the contents of the RAM storing the ADC results back to a host computer via a serial bus, the I^2C Interface. This greatly simplifies the system level design and in particular the design of the mother-board and associated chip-boards. Moreover, storing data in a digital format on-chip before transmittal to a host computer over the I^2C Interface will result in an improved system performance since the transmission of digital data is much less susceptible to interference form environmental noise sources.

Object and Scope of Thesis

The object of this thesis is to design an I^2C Interface which can be used to transfer data from an on-chip RAM to a host computer suitable for integration with a series of custom ASICs developed or currently under development by the IC design research laboratory at SIUE. The ASICs are intended for use in the detection of ionizing radiation both in instruments intended for experiments in low-and intermediate-energy nuclear physics and (perhaps someday) in real-time homeland security nuclear threat detection systems.

Specifically, this thesis describes in detail an I^2C Slave that can be incorporated (along with an on-chip ADC and RAM) onto a second generation version of the existing PSD8C chip. It also describes an I^2C Master connected to I^2C Slaves using an I^2C bus. The I^2C Master can be downloaded to a FPGA board. The board in turn is connected to a host computer by a RS232 link. Finally, software running on the host computer, which is capable of establishing a serial port connection and drawing waveforms based on received data, will be discussed. The I^2C Interface is operating in 7-bit address mode, meaning one master is able to manage 2^7 or 128 slaves.

There are five chapters in this thesis. In Chapter 2, we will discuss the I²C serial interface protocol to provide readers a basic knowledge of the interface. Chapter 3 describes, in detail, the implementation of the I²C Slave and the I²C Master on the FPGA. In Chapter 4, a prototype system for fully testing the I²C Interface is described. The prototype system includes two Slaves with different addresses, a Master, a serial communication link (RS232) between the Master and a host computer, and software running on the host computer for display purposes. Finally, Chapter 5 provides a summary and suggests possible future work.

CHAPTER 2

I²C INTERFACE SPECIFICATION

Introduction

The Interconnect Integrated Circuit or I²C interface [Phi:00] was originally developed by Philips Semiconductors Company for data transfer among ICs at the Printed Circuit Board (PCB) level in early 1980s. All I²C-bus compatible devices have an interface allowing them to communicate directly with each other via the I²C-bus. The concept provides an excellent solution for problems in many interfacing in digital design. I²C is now broadly adopted by many leading chip design companies like Intel, Texas Instrument, Analog Devices, *etc*.

In I²C, only two bi-directional lines are required to carry information between devices, a Serial Data (SDA) line and a Serial Clock (SCL) line. Each device can be recognized by a unique 7 or 10 bits address. The device initiates a communication is called the Master, and at that time, all the other devices on the bus are considered Slaves. Normally, Masters are Microcontrollers. The I²C bus is a multi-Master bus, but only one Master can initiate a data transfer at any time.

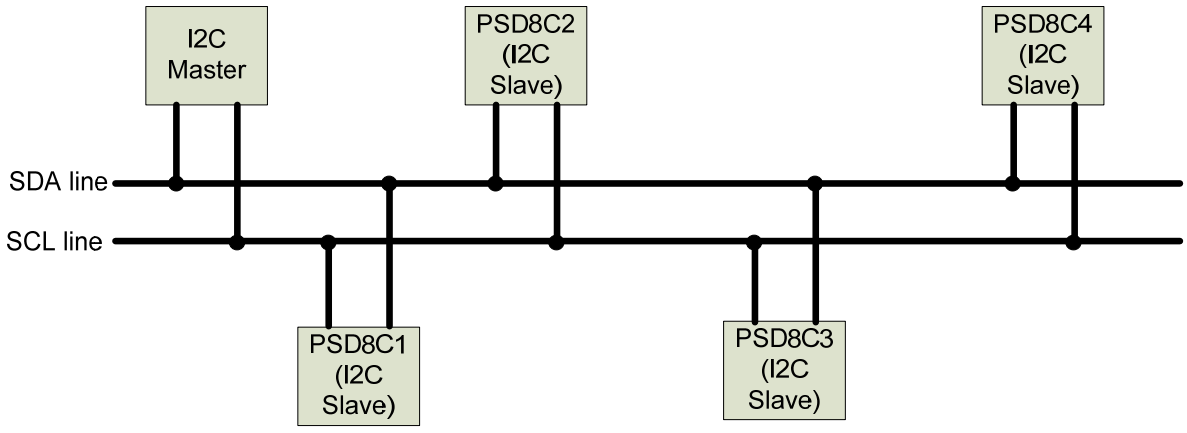


Figure 2.1: I²C-bus configuration [Phi:00].

In the figure above, there is one Master; the other devices are all Slaves. When a Master wants to initiate a communication, it issues a “START” condition. At that time, all devices, including the other Masters, have to listen to the bus for incoming data. After the “START” is issued, the Master sends the “ADDRESS” of the Slave that it wishes to communicate with along with a bit to indicate the direction of the data transfer (either read or write). All Slaves will then compare their addresses with the address received on the bus. If the addresses are identical, the Slave with the matching address will send an “ACKNOWLEDGEMENT” (ACK) to the Master. Slaves whose addresses do not match will not send an ACK. Once communication is established, the two lines are busy. No other device is allowed to control the lines except the Master and the Slave which was selected. When the Master wants to terminate communication, it will issue a “STOP” signal. After that, both SCL line and SDA line are released and free.

So far we have introduced the “START”, “ADDRESS”, “ACKNOWLEDGEMENT” and “STOP” signals. We will discuss these signals in more detail later. Before continuing, we need to understand the physical connection of the bus.

Physical Connection of the Bus

Both the SDA line and the SCL line are bi-directional, which means for a particular device, the bus can be controlled by the master or by one of many slaves. In order to implement that, a wired-OR function is employed.

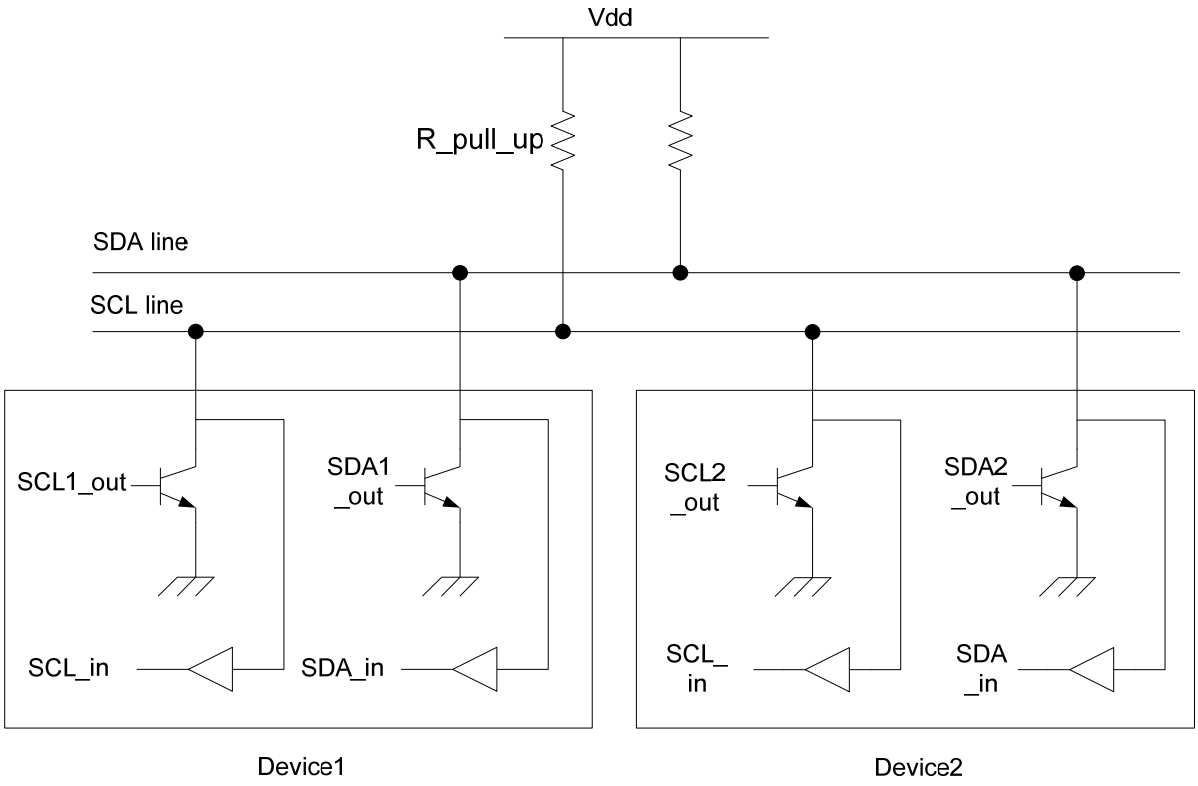


Figure 2.2: Connection of devices to I²C-bus [Phi:00].

The outputs of the devices must be connected to the bus using an open-collector bi-polar (or open-drain field-effect) transistor to perform the wired-OR function. The bus itself is connected to the supply voltage (V_{dd}) using a pull-up resistor, $R_{pull\text{-}up}$. Data on the I²C-bus can be transferred at rates of up to 100 KBit/sec in the Standard-mode, up to 400 KBit/sec in the Fast-mode, up to 3.4 Mbits/s in the High-speed mode, or even higher. When the bus is free, both lines are high and vice versa. Whenever a device wants to control the bus, it has to investigate the bus for a sufficient period of time. If the bus is free, the device can put a signal on the bus by driving the output transistor, pulling the bus to a "Low" level [Phi:00].

Start and Stop Conditions

When a Master wants to initiate a data transfer, it issues a Start condition and when it wants to terminate the transfer, a Stop condition will be initiated. There can be multiple Starts during one transaction called a repeated Start. The Master can then release the Stop condition whenever it wants to.

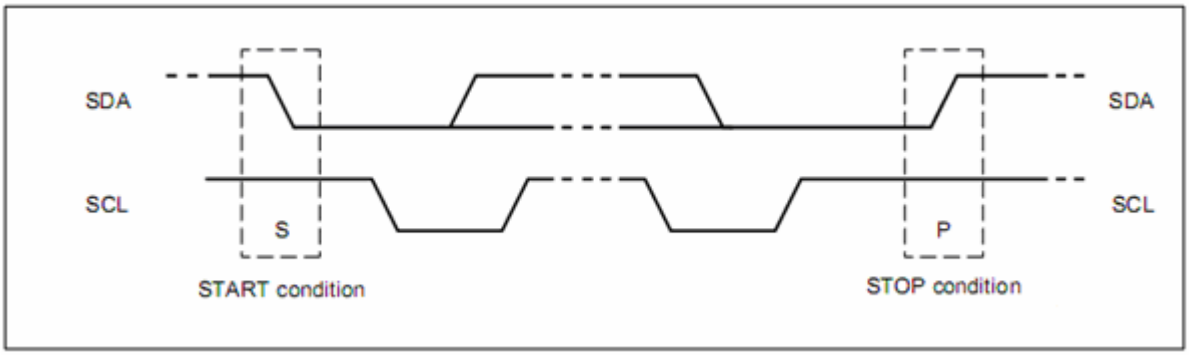


Figure 2.3: Start and Stop conditions [Phi:00].

As you can see in Figure 2.3, a Start is issued by bringing the SDA line low while the SCL line is high. After that the Master controls the SCL line and can generate clock signals. A Stop condition is implemented by transitioning the SDA line high while the SCL line is high.

Starting Bytes

The I²C bus is a byte-oriented protocol. After signaling Slaves by the Start condition, the Master sends "starting bytes" to the Slave. There are two components that make us the "starting bytes": Slave address and data direction (Read or Write).

The Master sends the MSB (Most Significant Bit) first and the LSB (Least Significant Bit) last. There are two addressing modes in the I²C protocol: the 7-bit and 10-bit address modes.

We will first consider the 7-bit addressing mode. After the START condition (S), a Slave address is sent. This address is the first 7 bits, the eighth bit is a data direction bit (RnW). If the direction bit is '0', it indicates a transmission (or WRITE). If the bit is '1', it indicates a request for data (or READ) (see Figure 2.4). A data transfer is always terminated by a STOP condition (P) generated by the Master. However, a Master can generate a repeated START condition (Sr) and address another Slave without first generating a STOP condition.

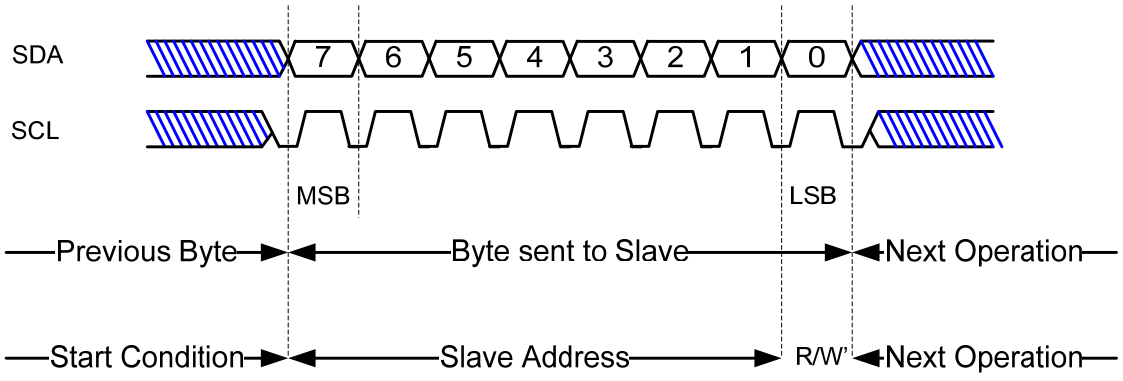


Figure 2. 4: Starting byte in 7-bit address mode [Phi:00].

We now consider the 10-bit addressing mode. When the I^2C bus became more popular, it was recognized that the number of available addresses in the 7-bit addressing mode is too small. Therefore, a new addressing mode (the 10-bit mode) was developed. The new addressing mode also supports the old one. Devices with 7-bit addresses can be connected with devices with 10-bit addresses on the same bus. In this mode, the first two bytes are dedicated for address and data direction. The format of the first byte is 11110xx, the last two bits of the first byte, combined with eight bits in the second byte form the 10-bit address (see Figure 2.5).

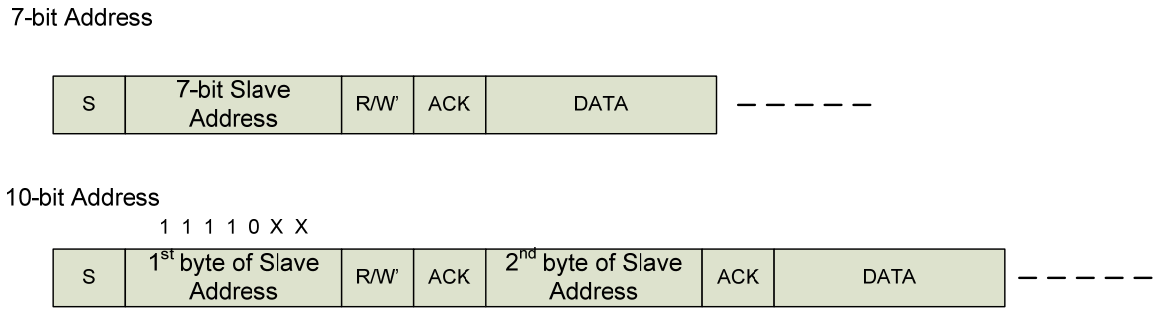


Figure 2.5: Starting bytes in 7-bit address and 10-bit address [Phi:00].

Acknowledgement

Acknowledgement is obligatory in order to inform the transmitter that data has been successfully transmitted. Figure 2.6 illustrates the acknowledgement mechanism. The Master generates the acknowledge-related clock pulse and the transmitter releases the SDA line (HIGH) during the acknowledge clock pulse so that the receiver can take control of the SDA line. If the receiver does not acknowledge, leaving the SDA line high, the transfer must be aborted. If it acknowledges by pulling the SDA line low, the transmitter knows that data has been successfully received, so it keeps sending data to the receiver.

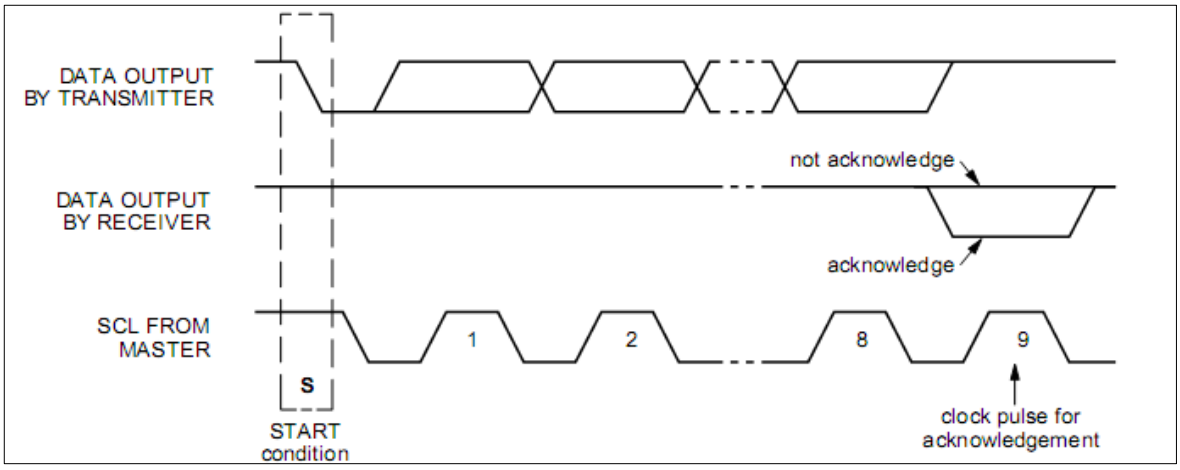


Figure 2.6: Acknowledgement on the I²C bus [Phi:00].

A Complete Data Transfer

All of the major aspects of I²C bus discussed so far are combined to create a complete data transfer from the transmitter to the receiver as in Figure 2.7. The SCL signal, Start and Stop signals, and the first byte must be generated by the Master. The Acknowledgement of the first byte must be generated by the Slave when it recognizes its address on the bus. The other Acknowledgements are generated by the receiver.

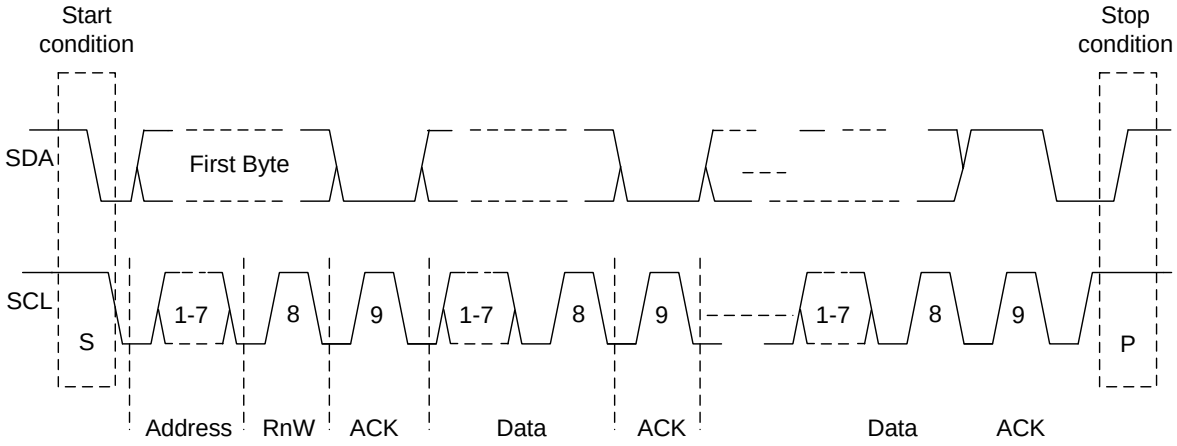


Figure 2.7: A complete data transfer in 7-bit addressing mode [Phi:00].

In this chapter, we have discussed the specifications relating to the I²C Interface. In the next chapter, the I²C Slave and I²C Master will be described using the Verilog® hardware description language and implemented on FPGA to meet all the I²C specifications presented here in Chapter 2.

CHAPTER 3

I²C IMPLEMENTATION ON FPGA

In this chapter, we are going to implement an I²C Master and an I²C Slave on Xilinx Spartan3E (500E) FPGA development boards manufactured by Digelent. Both the Master and the Slave will be described using Verilog. Before going into detail about how to implement the I²C Master and I²C Slave, it is important that one understands the architecture of the FPGA and the development board.

Xilinx Spartan 3E Architecture

The Spartan-3E Starter Kit board has the following key features [Xil:08]:

- Spartan-3E FPGA specific features
  - Parallel NOR Flash configuration
  - MultiBoot FPGA configuration from Parallel NOR Flash PROM
  - SPI serial Flash configuration
- Embedded development
  - MicroBlaze™ 32-bit embedded RISC processor
  - PicoBlaze™ 8-bit embedded controller
  - DDR memory interfaces

It has the following components:

- Xilinx XC3S500E Spartan-3E FPGA
  - Up to 232 user-I/O pins
  - 320-pin FBGA package
  - Over 10,000 logic cells
- Xilinx 4 Mbit Platform Flash configuration PROM
- Xilinx 64-macrocell XC2C64A CoolRunner™ CPLD

- 64 MByte (512 Mbit) of DDR SDRAM, x16 data interface, 100+ MHz
- 16 MByte (128 Mbit) of parallel NOR Flash (Intel StrataFlash)
    - FPGA configuration storage
    - MicroBlaze code storage/shadowing
- 16 Mbits of SPI serial Flash (STMicro)
    - FPGA configuration storage
    - MicroBlaze code shadowing
- 2-line, 16-character LCD screen
- PS/2 mouse or keyboard port
- VGA display port
- 10/100 Ethernet PHY (requires Ethernet MAC in FPGA)
- Two 9-pin RS-232 ports (DTE- and DCE-style)
- On-board USB-based FPGA/CPLD download/debug interface
- 50 MHz clock oscillator
- SHA-1 1-wire serial EEPROM for bitstream copy protection
- Hirose FX2 expansion connector
- Three Digilent 6-pin expansion connectors
- Four-output, SPI-based Digital-to-Analog Converter (DAC)
- Two-input, SPI-based Analog-to-Digital Converter (ADC) with programmable gain preamplifier
- ChipScope™ SoftTouch debugging port
- Rotary-encoder with push-button shaft
- Eight discrete LEDs
- Four slide switches
- Four push-button switches
- SMA clock input

• 8-pin DIP socket for auxiliary clock oscillator

Of those components, this thesis work required the use of the FPGA to download the I^2C Master and Slave, the MicroBlaze soft processor to control the I^2C Master, the RS232 port to transfer data from I^2C Master to host computer, the ADC for testing I^2C Interface, and the Hirose FX2 expansion connector for testing and some switches for resetting purposes.

The MicroBlaze is a soft microprocessor which can be placed into a Xilinx Field Programmable Gate Array (FPGA). The MicroBlaze processor is a 32-bit Harvard Architecture Reduced Instruction Set Computer (RISC) architecture. It has separate 32-bit instruction and data buses, and it can access data from both on-chip and external memory at the same time. The MicroBlaze can be connected to many types of peripherals; for examples, UARTs, GPIOs, Flash peripherals, *etc*. This is done via the Processor Local Bus (PLB) or the On-chip Peripheral Bus (OPB) [Jes:06]. In the figure below, we can see an example of MicroBlaze System. We can also add a User Peripheral (in our case the I^2C Master) to perform a specific application and then use the MicroBlaze to control it. The User Peripheral itself can be connected to an external hardware either inside or outside the Spartan 3E board, *i.e.* ADC, DAC, DIP Switches, LCD, LED or I/O pins, *etc*. See Figure 3.1.

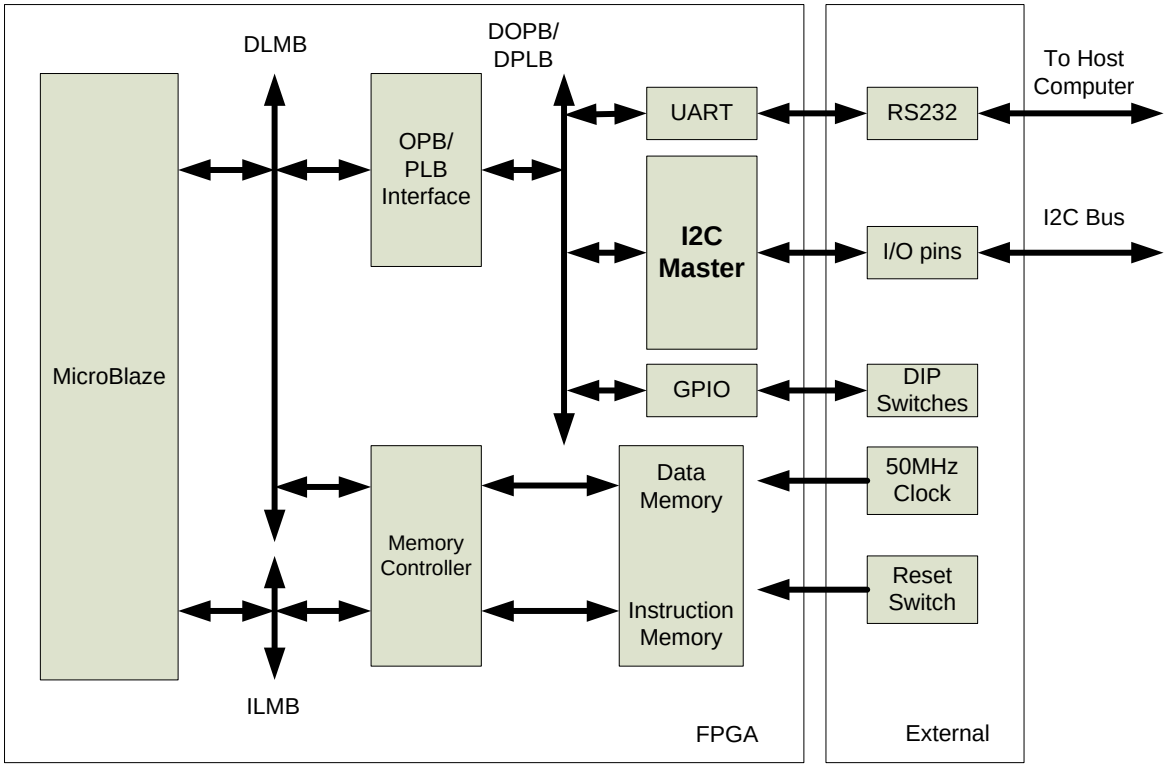


Figure 3.1: Example of a MicroBlaze System [The:06].

The processor system is connected using the On-chip Peripheral Bus (OPB) and/or the Processor Local Bus (PLB), so the custom peripheral must be OPB or PLB compliant. This means the top-level module of your custom peripheral must contain a set of bus ports that is compliant to the OPB or PLB protocol so that it can be attached to the system OPB or PLB [Jes:06]. See Figure 3.2.

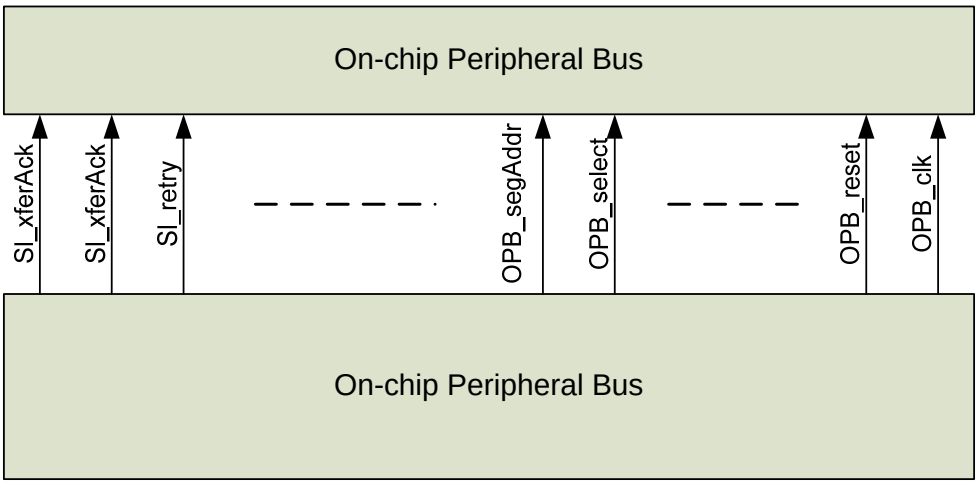


Figure 3.2: Interface between User Peripheral and OPB [Jes:06].

However, using OPB or PLB signals is very complicated and requires a deep understanding of PLB protocol. Fortunately, the Xilinx Embedded Development Kit software tools, which come with the board, provide an Intellectual-Property Interface (IPIF) library to implement common functionality among various processor peripherals. It is verified, optimized and highly parameterizeable. It also gives you a set of simplified bus protocol called IP Interconnect (IPIC) so that users do not have to worry about the PLB/OPB protocol anymore. The designer only has to learn the IPIC bus protocol, which is much easier, rather than working with the OPB or PLB bus protocol directly [Jes:06]. Figure 3.3 shows Custom Functionality, which is connected to the IPIF using IPIC. Custom Functionality is where we build our custom hardware for our specific purposes [Jes:06].

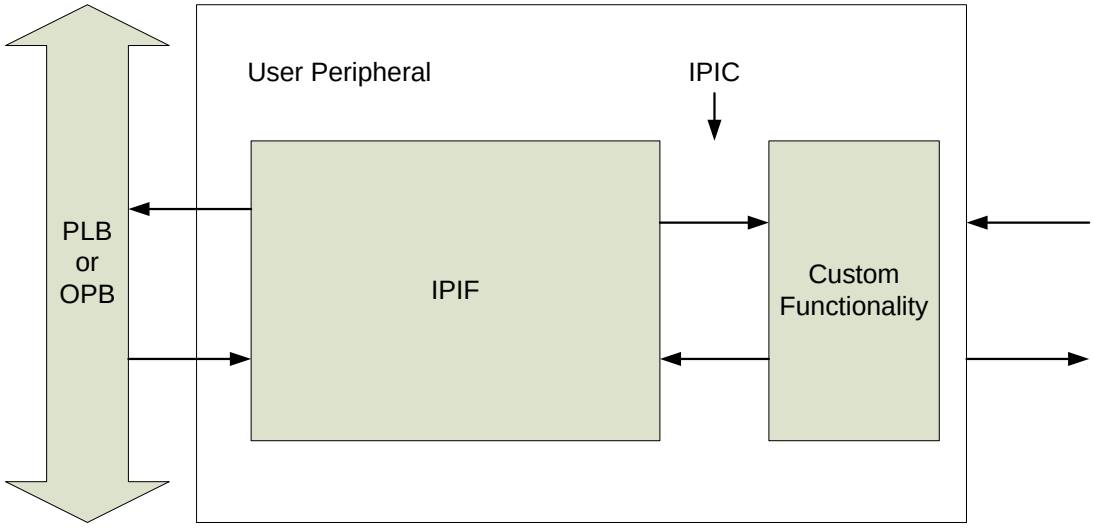


Figure 3.3: A detailed view of a User Peripheral [Jes:06].

A custom block can be written in both Verilog and VHDL to provide user-defined functions. In this section, only an overview is provided to give readers a general idea of the MicroBlaze, for more detail and specific information about how to build a Custom Functionality to interface to the PLB Bus, please refer to the "I^2C Microcontroller" subsection, and the "Master Implementation on FPGA" section of this chapter.

Slave Implementation on FPGA

As mentioned above, there are two types of devices on the I^2C bus, Master and Slave. A Slave cannot initiate a data transfer. It just monitors the SDA and SCL lines waiting for a START signal. After detecting a START signal, the Slave will compare its address to the address received. If the addresses match, it will perform an action requested by Master by either sending or receiving data.

There are six main functional blocks in the design of a Slave:

- Shift Register
- Start Stop Detector

- Sequence Counter
- Address Comparator
- ACK Block

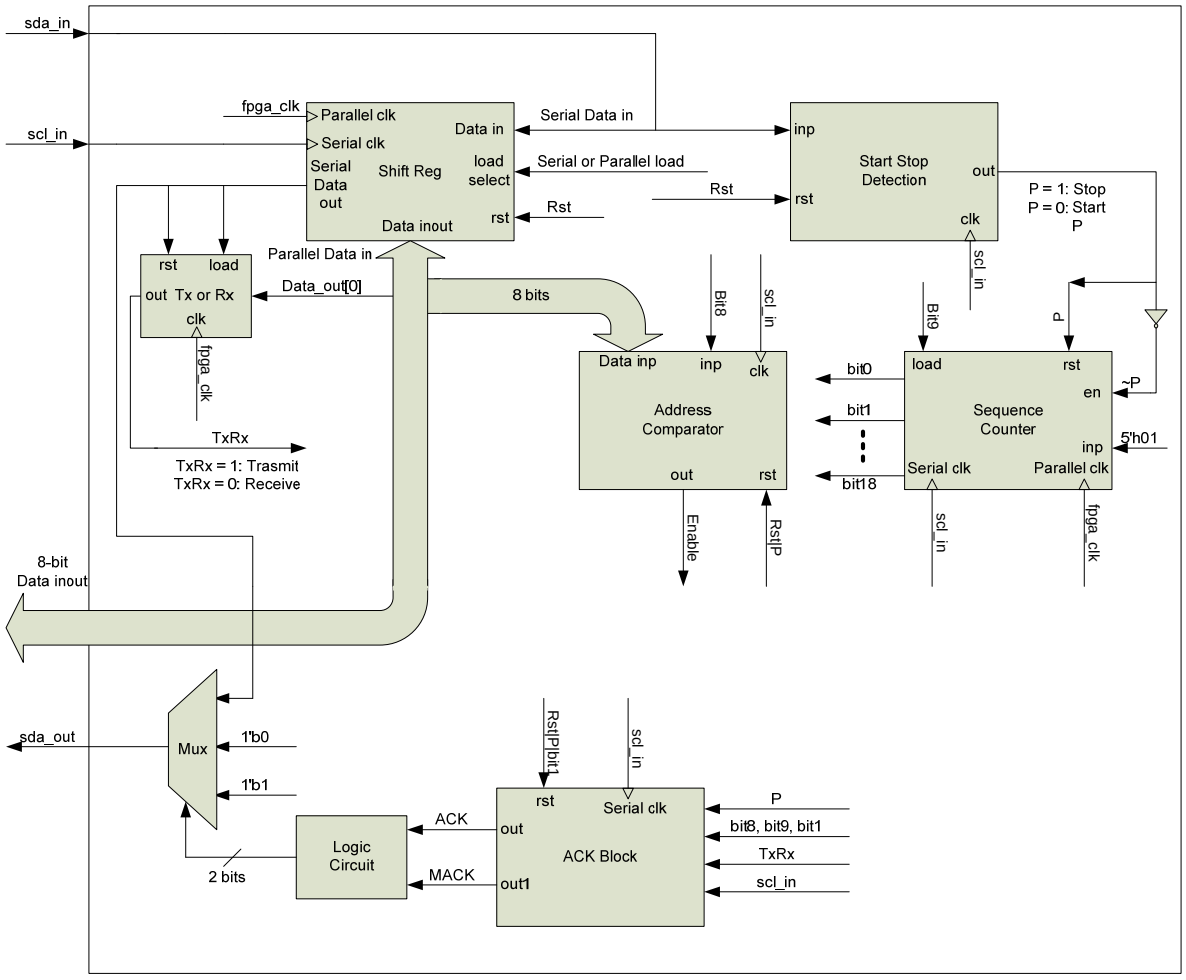


Figure 3.4: Slave block diagram.

A **Shift Register** is used to receive data from a Master and to transmit data previously stored in a memory somewhere on the slave controller. In our application the data input/output of the Shift Register is connected to a 32x8 RAM module, which will be described in more detail in Chapter 4.

The Slave employs a **Start or Stop Detector** module to detect a Start signal. Whenever there is a start signal, it will sample the SDA line to get the address by

using the Shift Register. The first byte and the second byte received will be compared with the Slave address (which is fixed and assigned to each Slave on reset) by the **Address Comparator**. If the address is correct, an enable signal will be initiated to activate the whole Slave system.

The **Sequence Counter** works as a state machine to help the Slave in making decisions at appropriate times. The main functions of the **ACK Block** are to detect acknowledgements from the Master and to generate acknowledgements to reply to the Master. Each of the above components will be described in additional detail in the following sections.

Shift Register

The Shift Register is used for parallel-to-serial and serial-to-parallel conversion. It is an 8-bit Shift Register with serial or parallel load select. In the receiving mode, the Shift Register samples data on the SDA line at the rising edge of the SCL signal and shifts the received bit to the left. After getting the whole byte, the data will be parallel loaded to the RAM. In the transmitting mode, the Shift Register is loaded with parallel data from the RAM first and then the data will be shifted to the left to transfer each bit to a receiver on the negative edge of the SCL signal. Figure 3.5 presents a block diagram for the Shift Register Module.

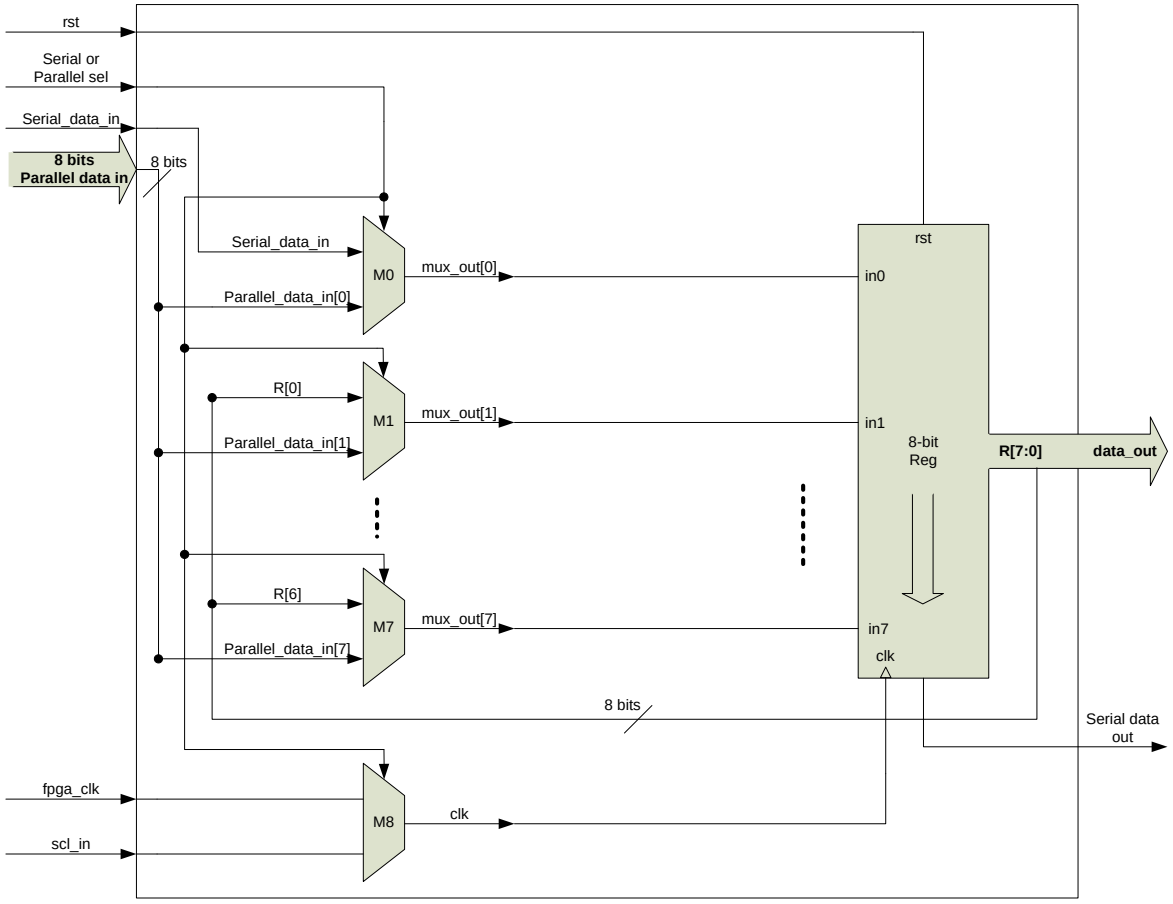


Figure 3.5: Shift Register.

The Shift Register has two operating modes, Serial and Parallel, associated with two clock inputs, scl_in and fpga_clk. In the parallel mode, the fpga_clk and parallel data from the RAM are selected by a multiplexer system. As the result, 8 bits from the RAM will be loaded to the register. In the serial mode, scl_in will be selected as the clock and data will be selected so that serial data input is loaded to the first bit and the other bits are shifted to the bottom of the register. The serial data output is taken from the MSB of the register.

Start/Stop Detector

A Start event occurs when there is a negative edge on the SDA line while the SCL line is high and a Stop event occurs when there is a positive-going transition on the SDA line while the SCL line is high. Thus, we need a device that works at both edges. Nevertheless, in fact, there is no flip flop that works at both edges. To create the device, we need to use a toggle register (toggles on the positive edge of its clock input) and two other registers. The device described below has the ability to detect Start, Stop and even repeated Start signals.

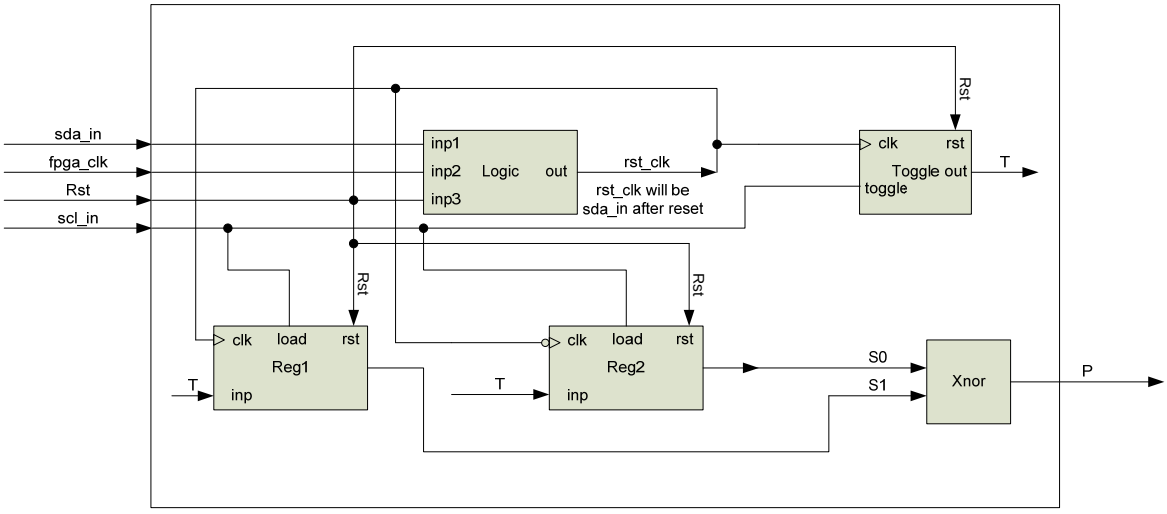


Figure 3.6: Start Stop Detector.

The rst_clk is a combination of three input signals. Initially, when the reset signal is high, rst_clk is actually fpga_clk. Therefore, the three registers will be reset. After the reset signal goes low, the sda_in signal becomes rst_clk signal or clock input of the three registers. Of the three registers, Reg2 is a negative edge triggered register which is sensitive to the negative edge of the sda_in to detect a Stop signal, Reg1 is sensitive to the positive edge of the sda_in to detect a Start signal and the Toggle will change its output if there is a positive edge of the sda_in while SCL line is high. The output of the Toggle (T) is fed to the inputs of the other

two registers. Initially, T is reset to '1'. Table 3.1 describes the operation of the Start/Stop Detector. If P is '1', Slave is in Stop condition, if P is '0', Slave is started. If there is more than one start continuously, or in other words, repeated Start, the Start/Stop Detector still works.

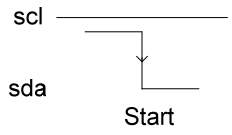
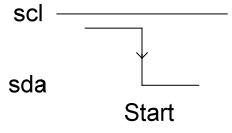
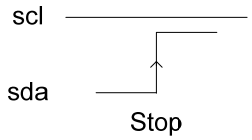
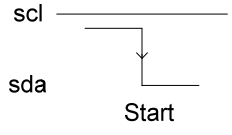
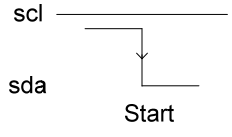
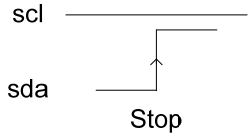
Start/Stop	T (posedge)	S0 (negedge)	S1(posedge)	P = ~(S0^S1)
Initial value	1	0	0	1
	1	1	0	0
	1	1	0	0
	0	1	1	1
	0	0	1	0
	0	0	1	0
	1	0	0	1

Table 3.1: Operation of the Start/Stop Detector.

Sequence Counter

The Sequence Counter is reset to 0 and counts up from 1 to 9 when it receives 7-bit address or reset to 1 and counts up from 1 to 9 when it is in data-transfer-operating mode. Whenever there is a stop, the Sequence Counter is reset to 0, and after counting up to 9, the counter will be loaded with 1 (see Counter_load signal on Figure 3.7). It only counts up when the rising edge of scl_in happens.

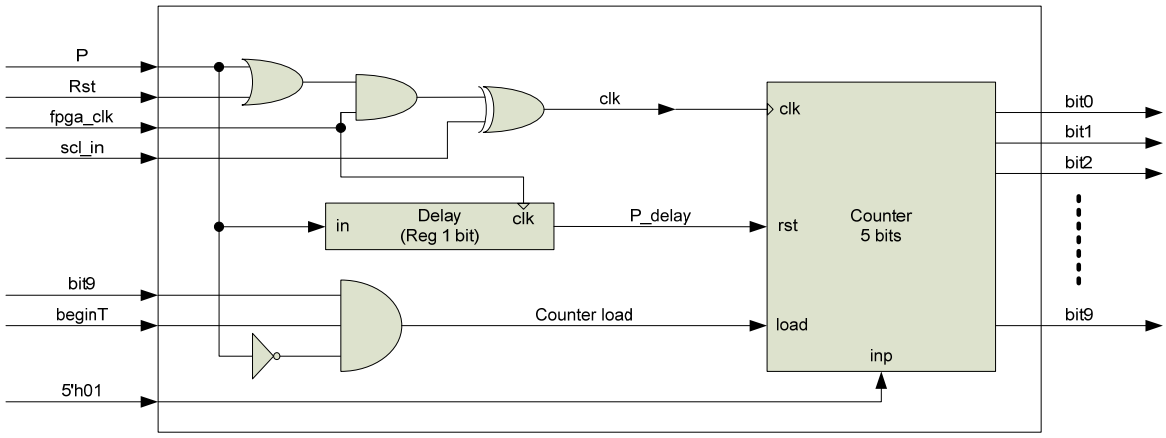


Figure 3.7: Sequence Counter.

Based on the outputs of the Sequence Counter, Slave will determine necessary actions corresponding to the request of Master.

Address Comparator

The Address Comparator has an 8-bit comparator, two registers and two multiplexers. The 8-bit comparator compares Slave address with the content of the Shift Register; the output of the 8-bit comparator is fed to inputs of Reg1 and Reg2 (see Fig. 3.8). We just choose the right time to load the comparator result to the registers.

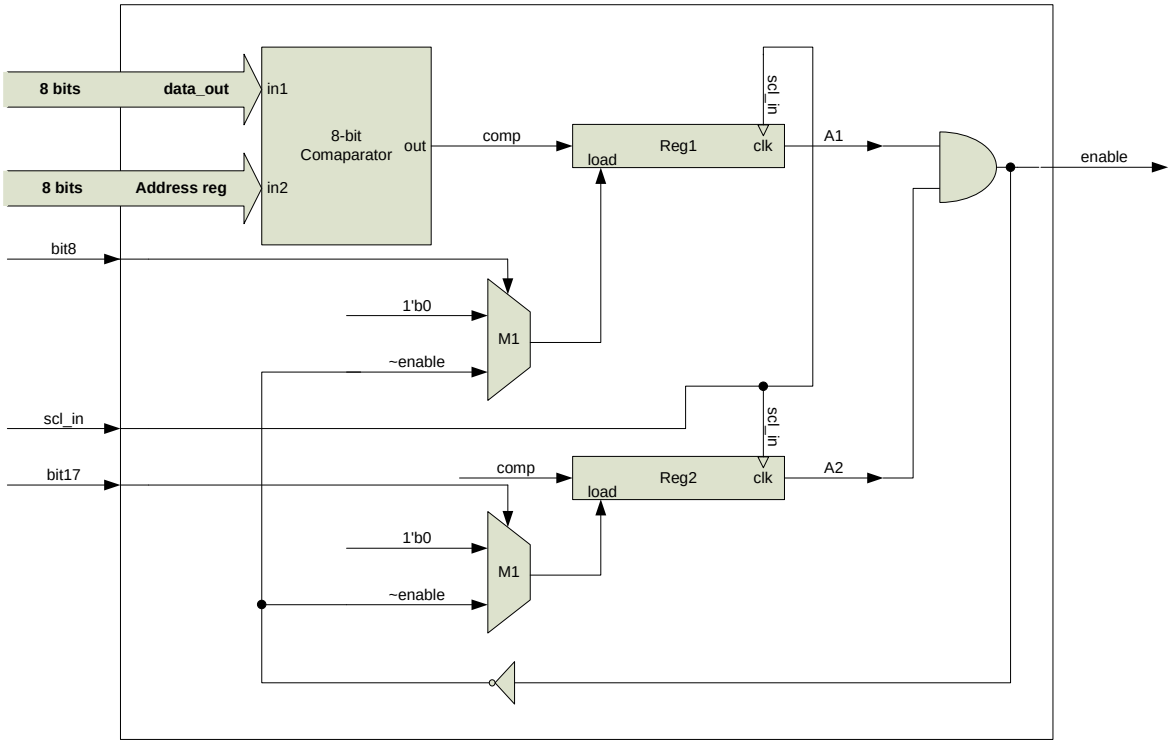


Figure 3.8: Address Comparator .

At bit 8 of the first byte, we load the first address register and at bit 8 of the second byte, we load the second address register (in 10-bit address mode). If the address is correct, the output of the Address Comparator, “enable”, will be high, enabling the Slave.

Acknowledgement Block

The Acknowledgement Block has two main functions: Creating ACK after receiving every byte and detecting Master ACK after sending every byte. The Acknowledgement Block contains two registers and several logic gates. For creating ACK, at the negative edge of the 8th Serial Clock, if the address is correct and Slave is in receiving mode, Slave will release an ACK to acknowledge Master.

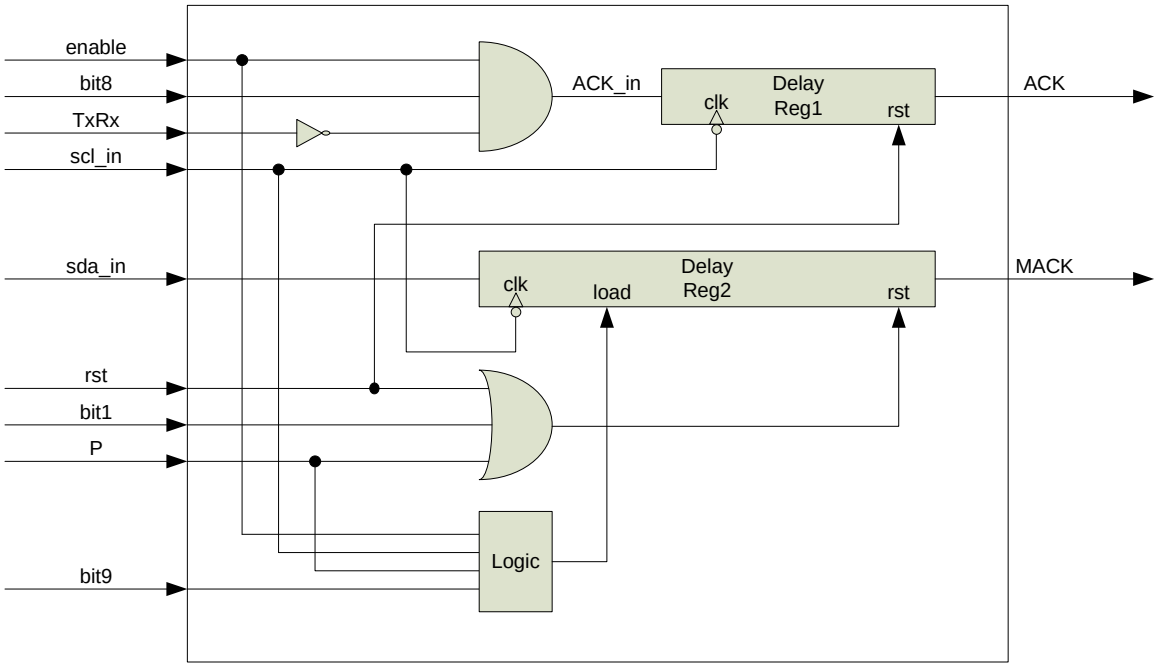


Figure 3.9: ACK Block.

For detecting the Master ACK, Reg2 will be loaded at bit 9 if the Slave is in the transmitting mode. The ACK and MACK will be then used as part of a selection signal for a multiplexer to select either '0' or '1' or the serial data out from the Shift Register to drive the SDA line (see Fig. 3.4).

Master Implementation on FPGA

Basically, an I²C Master is composed of a Slave and some additional components. The I²C Master is connected to a microcontroller (here, a MicroBlaze processor is employed) from which it receives destination address, control signals and data. A Master must have the ability to create the I²C Serial Clock, START and STOP signals. It also has to keep track of the number of transferred bytes to determine an appropriate time to stop a data transfer.

Based on the above characteristics, a shortcut was used to take advantage of the already created Slave. We observed that the Master functions exactly the

same as a Slave after sending the Start Bytes, including the START signal, Slave address, R/W bit, and before creating a STOP signal. Hence, one only needs to design some multiplexers to select appropriate SCL and SDA signals. In addition, obviously, there must be an enhanced Finite State Machine to aid the Master in making decisions, and a Clock Generator to generate the I²C serial clock. A block diagram for the Master is presented in Figure 3.10.

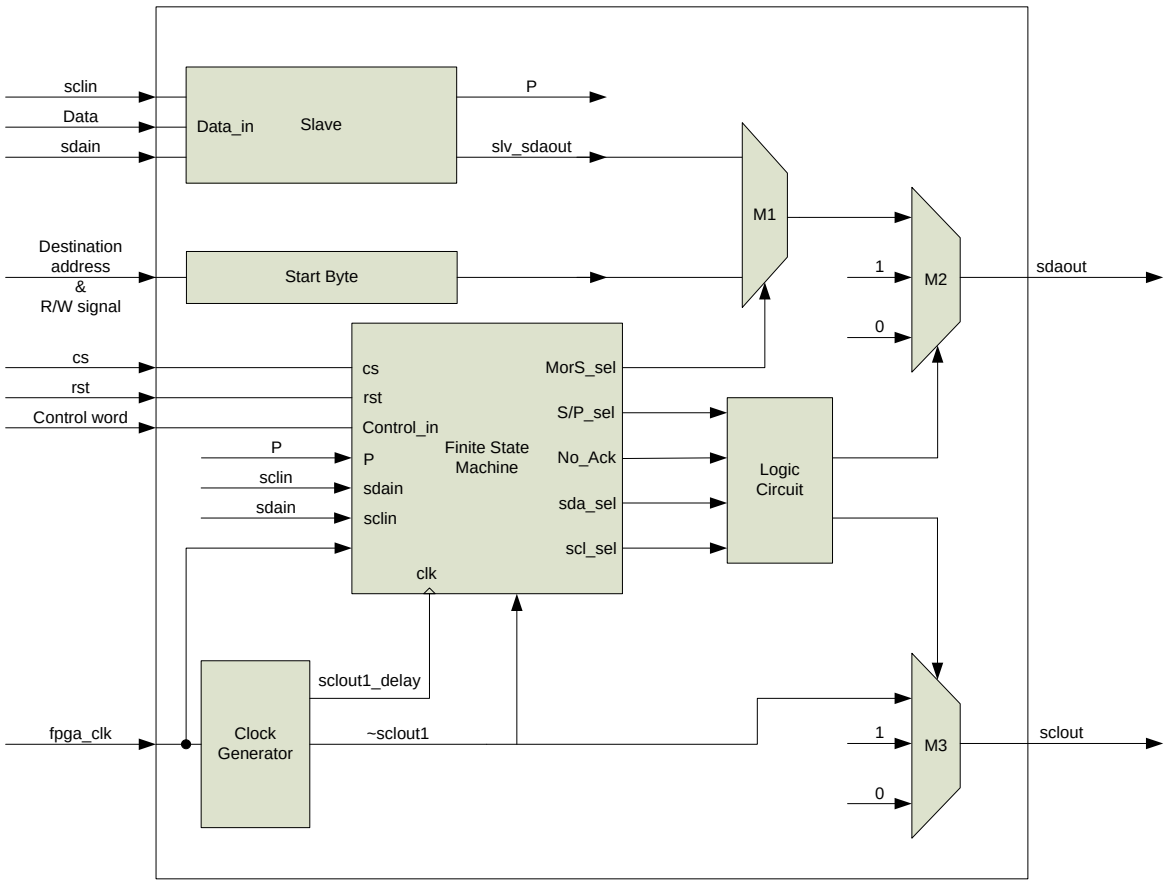


Figure 3.10: Master block diagram.

The I²C Master consists of four major components:

- A Start Byte module.
- A Finite State Machine (FSM).
- A Clock Generator.

- An I²C Slave.

Control signal and data are loaded at reset. Slave address, R/W control bit are loaded to the Start Byte. After being reset, the Master creates a START signal by controlling sda_sel and scl_sel signals, and then it selects the output of the Start Byte to drive the SDA line by raising the MorS_sel signal to high. Meanwhile, the Slave inside the Master is running in the background. It detects the START signal. It also detects the R/W signal. It just does not compare the Slave address with its address, assuming that the address is always correct.

At the time the starting bytes were sent, the Master will now operate exactly as a Slave except that instead of being in the transmitting mode, it will be in the receiving mode and vice versa. The MorS_sel signal will be set to low to select the slv_sdaout as the driver for the SDA line. At the end of the session, the FSM will set the values of selection signals again to initiate the STOP signal.

Start Byte module

The Start Byte module is merely a parallel-to-serial register (see Fig. 3.11).

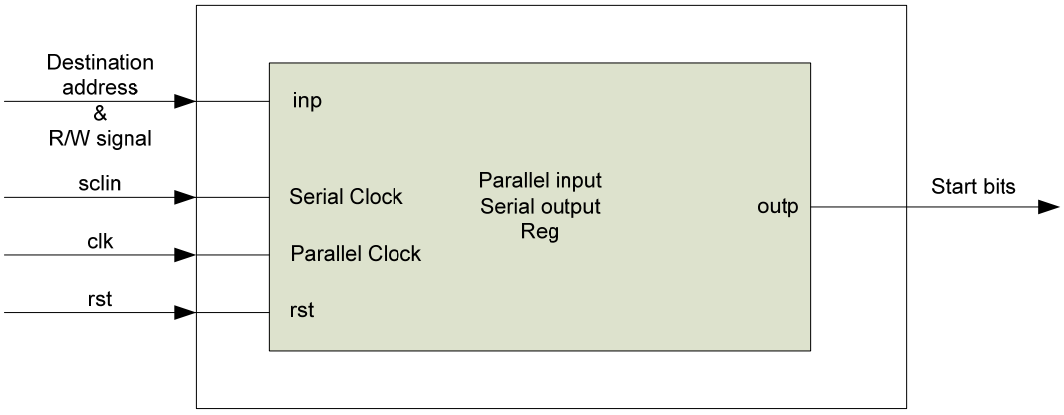


Figure 3.11: Start Byte module.

The clock signal is a combination of sclin (serial clock) and fpga_clk (parallel clock). When reset is high, the parallel clock is selected to set the register to its input value and after resetting the serial clock is selected to shift each bit to the output of the register. The register operates on the negative edge of the clock to guarantee that data will be available before the rising edge of the SCL signal.

Clock Generator

For testing purpose, the I²C Master will be downloaded to a Spartan 3E board, which operates at a frequency of 50MHz. In order to create an approximately 100 KHz SCL signal, we need to divide the board clock signal by 512 times.

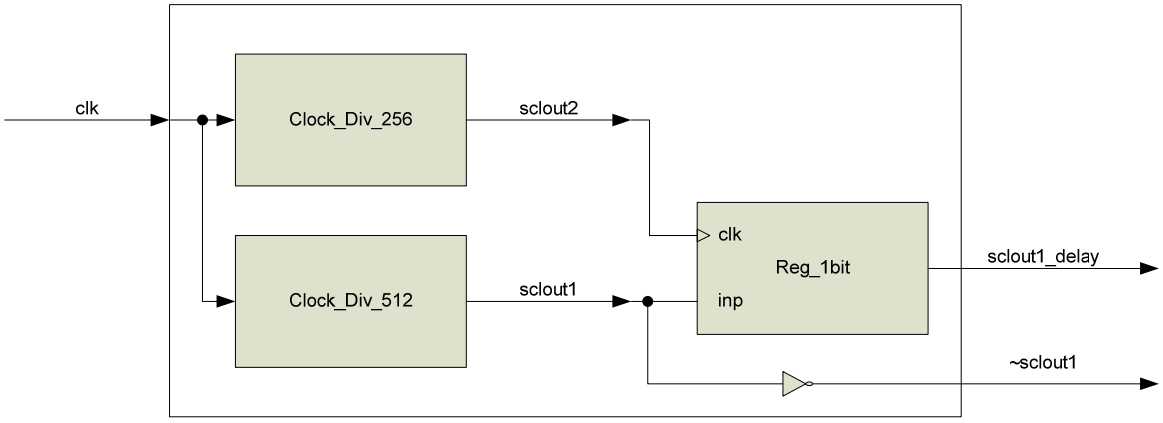


Figure 3.12: Clock Generator.

Except for the START and STOP signal, the I²C Master only changes the SDA signal when the SCL line is low, or equivalently, the Master can change SDA line signal at negative edge of SCL signal. Hence, before negative edges of SCL, the circuit conditions, based on which the Master makes its decisions, must be stable and valid. For this reason, we need another signal in addition to sclout1 called sclout1_delay (see Fig. 3.13). This signal is used as the operating clock signal for

the 32-bit counter inside the FSM. In order to create this signal, in addition to the Clock_Div_512 module, we need a Clock_Div_256 module and a 1-bit register. This clock signal is delayed for ¼ $T_{sclout1}$ compared to sclout1 and used as an input for the Finite State Machine.

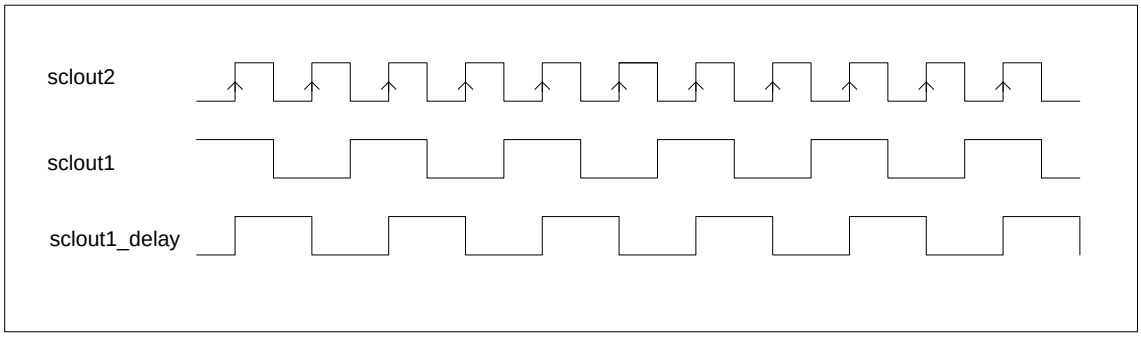


Figure 3.13: Timing diagram of clock signals.

Finite State Machine

The Finite State Machine (FSM) is the most important component in the design of the Master. In the I^2C specifications, the Master must determine the number of bytes to be transferred. However, in this specific application, it is required that Slave will decide number of bytes to be transferred to the Master. To accommodate this requirement, both Master and Slave Microcontroller are slightly changed from its original designs. In the transferring mode (Master sends data to Slave), the FSM gets number of bytes from the Microcontroller before releasing a Start condition.

But in the receiving mode, the FSM gets the number of RAM locations in the first byte received from Slave. After that it holds the SCL line low to freeze the communication, sending number of locations back to MicroBlaze. MicroBlaze will manipulate the number of locations to get number of bits and send it back to Master. After receiving the number of bits from the MicroBlaze, the FSM will use

the number to set its counter, releasing the SCL line and keep communicating with the Slave.

It is important to emphasize that instead of using hardware to manipulate number of locations, which requires a lot of time and resources; I send the number back to MicroBlaze and have it processed by MicroBlaze, which requires less time and resources of the FPGA.

Figure 3.14 describes the schematic of the FSM. The FSM is connected to a control logic, which is connected to MicroBlaze. It receives a control signal from MicroBlaze and based on that, driving its output signals to control the SDA line and the SCL line.

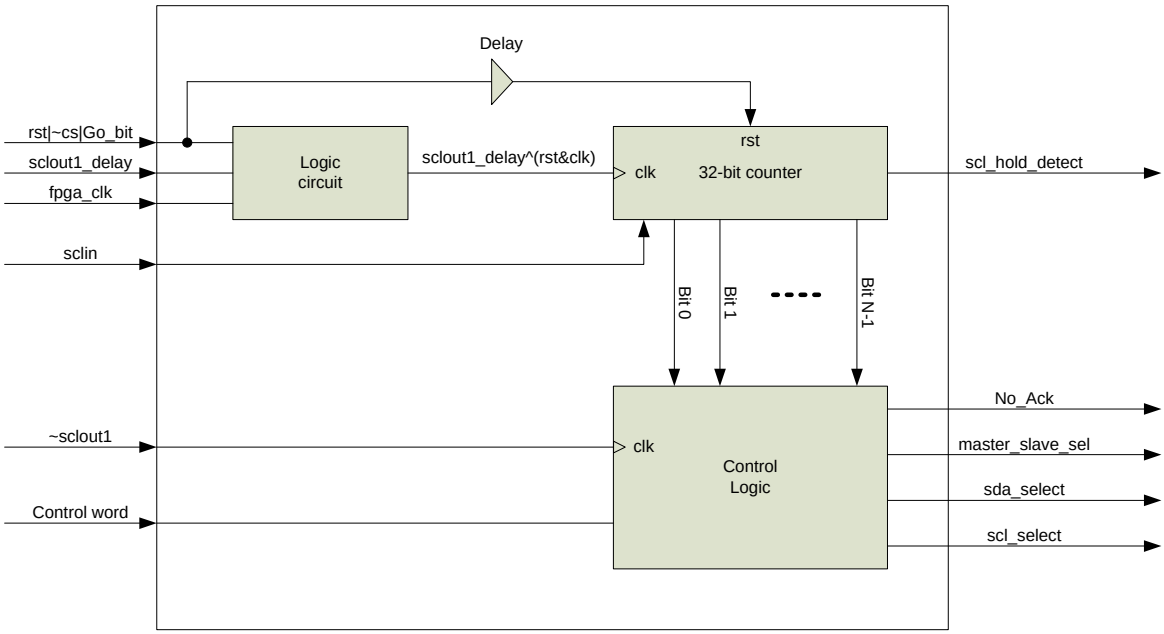


Figure 3.14: Finite State Machine.

The 32-bit counter starts counting up when there is a “Go” signal from the microcontroller. Its clock signal is a combination of fpga_clk and sclout1_delay signals to guarantee that the counter can be reset even if sclout1_delay is not available. After resetting, the counter counts up whenever there is a positive edge

of sclout1_delay. Figure 3.15 illustrated some example operations of the Finite State Machine (FSM). The first example is how the FSM can create a START signal: At negative edge of sclout1, if the output of the counter is “bit 1” then the FSM will drive selection signals to select corresponding values for SDA line and SCL line in order to initiate a START signal.

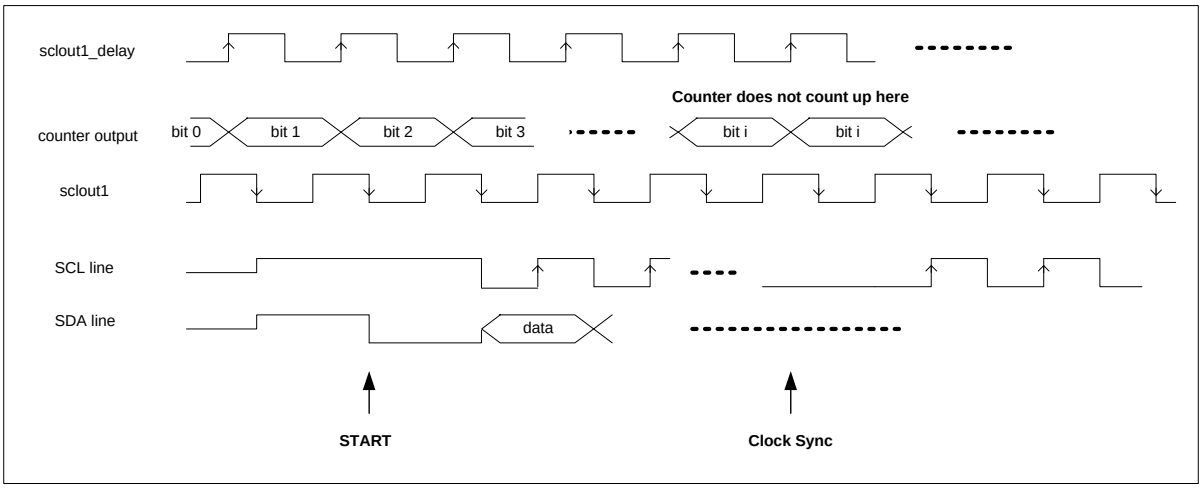


Figure 3.15: FSM controls SDA line and SCL line.

The second example is how the Master deals with clock synchronization. As we know, the I^2C Slave or Master can hold the SCL line low if it is busy and needs more time to process data. In this case, there is no clock on the SCL line, if the counter keeps counting up (because sclout1_delay is still clocking), the FSM will malfunction; for example, it may release a STOP signal while data transferred is not complete.

The second example describes how the FSM deals with this error. On the positive edge of sclout1_delay, whenever the FSM detects that the SCL line is low, it will hold the counter at its current value and wait until the SCL line goes high again (when Slave releases the SCL line). The sclout1_delay is then necessary; otherwise, if we use sclout1 as the clock for the counter, we cannot determine if the

Slave is holding the SCL line because SCL is actually a delayed version of sclout1 (the inverted sclout1 drives the gate of a FET transistor, which has its drain connected to the SCL line). For more information, please refer to Appendix B, SandP_Control module.

I²C Microcontroller

The I²C Microcontroller basically contains two user-defined components: a C Program executed by the MicroBlaze and a Control Logic component. In order to interface the created Master to the MicroBlaze, we need to create a User Peripheral by using the EDK software. The EDK automatically create a hardware component named Control Logic (see Fig 3.16). The Control Logic is connected to the MicroBlaze using Processor Local Bus (PLB). It contains two parts: a Wrapper (created in VHDL, see file master5.vhd in Appendix C) and a user logic part (created in Verilog, refer to file user_logic.v in Appendix C).

The Wrapper has some standard PLB I/O ports to connect to the PLB and some user defined I/O ports connected to the user logic component. Control data is sent from the MicroBlaze to the Wrapper using the PLB, stored in the user logic module and delivered to the Master. In the other direction, interrupts and data are sent from the Master to the MicroBlaze to inform the MicroBlaze about hardware status.

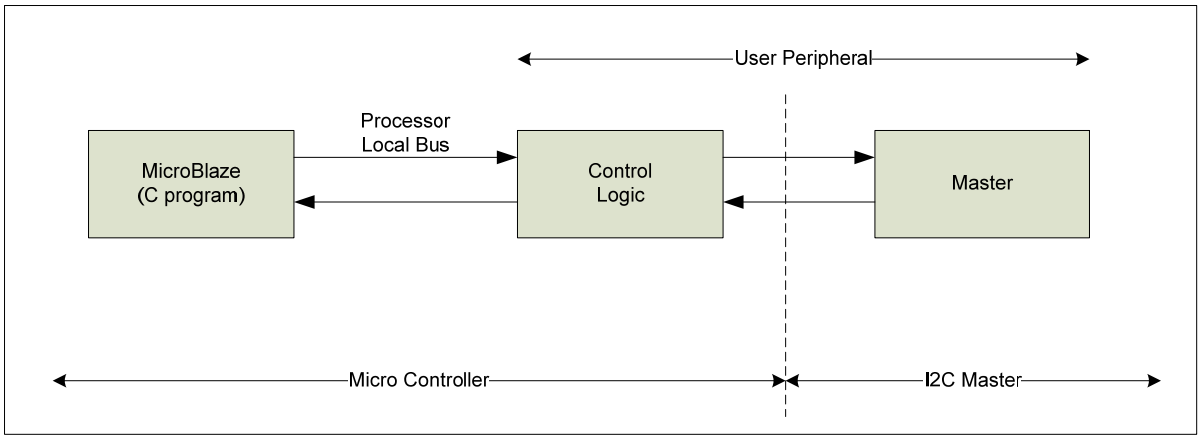


Figure 3.16: Microcontroller for Master.

The user logic is automatically created by the EDK. It contains sixteen 32-bit User Software (S/W) registers and methods for accessing those registers. It provides a very effective way for a user to access the registers from software just by writing a C Program. To interface our I²C Master to the MicroBlaze, we have to modify this component make it become a Master Controller. In the following sections, we will discuss in more detail about the three components of the I²C Microcontroller, the Wrapper, the Master Controller and the C Program.

C Program

The C Program has two main responsibilities. First, it has to control the Master to receive/send data from/to the Slaves and then it has to send the received data via the RS232 port to a host computer. In this chapter, we will only discuss the first function, (controlling the Master) the second one will be described in Chapter 4. Figure 2.23 shows the S/W registers and the function of each of their bits.

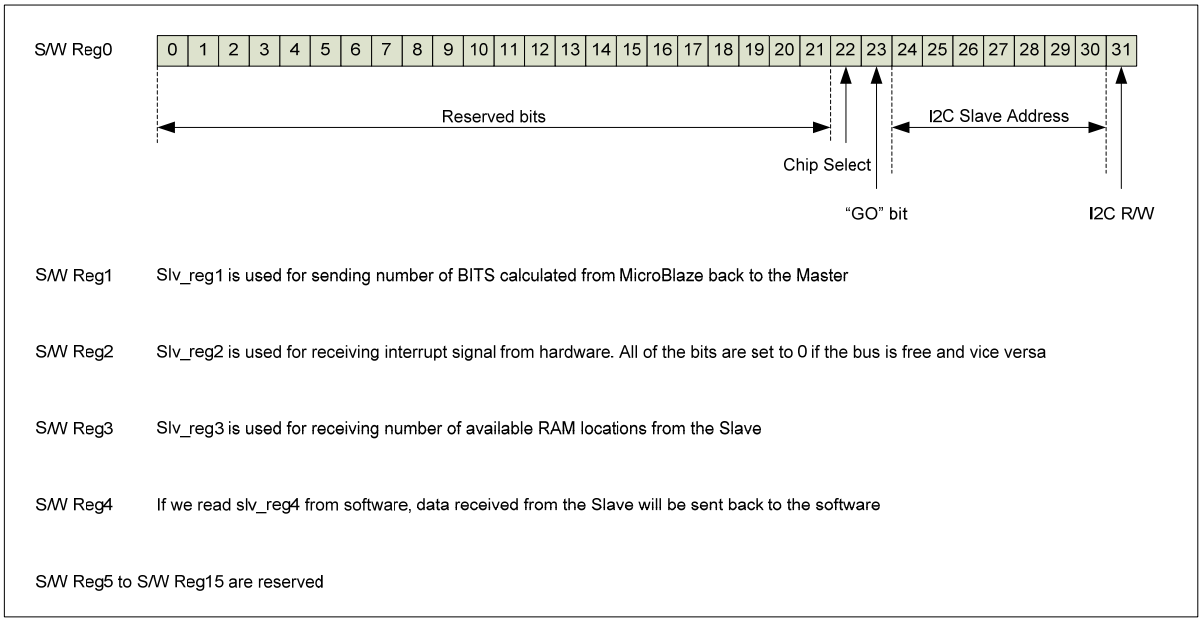


Figure 3.17: S/W registers

Initially, the C Program evaluates the interrupt signal, which is the status of the I²C bus, to determine whether to start a new communication. If the bus is free, it first sends a Slave address that it wants to connect to by sending an address value to one of the S/W registers. Then, it selects the Master by raising a chip select value to high and initiates a "GO" bit, signaling the Master to start communicating.

It is then up to the Slave to decide the number of bytes to be transferred. After receiving the correct address, the Slave will send the number of bytes available in the RAM to the Master. Thus, after initiating a "GO" command, the Master is in a "waiting" mode, monitoring the S/W Reg3 (see Fig. 3.17), and receiving the number of RAM locations available. Recall that each RAM location is 18-bit wide or round up to 3-byte long. Based on the number of RAM locations, the software has to calculate the number of bits to be transferred and send it back to the Master to set the Master's Counter inside the Master FSM.

The Master will freeze the communication by holding the SCL line Low until it receives the number of bits. After getting the number of bits, the Master will restart its communication with the Slave, store received data in its buffer and send the data to the software using S/W Reg4. The I²C transaction ends here after the software receives the data from the Slave.

The Wrapper

The Wrapper functions as an interface between the User Peripheral and the PLB Bus. It includes an IPIF module and its main function is to map the PLB standard signals to user friendly signals. Figure 2.22 illustrated the Control Logic.

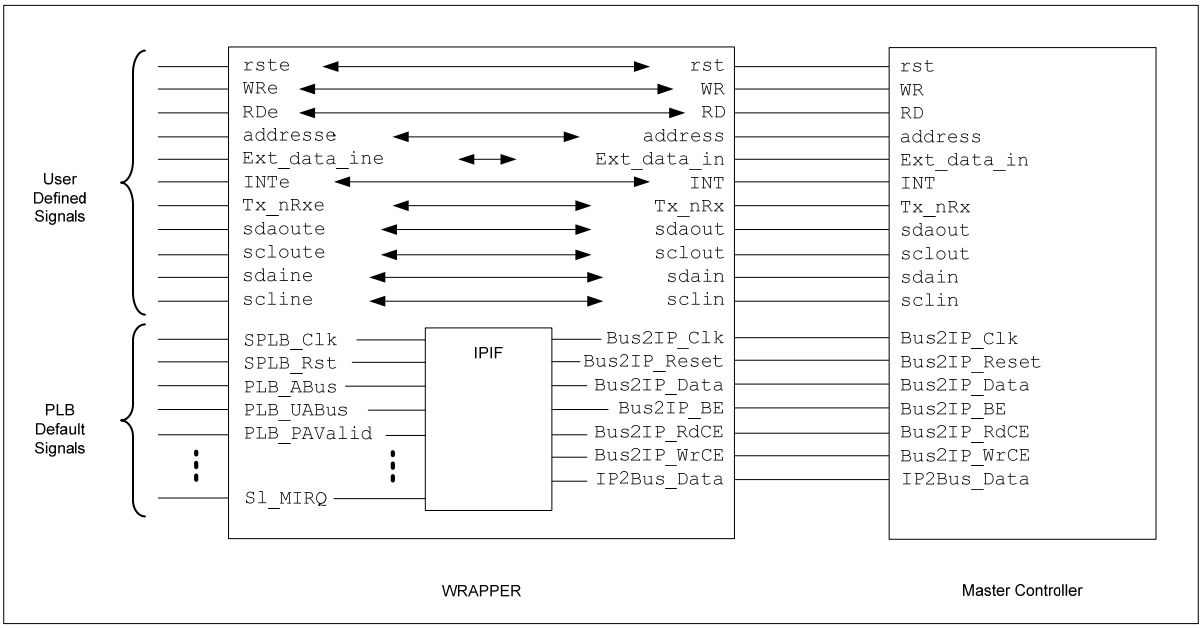


Figure 3.18: Control Logic.

As you can see in Figure 3.18, there are two types of ports for the Wrapper. The first type is user-defined port; including rste, WRe, RDe, etc. These ports are made external to connect to Spartan 3E board I/O pins from which, we hook our I²C signals, reset signal and some other necessary signals. The other port type is

the PLB default port which includes SPLB_Clk, SPLB_Rst, etc. Those signals are fed to the Intellectual Property Interface (IPIF), which is automatically created by the EDK software, to produce the user friendly signals: Bus2IP_Clk, Bus2IP_Reset, *etc.* These signals are very important because they are the only mean for us to control our hardware from a C Program, which is compiled by the EDK software and executed by the MicroBlaze. (See Table3.2)

Table 3.2 lists the user friendly signals and their functions:

Name	Function
`Bus2IP_Clk`	Bus to IP clock, this is the 50MHz clock signal.
`Bus2IP_Reset`	Bus to IP reset
`Bus2IP_Data`	Bus to IP data bus, we have to use this bus if we want to transfer data from the MicroBlaze to the User Peripheral.
`Bus2IP_BE`	Bus to IP byte enables
`Bus2IP_RdCE`	Bus to IP read chip enable. This signal selects an S/W register to read data from.
`Bus2IP_WrCE`	Bus to IP write chip enable. This signal selects an S/W register to write data to.
`IP2Bus_Data`	IP to Bus data bus, we have to use this bus if we want to transfer data from the MicroBlaze to the User Peripheral.
`IP2Bus_RdAck`	IP to Bus read transfer acknowledgement
`IP2Bus_WrAck`	IP to Bus write transfer acknowledgement
`IP2Bus_Error`	IP to Bus error response

Table 3.2: IPIF signals.

Master Controller

Recall that the Master Controller is a modified version of an automatically created Verilog file named “user_logic.v”. The Master Controller has 16 software-accessible S/W registers from which, a user can get hardware status, data, *etc.* and to which, the user can send his command to control the Master. The interface between the Master Controller and the Master is illustrated in Figure 3.19.

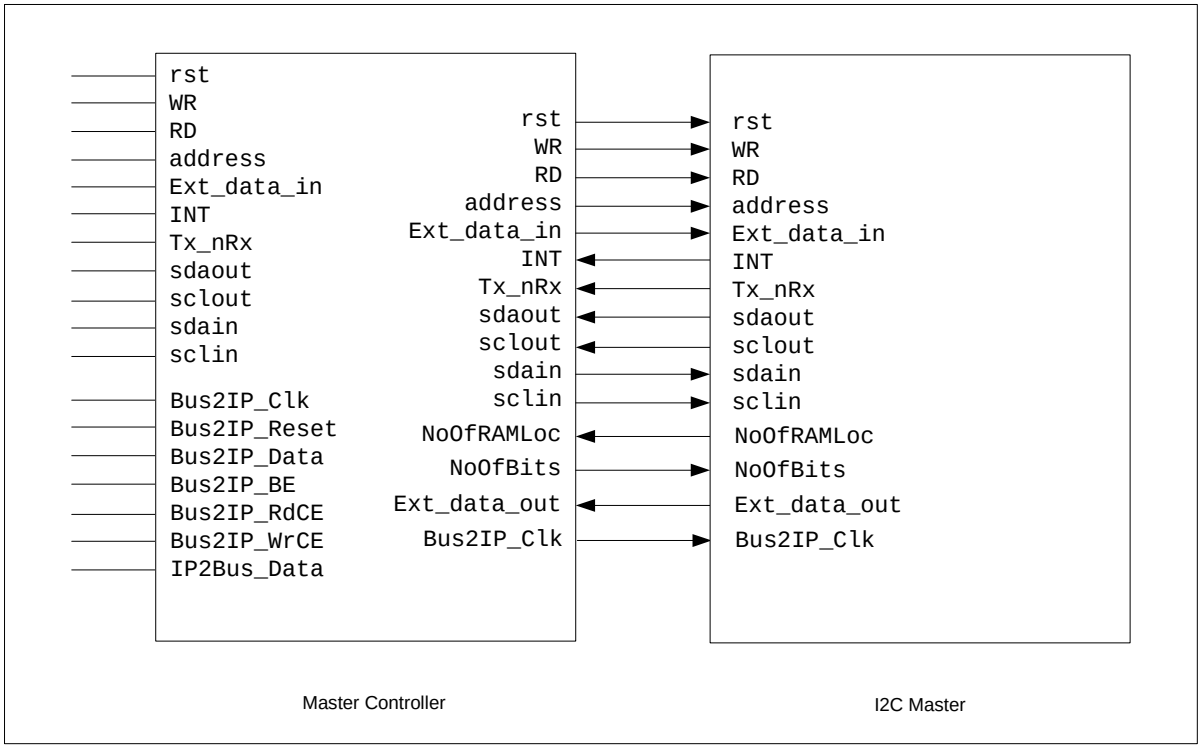


Figure 3.19: Interface between the Master Controller and the I²C Master.

The Master Controller receives control data from the software via Bus2IP_Data port and sends it to the I²C Master. In the other direction, it receives I²C data, interrupt signals, I²C signals and sends them either to the C Program or to the external I/O pins. The I²C Master Controller also includes a 96 location by 80bit wide Buffer to store data received from the I²C Slaves. The Buffer has three clock signals, one for resetting, one for writing and the other one for reading. We

need three clock signals because reading the content of the Buffer to send it to the MicroBlaze requires a software clock while writing data to the Buffer requires a clock that is synchronized with the SCL clock signal. Data received from the Slaves will be stored here before sending back to the C Program.

CHAPTER 4

PROTOTYPE SYSTEM

In Chapter 3, we have described the implementation of a Master microcontroller, an I²C Master, and an I²C Slave. Here in Chapter 4, a prototype system will be introduced to integrate the I²C Interface onto an emulated PSD8C chip (using an FPGA) and prove that the I²C Interface is likely to work on the PSD8C chip.

As we know, in the PSD8C chip, the analog output signal will be converted to digital signal using an on-chip ADC and stored in an on-chip RAM. To emulate the chip, we will take advantage of the ADC on the Spartan 3E board to sample an input signal, store the result in a RAM and transfer data to a host computer using an I²C bus.

Prototype System

Figure 4.1 illustrates how the prototype system works. At first, the ADC's will be reset to sample two analog input signals and the results will be stored in two separate RAMs which are connected to two Slaves, Slave1 and Slave2. The Master, downloaded to another board, is then reset to start communicating with either Slave1 or Slave2 and transfer data it receives over the RS232 DCE port to a host computer. On the host computer, a C# Program is created to receive data from its RS232 DTE port, convert digital signal back to an analog format and display the signal as a waveform on a display screen. The C# Program has an option for a user to select which channel (Slave 1 or Slave 2) to display.

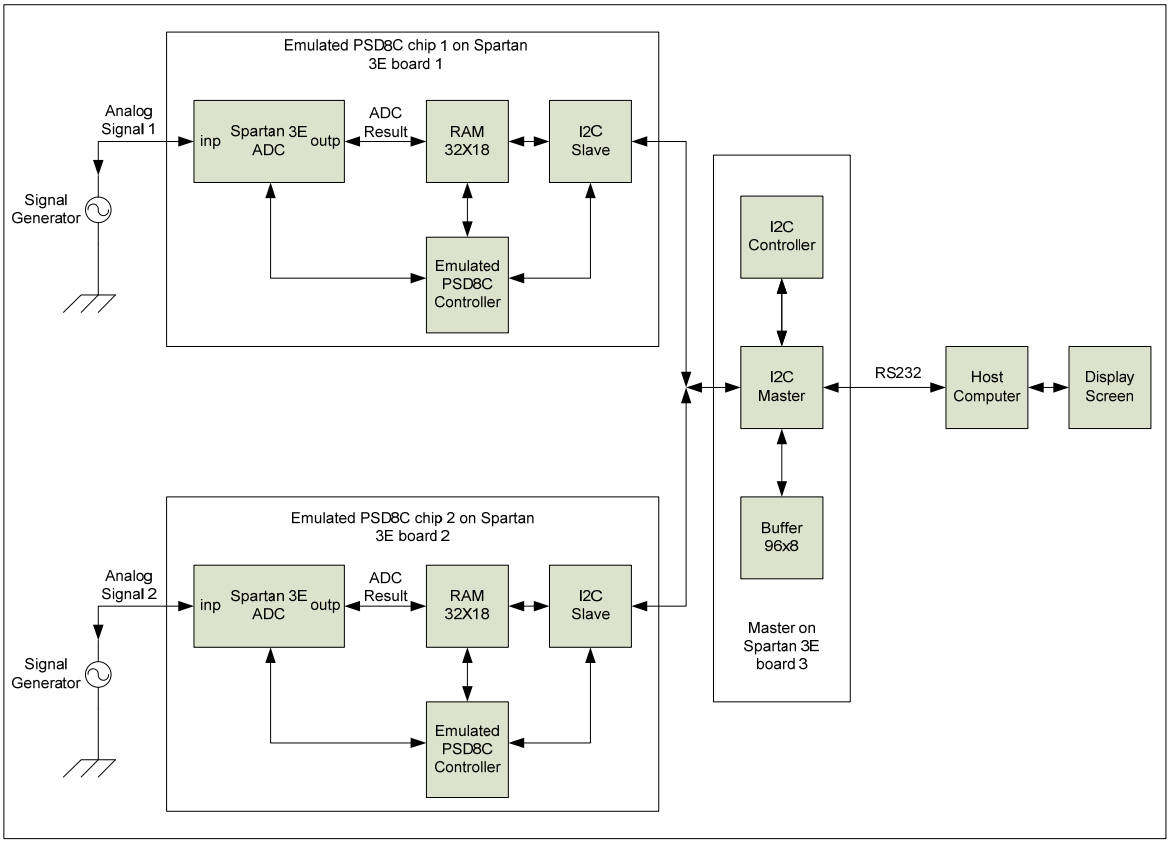


Figure 4.1: Prototype System.

Analog input signal 1 and analog input signal 2 will be different so that the user can differentiate the two signals by seeing the result waveforms on the screen. Once again the prototype system includes the flowing components:

- Two emulated PSD8C chips.
- An interface between the host computer and the Master.
- C# program running on the host computer for receiving data via the RS232 port and drawing waveform based on the received data.
- An I^2C Master controlled by a MicroBlaze processor.

PSD8C Chip Emulator FPGA

As shown in Figure 4.1, an emulated PSD8C chip includes the following components:

- An Analog Capture Circuit on the Spartan 3E board and an ADC control module to control the circuit.
- A 32 location by 18 bit RAM.
- An I^2C Slave.
- An Emulated PSD8C Controller to control the above components.

An analog signal is converted by the Analog Capture Circuit to a 14-bit digital representation which is in turn written into the RAM. The 18-bit output of the RAM will be divided into three bytes, each byte is 8-bit long and there is a 4-input multiplexer to select either of the three bytes to the data input of the I^2C Slave.

Spartan 3E Analog Capture Circuit

The Analog Capture Circuit on the Spartan 3E board consists of two ADC channels. Each channel has a programmable input gain amplifier and produces a 14-bit 2's complement output as shown in Figure 4.2. The 14-bit output is expressed by the following equation:

$$D[13:0] = GAIN * \frac{(Vin - 1.65v)}{1.25v} * 8192$$

The GAIN is obtained from an ADC Control Module stored in the FPGA. The input voltage is centered around 1.65 Volts, and the maximum range of the ADC is $\pm$ 1.25 Volts. The FPGA is connected to the amplifiers using a SPI Control Interface

[Xil:08]. Table 4.1 lists the signals on the interface and their corresponding pins on the FPGA.

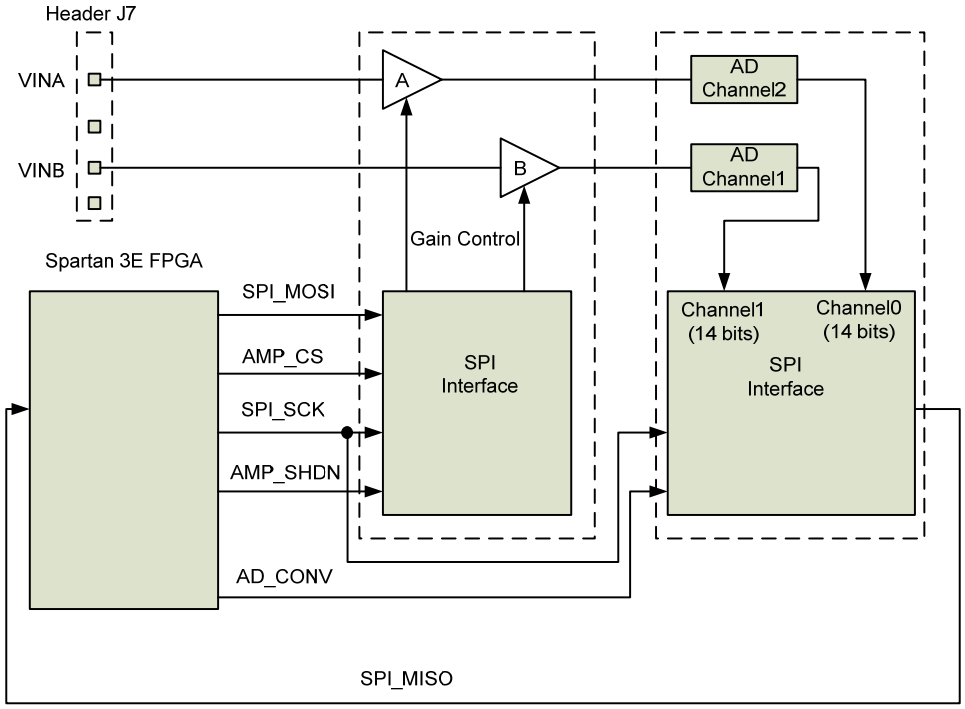


Figure 4.2: Analog Capture Circuit [Xil:08].

Signal	FPGA Pin	Direction	Description
SPI_MOSI	T4	FPGA → AD	Serial data: Presents 8-bit programmable gain settings.
AMP_CS	N7	FPGA → AMP	Active-low chip select.
SPI_SCK	U16	FPGA → AMP	Clock.
AMP_SHDN	P7	FPGA → AMP	Active-high shutdown, reset.
AMP_DOUT	E18	FPGA → AMP	Serial data. Echoes previous amplifier gain.

Tabel 4.1: Amplifier Interface signals [Xil:08].

We also have control over the ADC by driving the following interface signals shown in Table 4.2.

Signal	FPGA pin	Direction	Description
SPI_SCK	U16	FPGA → ADC	Clock.
AD_CONV	P11	FPGA → ADC	Active-high shutdown and reset.
SPI_MISO	N10	FPGA ← ADC	Presents 14-bit output of the ADC.

Tabel 4.2: ADC Interface signals [Xil:08].

The output of the ADC is fed back to the FPGA using SPI_MISO and stored in the RAM at the appropriate times.

A bus transaction is illustrated in Figure 4.3. When the AD_CONV signal is high, the circuit is reset. When it is low, the circuit starts operating. An A/D conversion requires 34 clock cycles to finish.

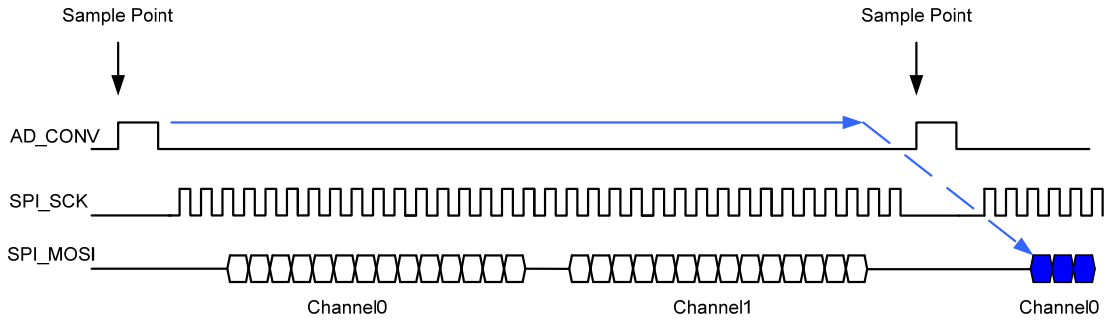


Figure 4.3: ADC timing [Xil:08].

As shown in Figure 4.3, both ADC channels are placed on the SPI_MISO bus serially. However, the results of this conversion are not presented until the next time AD_CONV is asserted, introducing a latency of one sample. The first valid

output bit is available after the rising edge of the 3rd SPI_SCK clock signal (8ns) [Xil:08].

The ADC Control Module is created to generate input signals for the SPI Control Interface and the ADC. It also has the ability to store the ADC output and send it to the RAM. The module has to satisfy timing constrains as shown in Figure 4.3. The ADC Controller is downloaded to the Spartan 3E FPGA. The reader should refer to Figure 4.2 for the connections between the ADC Controller and the SPI Interface. A block diagram for ADC Controller is presented in Figure 4.4.

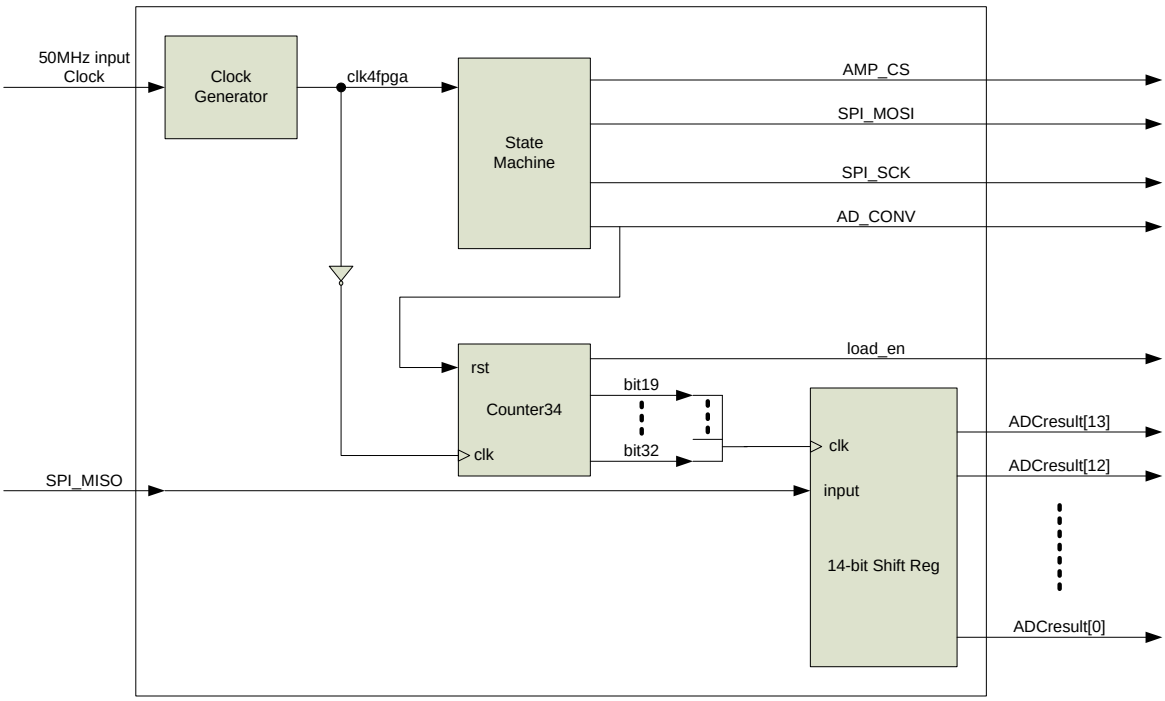


Figure 4.4: ADC Controller.

In Figure 4.5, a Clock Generator generates clock signal for a State Machine from the 50MHz input clock. The State Machine has four states, based on which it will control its output signals, AMP_CS, AD_CONV, SPI_SCK and SPI_MOSI. There is also a Counter. The Counter is reset each time the ADC_CONV signal goes high.

Thirty-four clock cycles is the time required to finish one period of the Analog Capture Circuit.

The ADC result is fed back to the FPGA via signal SPI_MISO. This signal is connected to an input of a 14-bit Shift Register, which always shifts to the right. That means we will load the ADC result at the negative edge of the 19th clock, 20th clock …, 32nd clock, or in other words, ADC result stored in Channel1 will be loaded to the 14-bit Shift Register. We want to load Channel1 because the analog input signal is taken from VINB (see Fig. 4.2; hence, a 14-bit ADC result will be stored in Channel1. In addition to the 14-bit ADC result output, there is a "load_en" output to inform that the 14-bit ADC result is ready to load. Based on this signal, the 32x18 RAM is able to determine appropriate times to load the ADC result.

32x18 RAM

The RAM is instantiated from a RAM primitive in the Spartan 3E Libraries called RAM32X1S. RAM32X1S primitive is a 32-word by 1-bit static random access memory with synchronous write capability (Fig. 4.6).

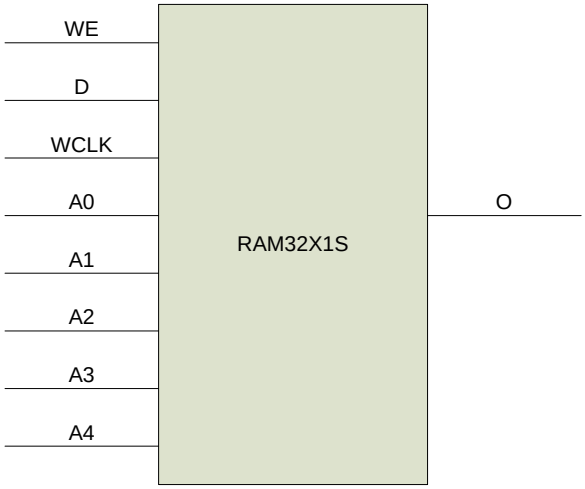


Figure 4.5: RAM32X1S.

When the write enable (WE) is set Low, data stored in the RAM is not affected. When WE is set High, any positive transition on WCLK loads the data on the data input (D) into the word selected by the 5-bit address (A4 – A0). Address and data inputs must be stable before a Low-to-High WCLK transition. The signal output on the data output pin (O) is the data that is stored in the RAM at the location defined by the values on the address pins [Xil:07].

Emulated PSD8C Controller

The Emulated PSD8C Controller has three outputs, the Address and RnW signals for the RAM, and a selection signal for a 4-to-1 Multiplexer. The Controller is operating on the positive edge of its clock. There are three clock signals for the controller but only one is selected at a time.

Initially, when Reset is high, the 50MHz clock is selected for resetting. In this mode, the Controller will set the Address bits to low and the ADC enable (ADC_en) signals to high. Setting ADC_en to high will disable the I^2C Slave.

Next, when Reset is low, an inverted AD_CONV signal will be selected as the clock to increment the address of the RAM on the negative edge of the AD_CONV signal, thus pointing to the next location to write the ADC output to the RAM. Notice that the RAM transfers occur on the positive edge of the AD_CONV signal (see Fig. 4.7); hence, we need to change the address before the positive edge of AD_CONV or specifically, we choose to increment the address at the negative edge of AD_CONV signal.

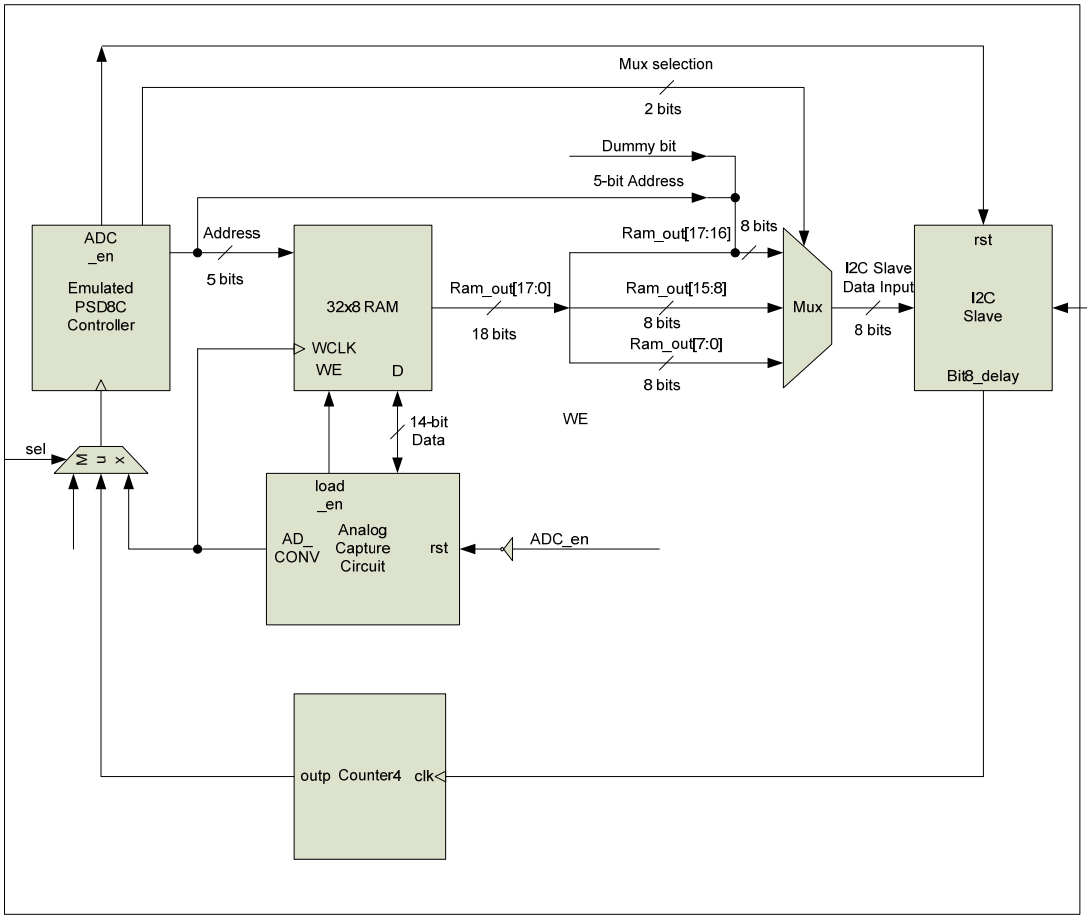


Figure 4.6: A detailed view of the PSD8C Emulated Chip.

Finally, when the RAM is full, ADC_en will be set to low, enabling the I²C Slave and disabling the ADC. In this mode, another clock related to the I²C SCL clock will be selected to sequentially decrement the RAM address to read all the ADC outputs and send them to the Master. This clock is generated from a 2-bit counter (Counter4, see Fig. 4.7. The clock is high every time the I²C Slave finishes sending 3 bytes to the Master. At that moment, we need to decrement the RAM address to read the next location because each RAM location is equivalent to 3 bytes.

Data in the RAM is sent out via the Ram_out bus. The bus is 18-bit wide but the data input for the I²C Slave is 8-bit wide (or a byte). For this reason, we need

to divide the bus into three bytes; each byte is 8-bits wide (see Fig. 4.7). The first byte consists of a dummy bit, a 5-bit address, and the two most significant bits from the bus. The 5-bit address is useful for the Master to calculate the number of bytes available in the buffer of the Slave. The next two bytes contain the other 16 bits of the bus. Of the three bytes, only one is selected by a Multiplexer to be transferred to the I^2C Master at anytime.

Although we multiplex the clock signal, we do not have any problem because the three clock signals above are mutually exclusive.

Host Computer Interface FPGA

A host computer is connected to the board using the RS232 port. A C# Program running on the host computer will request the board to establish a connection with a specific Slave by sending the Slave address to the board. The software running on the board gets the Slave address and sends it to the Master to initiate a communication with the selected Slave. After getting back data from the Slave, the software on the board will send the data to the host computer. On the board, this procedure is implemented by using some built-in functions provided by Xilinx in the library `"xuartlite.h"`. The following functions were used:

- `XUartLite_Initialize()`: Initializes the UartLite, specifically the RS232 port.
- `XUartLite_Recv()`: Receives data from RS232 port.
- `XUartLite_Send()`: Sends data to RS232 port.

C# Program Running on Host

The C# Program was specifically written for this application.

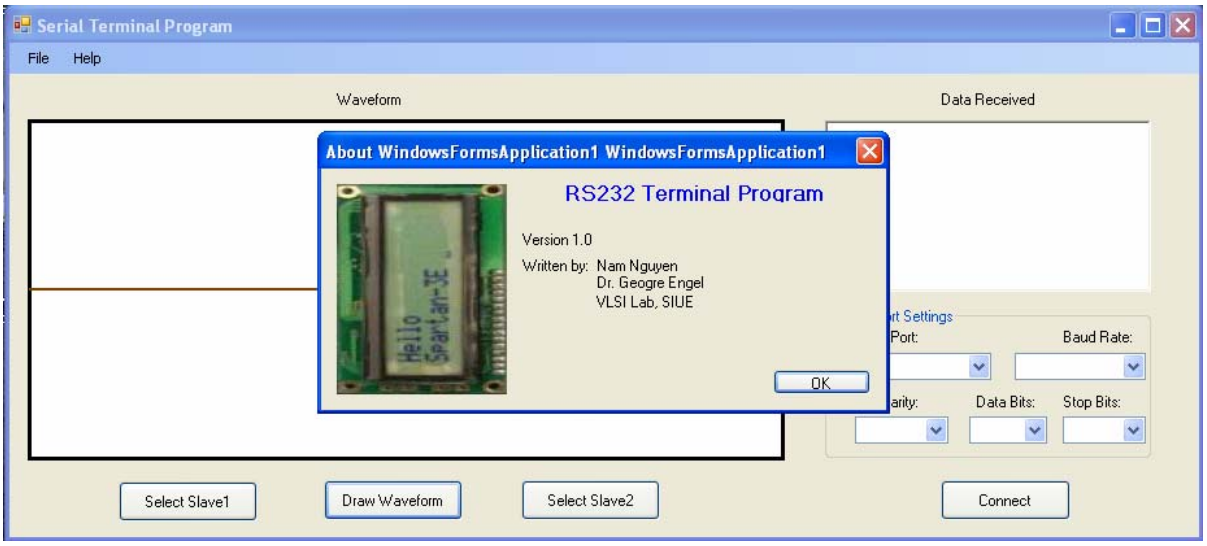


Figure 4.7: About the Program.

The "Serial Terminal Program" has three main functions:

- It must establish a terminal connection with the serial port on the computer. User can configure the port by selecting which serial port to connect to, the transmitting baud rate, parity check, number of data bits per one session and number of stop bits to be used (see Fig. 4.9).
- It must select which Slave to communicate with, either Slave1 or Slave2. In the prototype system, there are two Slaves with address of 1 and 2 correspondingly. If a user clicks on "Select Slave1", the Program will send "1" to the Spartan 3E board. If he clicks on "Select Slave2", the number "2" will be sent to the board. The I^2C Master Microcontroller will then send the address to the I^2C bus. After all of the data is received from the slave, the Microcontroller will send the data back to the host via the RS232 port. At that time, the C# Program will collect the data, storing it inside a buffer.
- It must draw waveforms on the screen based on received data: The program will draw the waveform based on data in the buffer on the screen. A user

can look at the analog input signal of the ADC from the Slaves and compare it to the waveform on the screen.

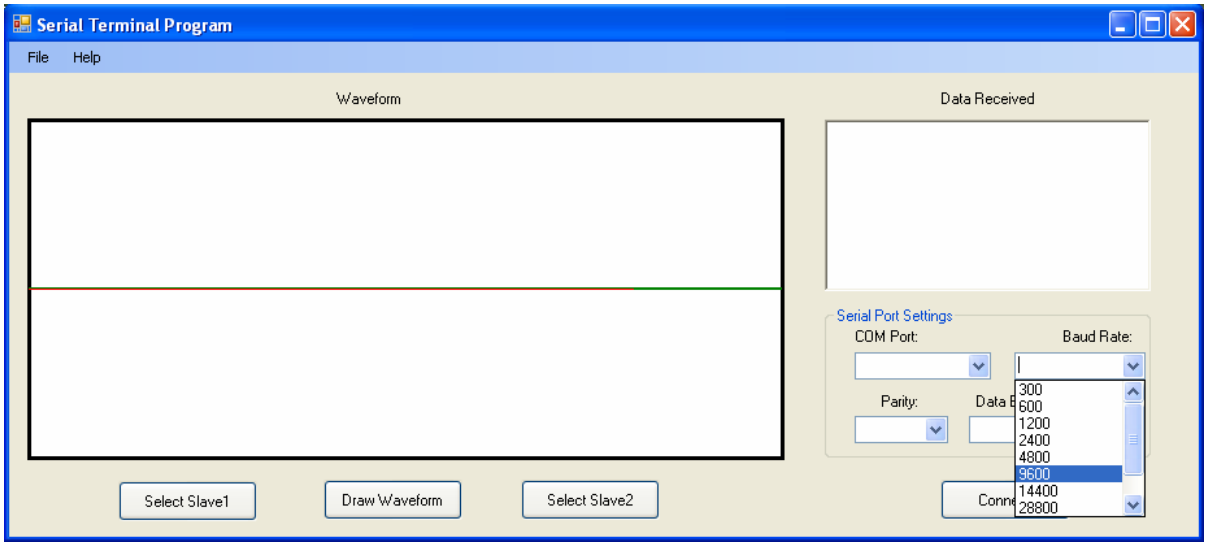


Figure 4.8: Program user interface.

Testing of Prototype System

Figure 4.9 illustrates the setup used testing the prototype. There are three Spartan 3E boards; one for the Master, and the other ones are for the Slaves. There is a signal generator connected to inputs of two ADCs on the two boards. There are also two PSoCs, the first one has one Slave on that with address of 3 and the second one is used for resetting the whole circuit. The physical I²C bus is built on an extra breadboard and on the breadboard of one PSoC. There are two pull-up transistors, connecting the I²C bus to 3.3 Volts. There are also some transistors functioning as wire-OR devices for the SCL and SDA lines.

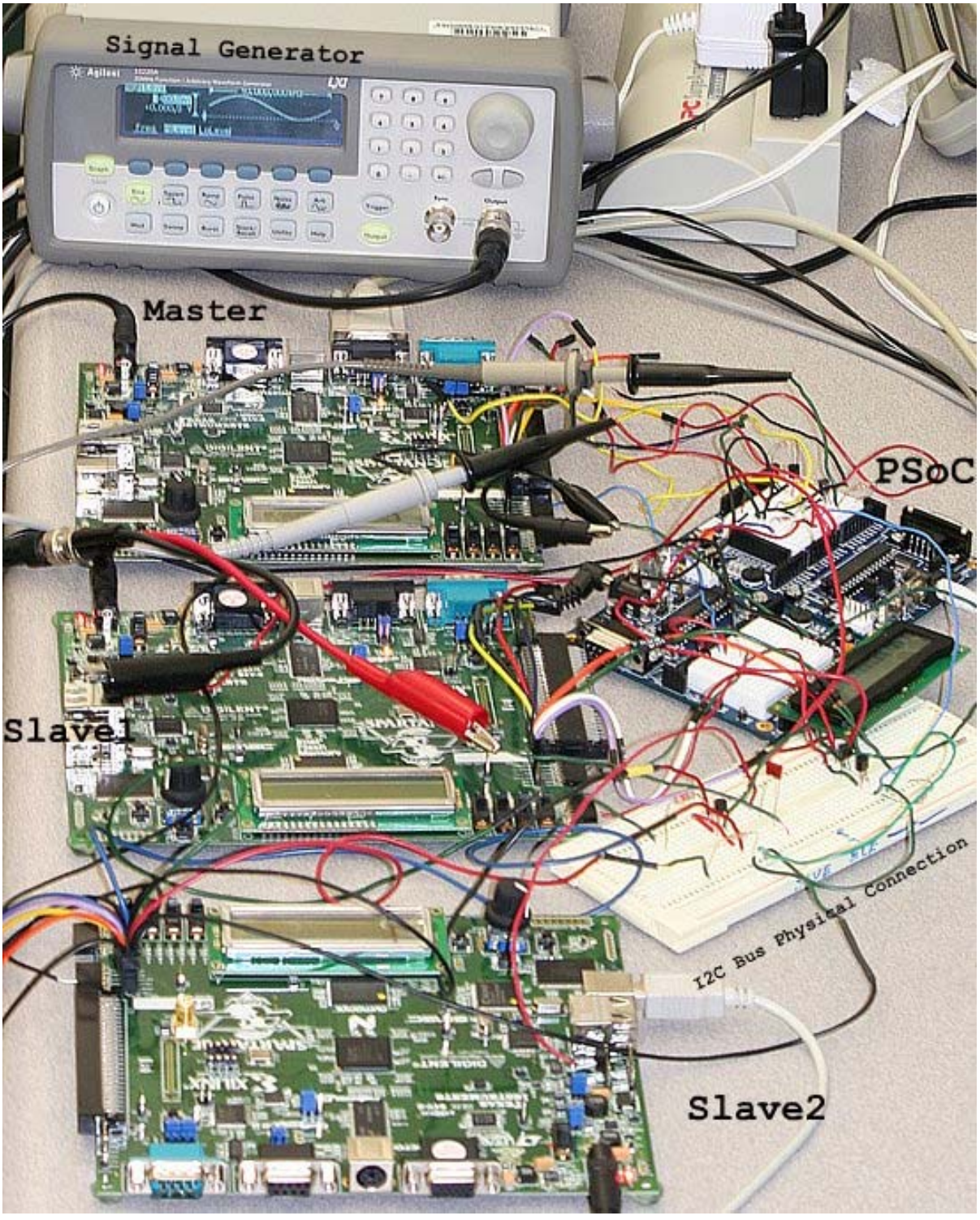


Figure 4.9: Testing Prototype System.

First, the user must press the reset button on the second PSoC to reset the Slaves and the Master. After pressing the button, the ADC on the two Slaves will start operating, sampling the analog input generated by the signal generator until the RAM is full. After that, Slave 1 and Slave 2 will be in "waiting" mode.

Then, the user has to depress the South Button (labeled BTN_SOUTH, pin K17) on the Spartan 3E board of the Master to reset the Master (see Figure 4.10). This will reset the Master and force it into a "waiting" state. The Master is now waiting for the Slave address to be sent from the host computer.

Finally, the user can select either Slave 1 or Slave 2 by clicking on the corresponding buttons on the program user interface. By clicking on either Select Slave1 or select Slave2, the user will send the Slave address to the Master and the Master will in turn initiate a communication with the selected Slave. Figure 4.10 shows the output waveform on the screen of the host computer. As you can see, it is exactly the same as the waveform displayed on the Signal Generator; the data sent from the Slave was received correctly by the Master.

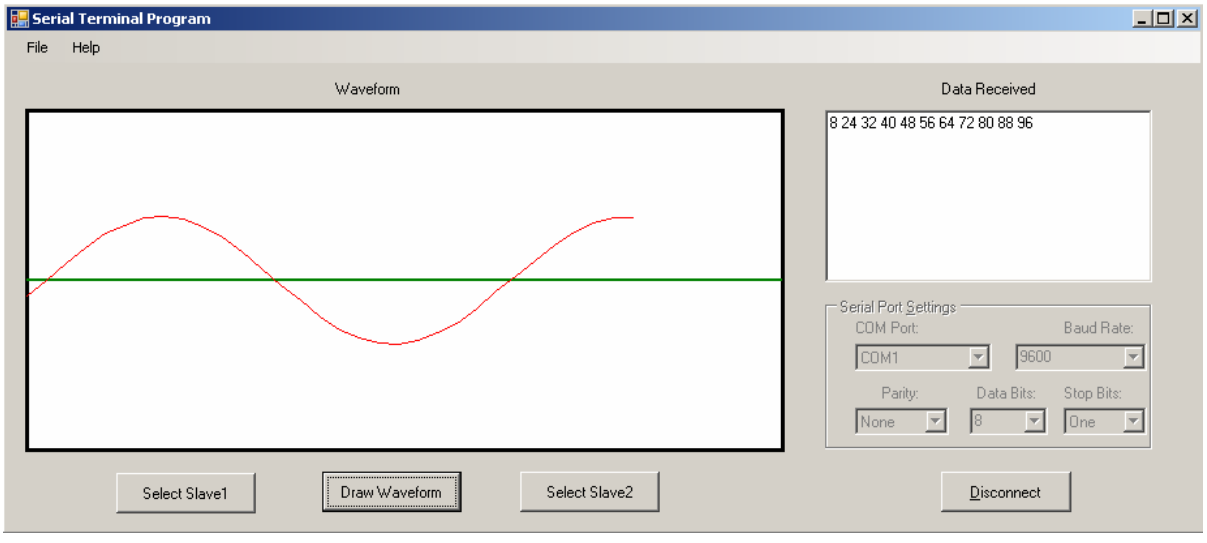


Figure 4.10: Screen shot of the output waveform

CHAPTER 5

SUMMARY/FUTURE WORK

Summary

An I^2C interface (described using Verilog®) has been developed and fully tested on inexpensive, readily available Xilinx FPGA development boards. The I^2C interface was tested at an operating of frequency of 100 KBits/sec. Further testing and additional work is necessary in order to achieve the desired operating frequency of 10 MBits/sec.

The present system operates in 7-bit address mode and has the ability to transfer up to 2^{29} bytes per session. The I^2C Interface not only simulates successfully but operates correctly when implemented on a Xilinx FPGA development board. The prototype system described in Chapter 4 demonstrates that the I^2C interface circuits described in this thesis should meet the requirements of the proposed application described in Chapter 1 of this thesis.

Future Work

The current PSD8C IC provide for analog outputs. Storing data in a digital format on-chip before transmittal to a host computer over the I^2C Interface will result in an improved system performance since the transmission of digital data is much less susceptible to interference form environmental noise sources. As described in Chapter 1 a full system requires the use of many (a few dozen) PSD8C chips.

The system is arranged in the following way. At present, two PSD8C chips are mounted on a chip board along with cable driver circuits that allow the analog outputs of the PSD8C chip to drive long cables which connect to a VME-based

ADC. Up to 16 chip boards can then be plugged into a motherboard. In the near future, we plan to modify the PSD8C chip-board. The current plan is to remove the cable drivers from the chip board and replace them with an ADC (similar to or perhaps even the same ADC which is on the Xilinx development board used in this project) and an FPGA which would be used to implement the I^2C Slave interface described in this thesis. It will also be necessary to modify the FPGA code to control the ADC and to interact with the PSD8C readout electronics.

Finally, in the distant future the I^2C Slave may be combined with a two-stage flash ADC (currently under development) and a 32 location by 8 bit static RAM and integrated onto a second generation PSD8C chip. Since the I^2C Slave was described using a hardware description language, porting the design to an IC is relatively straightforward. One only needs to re-target the design for a standard cell library rather than for the Xilinx FPGA.

REFERENCES

[Xil:07] Xilinx, “Spartan-3E Libraries Guide for HDL Designs”, (1995-2007).

[Xil:08] Xilinx, “Spartan-3E FPGA Starter Kit Board User Guide”, (June 20, 2008).

[Jes:06] Rod Jesman, Fernando, Martinez Vallina and Jafar Saniie, “MicroBlaze Tutorial: Creating a Simple Embedded System and Adding Custom Peripherals Using Xilinx EDK Software Tools”, 2006.

[Phi:00] Phillips Semiconductors, “THE I^2C-BUS SPECIFICATION”, Version 2.1, (January 2000).

[The:06] Theresa Chou, “MicroBlaze Overview”, (2006).

[Das:08] D. Dasari, “Design of On-chip ADC for Custom ASICs Used in the Detec-tion of Ionizing Radiation,” SIUE EE M.S. Thesis, (2008).

[Eng:07a] G. Engel, M. Sadasivam, M. Nethi, J. M. Elson, L. G. Sobotka and R. J. Charity, “Multi-Channel Integrated Circuit for Use in Low and Intermediate Energy Nuclear Physics - HINP16C” in Nucl. Instru. Meth. A573, 418-426, (2007).

[Eng:07b] G. Engel, NSF-MRI proposal. This can be found at the PI’s WEB site.

[Pro:07] J. Proctor , “Design of a Multi-Channel Integrated Circuit for Use in Nuclear Physics Experiments Where Particle Identification is Required,” SIUE EE M.S. Thesis, (2007).

[Sad:02] M. Sadasivam , “A multi-channel integrated circuit for use with silicon strip detectors in experiments in low and intermediate physics,” SIUE EE M.S. Thesis, (2002).

[Gan:00] M. Ganesan, “CMOS low-noise IC for capacitive detector,” SIUE EE M.S.Thesis, (2000).

APPENDIX A

Verilog Code for I2C Slave

I2C Slave Controller:

```
Module Prototype(bit8_1load1,I2Crst,IO9OUT,SPI_MISO,SPI_MOSI,SPI_SS_B,DAC_CS,SF_CE0
,FPGA_INIT_B,AMP_SHDN,AD_CONV,SPI_SCK,AMP_CS,test1,ext_data_out,INT, Tx_nRx,
I2Caddress,ext_data_in,sdaout,sclout,sdatest,clk,rst,WR,RD,sdain,sclin);
        input I2Crst,clk,rst,WR,RD,sdain,sclin,bit8_1load1;
        input  [1:0] I2Caddress;
        input  [7:0]  ext_data_in;
        output [7:0]  ext_data_out;
        output INT, Tx_nRx, sdaout,sclout,sdatest;
        output [31:0] test1;
        wire   [31:0]  test;
        input      SPI_MISO;
        output     SPI_MOSI;
        output       SPI_SS_B;
        output       DAC_CS;
        output       SF_CE0;
        output       FPGA_INIT_B;
        output       AMP_SHDN;
        output       AD_CONV ;
        output       SPI_SCK;
        output       AMP_CS ;
        wire        IO9IN;
        output       IO9OUT;
        wire       BUS_CLK;
        wire       CONV;
        wire       AMP;
        wire        DOUTin;
        wire       result_load,DOUTout;
        wire   [13:0]  ADC14bits;
        reg    [13:0]  ADC14bits1;
        reg    [13:0]  ADC14bits2 [0:31];
        wire   [7:0]   I2Cin;
        wire   [4:0]   tempaddr;
        reg            [4:0]   address;
        reg            RamRnW,cnt,cnt1;
        wire           bit8_1load;
        wire   [1:0]   cnt3,cnt2;
        wire   [17:0]  Data,Data1,Ram_out;

 wire   Bus2IP_Reset;                  // Bus to IP resetv
wire      [0 :31]           Bus2IP_Data;
wire      [0 : 3]         Bus2IP_BE;
wire      [0 : 1]              Bus2IP_RdCE;
wire      [0 : 1]              Bus2IP_WrCE;
wire     [0 : 31]           IP2Bus_Data;
  wire  IP2Bus_RdAck;                  // IP to Bus read transfer acknowledgement
  wire  IP2Bus_WrAck;                  // IP to Bus write transfer acknowledgement
  wire   bit8,P,bit8_2,bit8_1,IP2Bus_Error;
  reg          ADCen;
  wire  bit9_clear,bit9,EPSD_clk,EPSD_clk1;
  integer i;

//      assign EPSD_clk = (rst&clk)^((AD_CONV)^(bit8_1load));
//  assign EPSD_clk1 = ~EPSD_clk;

// RAM32X1S: 32 x 1 posedge write distributed (LUT) RAM
// All FPGA
// Xilinx HDL Libraries Guide, version 9.1i

        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst0 (
```

```
        .O(Ram_out[0]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(ADC14bits[0]), // RAM data input
        .WCLK(AD_CONV), // Write clock input
        .WE(result_load) // Write enable input
        );
// End of RAM32X1S_inst instantiation
        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst1 (
        .O(Ram_out[1]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(ADC14bits[1]), // RAM data input
        .WCLK(AD_CONV), // Write clock input
        .WE(result_load) // Write enable input
        );
        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst2 (
        .O(Ram_out[2]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(ADC14bits[2]), // RAM data input
        .WCLK(AD_CONV), // Write clock input
        .WE(result_load) // Write enable input
        );
        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst3 (
        .O(Ram_out[3]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(ADC14bits[3]), // RAM data input
        .WCLK(AD_CONV), // Write clock input
        .WE(result_load) // Write enable input
        );
        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst4 (
        .O(Ram_out[4]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(ADC14bits[4]), // RAM data input
        .WCLK(AD_CONV), // Write clock input
        .WE(result_load) // Write enable input
        );
        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst5 (
        .O(Ram_out[5]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(ADC14bits[5]), // RAM data input
```

```
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst6 (
.O(Ram_out[6]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[6]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst7 (
.O(Ram_out[7]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[7]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst8 (
.O(Ram_out[8]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[8]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst9 (
.O(Ram_out[9]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[9]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst10 (
.O(Ram_out[10]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[10]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst11 (
.O(Ram_out[11]), // RAM output
.A0(address[0]), // RAM address[0] input
```

```
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[11]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst12 (
.O(Ram_out[12]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[12]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst13 (
.O(Ram_out[13]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(ADC14bits[13]), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst14 (
.O(Ram_out[14]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(1'b0), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst15 (
.O(Ram_out[15]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(1'b0), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
RAM32X1S #(
.INIT(32'h00000000) // Initial contents of RAM
) RAM32X1S_inst16 (
.O(Ram_out[16]), // RAM output
.A0(address[0]), // RAM address[0] input
.A1(address[1]), // RAM address[1] input
.A2(address[2]), // RAM address[2] input
.A3(address[3]), // RAM address[3] input
.A4(address[4]), // RAM address[4] input
.D(1'b0), // RAM data input
.WCLK(AD_CONV), // Write clock input
.WE(result_load) // Write enable input
);
```

```
        RAM32X1S #(
        .INIT(32'h00000000) // Initial contents of RAM
        ) RAM32X1S_inst17 (
        .O(Ram_out[17]), // RAM output
        .A0(address[0]), // RAM address[0] input
        .A1(address[1]), // RAM address[1] input
        .A2(address[2]), // RAM address[2] input
        .A3(address[3]), // RAM address[3] input
        .A4(address[4]), // RAM address[4] input
        .D(1'b0), // RAM data input
        .WCLK(AD_CONV), // Write clock input
        .WE(result_load) // Write enable input
        );
        mux2to1_1bit   muxclk(EPSD_clk,rst,(~AD_CONV)^bit8_1load,clk);//change address at
negedge of AD_CONV while write at pos
        I2C     Slave(
P,bit8,bit9,bit8_1,test,sdatest,sclin,sdain,sclout,sdaout,ext_data_out,ext_data_in,I2Cin,WR,R
D,I2Caddress,rst|ADCen,INT,Tx_nRx,clk);
        adc
        adconverter(ADC14bits,result_load,rst|~ADCen,SPI_MISO,SPI_MOSI,SPI_SS_B,DAC_CS,SF_CE0,
FPGA_INIT_B,AMP_SHDN,AD_CONV,SPI_SCK,AMP_CS,IO9IN,IO9OUT,BUS_CLK,FPGA_CLK,CONV,AMP,DOUTin,DOU
Tout,clk,Bus2IP_Reset,Bus2IP_Data,Bus2IP_BE,Bus2IP_RdCE,Bus2IP_WrCE,IP2Bus_Data,IP2Bus_RdAck,
IP2Bus_WrAck,IP2Bus_Error);

//      assign ADC14bits = ADC14bits1;
//      RAM32x18              buffer(address, Data, RamRnW);
//      mux2to1_18bit_inout   mem_interface(Data,RamRnW,{4'h0,ADC14bits1},Ram_out);
        //mux2to1_18bit_inout (io, RamRnW, in, out);
        mux4to1_8bit
        mux2(I2Cin,cnt2,8'h00,{1'b0,address,Ram_out[17:16]},Ram_out[15:8],Ram_out[7:0]);
//      Select each byte from the same RAM cell to load to I2C. send the number of bytes
(address)
        buf(bit8_2,bit8_1);
        Counter_2bit   Counter(cnt2,bit8_1load, rst|P, bit8_1, clk);//bit8_1load will be high
after three bit8


        always @(posedge EPSD_clk)
                begin
                        if(rst)
                                begin
                                        address = 5'h00;
                                        ADCen <= 1'b1;

                                        ADC14bits1 = 14'h0;
                                        for(i=0;i<=31;i=i+1)
                                                ADC14bits2[i] <= i;
                                end
                        else if((address != 5'h1f)&ADCen&result_load)
                                begin
                                        ADC14bits1 = ADC14bits2[address];


                                        address = address + 1;
                                end
                        else if((address == 5'h1f)&ADCen)
                                begin
                                        ADCen <= 1'b0;

                                end
                        else if(~ADCen)
                                address = address -1;
                end
assign test1[0] = bit8_1load;//a6
assign test1[1] = SPI_MISO;//"B6"
assign test1[2] = EPSD_clk;//"E7"
assign test1[3] = result_load;//"F7"

//RAM32x18 (Address, Data, CS, WE, OE);
endmodule
```

I2C Slave

```
module I2C(
P,bit1,bit9,bit8_2,test,sdatest,sclin,sdain,sclout,sdaout,ext_data_out,ext_data_in,Ram_out,WR
,RD,address,rst,INT,Tx_nRx,fpga_clk);
        input fpga_clk,rst,WR,RD,sdain,sclin;
        input  [1:0] address;
        input  [7:0]  ext_data_in;
        output [7:0]  ext_data_out;
        wire   [7:0]   DR_in ,ADD1_out,data_out,ADD2_out, BB_in, BB_out;
        wire    [4:0]  ADDC_out;
        output  bit1,bit9,P,bit8_2,INT, Tx_nRx, sdaout,sclout,sdatest;
        output [31:0] test;
        wire comp01,comp02,S0, S1, a,sdaout0,DR_clk_out,TxRx,TxRx1;
        wire A1,A2,S,P1;
        wire clktest,SorPL01,SorPL1,bit9_4,bit9_3,bit9_2,bit9_1,bit7,bit8,bit17,bit18,
ACK,ACK1,ACK_in,MACK;
        wire SorPLtest,beginT, SorPL, SorPL0, Counter0;
        wire out1,sout,sout1, soutinv, clkSout, SDA_MUX_SEL, sdaout1,sdaoutg, sdaoutmux;
        input   [7:0]  Ram_out;
        wire    [9:0]  cnt;
        wire    bit8_1,C10_en;
        reg     C10_reset;
        wire buftest,beginT1,beginT2,A1_ld1,TxRx2,TxRx3, TxRx4, MACK_detect_in,
MACK_detect1,clk;
        reg     MACK_detect;

        assign clk = fpga_clk;
        assign sdatest = sdain&~sdaout;

//adc(ADC14bits,result_load,reset,SPI_MISO,SPI_MOSI,SPI_SS_B,DAC_CS,SF_CE0,FPGA_INIT_B,AMP_SH
DN,AD_CONV,SPI_SCK,AMP_CS,IO9IN,IO9OUT,BUS_CLK,FPGA_CLK,CONV,AMP,DOUTin,DOUTout,Bus2IP_Clk,IP
2Bus_Error)

//      adc
adconverter(ADC14bits,result_load,reset,SPI_MISO,SPI_MOSI,SPI_SS_B,DAC_CS,SF_CE0,FPGA_INIT_B,
AMP_SHDN,AD_CONV,SPI_SCK,AMP_CS,IO9IN,IO9OUT,BUS_CLK,FPGA_CLK,CONV,AMP,DOUTin,DOUTout,Bus2IP_
Clk,IP2Bus_Error);

////////////////////////////////// Start and Stop detection:////////////////////////////

        SandP_Detect  SPDET(sdain, sclin, P,rst,clk);
        assign INT=ACK|RD;
        assign sclout=1'b0;

////////////////////////////////// Bus buffer multiplexer. ////////////////////////////

        mux_4to1_8bit BBmux(BB_in, address, data_out,ext_data_out ,ADD1_out, ADD2_out);

////////////////////////////////////////// Data Reg ////////////////////////////////////

        Data_Reg DR(DR_clk_out,sout,sdain, Ram_out,data_out,1'b1,~SorPL,sclin,clk,rst);
//                DR(sout, sin, data_in,data_out,load,SorPL,sclk,clk,rst)


//      buffer: disable buffer if address is not correct
        buffer  buf8x8(buftest,DR_in,data_out,beginT2,TxRx,rst|P,clk,bit8_2);
//      buffer                (out  ,data_in,load    ,RnW  ,rst ,Pclk,clk);

// Serial or Paralell load signal

        assign SorPL = ~bit9_4&TxRx&beginT&~P&bit9;
// beginT=A1&A2: When addresses are equal, start transferring data
// Serial or Paralell load register
        Delay_2Clk delay2(bit9_1,bit9,rst,~clk);
        Delay_2Clk delay3(bit9_2,bit9_1,rst,clk);
        Delay_2Clk delay4(bit9_3,bit9_2,rst,~clk);
        Delay_2Clk delay5(bit9_4,bit9_3,rst,clk);
// delay bit9, make sure there is no extra value for
//      SorPL

        reg_1bit DRsReg(sout1, sout, 1'b1, rst, ~clkSout);
```

```
//delay for half sclin cycle and put the data out on negedge
//that makes the data changed only in when sclin is 0 and data is stable when sclin is high.
       reg_1bit ClkS(clkSout, sclin, 1'b1, rst, clk);//sync clin with the internal clk
//Serial data out multiplexer.(out, sel, in0, in1)
       mux2to1_1bit SDA_MUX(sout2,SDA_MUX_SEL , 1'b1, sout1);
       assign SDA_MUX_SEL = beginT&TxRx ;//TxRx = 1: transfer data; TxRx = 0, receive data
//if address is correct and want to send data --> select sout1 at the output of the mux
//if not, select logic high at the output of the mux.
//     assign sdaoutg= (sdaout1&Counter0&~sdaoutmux);

       mux2to1_1bit SDA_OUT_Mux(sdaoutg,MACK1|MACK_detect|/*(bit9&~sclin)|*/P,sdaout1,1'b0);
//     Define times to release the sda line:
/*     + At negedge of 9th clk, relsease the line if there is no MACK (MACK1 = 1'b1) to
detect stop transfering
       data: MACK1
       + MACK_detect high for the transmitter to release the sdaline at the negedge of 8th
bit, so that the
       receiver can ACK the transmitter at the 9th clk
       + Right after negedge of bit 9th, release the line (because if receiver, then I
already sent the
       ACK at the posedge of 9th bit, if transmitter, I dont want to use sdaline until bit
1??? Correct?
       + Obviously, during stop, I dont want to use the line.
*/
       assign sdaoutmux = SDA_MUX_SEL&(bit8|bit9)&ACK;
//Serial data out register.(out, sel, in0, in1)
//     reg_1bit SDA_OUT_REG(sdaout, sdaoutg, 1'b1, rst, clk);
       Delay_2Clk delay(sdaout0,sdaoutg,rst,clk);
       mux2to1_1bit SdaFinalMux(sdaout,P,sdaout0,1'b0);
// Bus Buffer REG
       Reg_8bit BB(BB_out,BB_in,BB_load, clk, rst);

       assign BB_load = bit8&beginT&~TxRx;
       assign ext_data_out=BB_out;

// ADD REG

//   Reg_8bit Add_reg1(ADD1_out,8'h01,ADD1_load,clk,rst);
   Reg_8bit Add_reg2(ADD2_out,ext_data_in,ADD2_load,clk,rst);
       assign ADD1_out = 8'h02;
//     assign ADD1_load= address[1]&~address[0]&WR;
//     assign ADD2_load= address[1]& address[0]&WR;


// COMPARATOR REG

// reg_1bit Aadd1(A1, comp01,A1_ld, rst|P, clk);
       reg_1bit Aadd2(A2, comp02,A2_ld, rst|P, clk);

/////TxRxTxRxTxRxTxRxTxRxTxRxTxRxTxRx TxRx from here
TxRxTxRxTxRxTxRxTxRxTxRxTxRxTxRx//////////////////////////////////////////


// Transfer Receive REG
       reg_1bit       beginT_delay(beginT1,beginT,1'b1,rst,sclin);
       reg_1bit       beginT_delay1(beginT2,beginT1,1'b1,rst,clk);

       reg_1bit_loadonce TxorRx1(TxRx1, data_out[0],bit8_2, rst|P, clk);
       reg_1bit_n TxorRx3(TxRx3, TxRx1,1'b1, rst, sclin);
       reg_1bit TxorRx(TxRx, TxRx1,1'b1, rst, sclin);
//      reg_1bit_n(out, in,load,clear, clk);
       reg_1bit_n TxorRx2(TxRx2, TxRx,1'b1, rst, sclin);
       reg_1bit TxorRx4(TxRx4, TxRx2,1'b1, rst, sclin);
//     TxRx4 is high only after the first 9 bits of the address, or only high at the begining
//     of 2nd byte
//     Delay TxRx for another cycle
//     Delay TxRx for one cycle, TxRx will be high at the negedge of 9th bit, that makes the
// A1_ld must be long enough to make sure that it will be high when there is a posedge of
sclin
// Detect 8th bit to see if transmit or receive, if TxRx=1, then the server wants to read
from the salve
// otherwise, the server wants to send data to the slave
assign Tx_nRx=TxRx&~P;
```

```
// A1 and A2 load
//      mux2to1_1bit M8(A1_ld, bit8, 1'b0, ~beginT); //if beginT is always low, that means
A1_load is wrong
// A1 load is used to load the comparator reg. It happens only at the first 8th bit of sdain

        mux2to1_1bit M16(A2_ld, bit17, 1'b0, ~beginT);
// Unable to detect changes in direction, once beginT is high, it will be high forever
because A1_ld
// is not high again, A1 and A2 keep the same value forever
// I want to change A2_ld, I want to extend it longer
        //reg_1bit regA1Load(beginT, A1, 1'b1,rst, sclin);
        assign beginT=A1;


////////////////////////// Load A1 here ////////////////////////////////////////

        Delay_2Clk ClkDelay(bit8_1,bit8,rst,clk);
        Delay_2Clk ClkDelay1(bit8_2,bit8_1,rst,~clk);
        Reg_Load_Once  Address1(A1,comp01,1'b1,rst|P,bit8_2,clk);//at first bit8, load, reset
after stop.
//      Reg_Load_Once(out,in,load,rst,sclk,pclk);


// SCL Counter 5-bit
        Counter_5bit Add_counter(ADDC_out, 5'b00001,~P, Add_cou_ld,rst|P/*|(MACK&bit9)*/,
sclin^((rst|P)&clk),clk);
//Counter_5bit(out, in, load, clear, clk);
//use load signal to reset the counter, whenenver reaches bit 9 or bit 18, reset counter. Is
this correct?
//should load when there is an ACK (either Master or Slave ACK)! Should Clear when the stop
signal is high
//clock for the counter is just the sclin.
assign Add_cou_ld= (bit9&beginT&~P);//|(bit1&~beginT);
// Address Comparators
        assign comp01=(~(ADD1_out[0]^data_out[1]|ADD1_out[1]^data_out[2]
                |ADD1_out[2]^data_out[3]|ADD1_out[3]^data_out[4]|ADD1_out[4]^data_out[5]
                |ADD1_out[5]^data_out[6]|ADD1_out[6]^data_out[7]));
        assign
comp02=(~(ADD2_out[0]^data_out[0]|ADD2_out[1]^data_out[1]|ADD2_out[2]^data_out[2]

        |ADD2_out[3]^data_out[3]|ADD2_out[4]^data_out[4]|ADD2_out[5]^data_out[5]
                                        |ADD2_out[6]^data_out[6]|ADD2_out[7]^data_out[7]));
        assign bit1=ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&~ADDC_out[4];
        assign bit7=ADDC_out[0]&ADDC_out[1]&ADDC_out[2]&~ADDC_out[3]&~ADDC_out[4];

        assign bit8=~ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&ADDC_out[3]&~ADDC_out[4];
//assign bit16=~ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&ADDC_out[4];
        assign bit9=ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&ADDC_out[3]&~ADDC_out[4];
   assign bit17=ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&ADDC_out[4];
        assign bit18=~ADDC_out[0]&ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&ADDC_out[4];
        assign Counter0=ADDC_out[0]|ADDC_out[1]|ADDC_out[2]|ADDC_out[3]|ADDC_out[4];
// ACKNOWLEDGE
                                                //(out, sel, in0, in1)
mux2to1_1bit ACK_MUX(sdaout1, ACK, soutinv, 1'b1);
/* Question: When should I send ACK?
        Send ack only in transfering mode and send out at bit 9*/
not (soutinv,sout2);
reg_1bit ACK_reg0(ACK1,ACK_in,1'b1, rst, clk);
reg_1bit_nCLK ACK_reg(clktest,ACK,ACK1,1'b1, rst|bit9_4, sclin/*clkSout*/, clk);//use bit9_4
to reset ACK to make
// sure that the slave will release the sdaline for stop signal
// Load ACK1 to ACK during negedge of bit 8
//reg_1bit_nCLK (out,in,load,clear, sclk,pclk);
// when rst = 0, clk = clkSout ( or sclin afer being synced with clk), load at the negedge of
the sclin
// that guarantees that the sda will not be pulled down when sclin is still high.
assign ACK_in= A1&bit8&~TxRx;//|(A2&bit17);
//if the output of the counter counts to 8 and if address is ok --> ACK? No, should have more
//logic to check for ACK condition. Add busy signal to the input of the I2C, and there should
be
//TxRx there, the slave send ACK only when it is in the receiving status
// Master acnoligment register// Master acnoligment register
//assign MACK_detect = A1&TxRx3&TxRx2;
```

```
//reg_1bit MACK_detect_reg0(MACK_detect1,MACK_detect_in,1'b1, rst, clk);
//reg_1bit_nCLK MACK_detect_reg(MACK_detect,MACK_detect1,1'b1, rst, sclin/*clkSout*/, clk);

//      MACK_detect high for the transmitter to release the sdaline at the negedge of 8th bit,
so that the
//      receiver can ACK the transmitter at the 9th clk
always @(negedge sclin)
        if(rst)
                MACK_detect <= 1'b0;
        else if(bit8&TxRx2)//that means there is no MACK_detect at the start byte
                MACK_detect <= 1'b1;
        else
                MACK_detect <= 1'b0;

//reg_1bit MACK_reg(MACK,sdain,/*SorPL0&sclin*/~P&TxRx4&bit9&sclin, rst|P, clk);
        reg_1bit_loadonce MACK_reg(MACK,sdain,~P&TxRx4&bit9_4,rst|P|bit7, clk);
//      reg_1bit_loadonce                               (out,in,load,rst,sclk,pclk);mack is
loaded EVERY bit9_4, so, use bit 7 to reset
// dont use bit one, because of extra sclin posedge and negedge from the master, it makes
extra bit 1 :D
reg_1bit_n      MACK_reg1(MACK1,MACK,1'b1,rst,sclin);
// Delay MACK for hafl clk cycle, for a transmitter stop transfering data detection
// Detect master ACK. For simulation purpose, input should be "sdain&~sdaout"
assign test[0] = sout;//a6
assign test[1] = TxRx;//"B6"
assign test[2] = bit8_1;//"E7"
assign test[3] = TxRx;//"F7"
assign test[4] = bit8;//a10
assign test[5] = SorPL0;//a11
assign test[6] = MACK1;//a12
assign test[7] = DR_clk_out;//a13
assign test[8] = sout;//a14
assign test[9] = sdaout;//a15
assign test[10] = TxRx;//a16
assign test[11] = P;//a17
assign test[12] = sclin^((rst|P)&clk);//a18
assign test[13] = data_out[0];
assign test[14] = data_out[1];
assign test[15] = data_out[2];
assign test[16] = data_out[3];
assign test[17] = data_out[4];
assign test[18] = data_out[5];
assign test[19] = data_out[6];
assign test[20] = DR_clk_out;
endmodule
```

ADC Converter:

```
module adc
(
  // -- ADD USER PORTS BELOW THIS LINE ---------------

         ADC14bits,
                        result_load,
                        reset,
         SPI_MISO,
                        SPI_MOSI,
         SPI_SS_B,
         DAC_CS,
         SF_CE0,
         FPGA_INIT_B,
         AMP_SHDN,
         AD_CONV,
         SPI_SCK,
         AMP_CS,
                        IO9IN,
                        IO9OUT,
                        BUS_CLK,
                        FPGA_CLK,
                        CONV,
                        AMP,
                        DOUTin,
```

```
                    DOUTout,

  // --USER ports added here
  // -- ADD USER PORTS ABOVE THIS LINE ---------------

  // -- DO NOT EDIT BELOW THIS LINE ------------------
  // -- Bus protocol ports, do not add to or delete
  Bus2IP_Clk,                     // Bus to IP clock
  Bus2IP_Reset,                   // Bus to IP resetv
  Bus2IP_Data,                    // Bus to IP data bus
  Bus2IP_BE,                      // Bus to IP byte enables
  Bus2IP_RdCE,                    // Bus to IP read chip enable
  Bus2IP_WrCE,                    // Bus to IP write chip enable
  IP2Bus_Data,                    // IP to Bus data bus
  IP2Bus_RdAck,                   // IP to Bus read transfer acknowledgement
  IP2Bus_WrAck,                   // IP to Bus write transfer acknowledgement
  IP2Bus_Error                    // IP to Bus error response
  // -- DO NOT EDIT ABOVE THIS LINE ------------------
); // user_logic// -- Bus protocol parameters, do not add to or delete

parameter C_SLV_DWIDTH                   = 32;
parameter C_NUM_REG                      = 2;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

// -- ADD USER PORTS BELOW THIS LINE -----------------
parameter   s0  = 0;
parameter   s1  = 1;
parameter   s2  = 2;
parameter   s3  = 3;
parameter   s4  = 4;

input      SPI_MISO,reset;
output     SPI_MOSI;
output       SPI_SS_B;
output       DAC_CS;
output       SF_CE0;
output       FPGA_INIT_B;
output       AMP_SHDN;
output       AD_CONV ;
output       SPI_SCK;
output       AMP_CS ;
input        IO9IN;
output       IO9OUT;
output       BUS_CLK;
output       FPGA_CLK;
output       CONV;
output       AMP;
input        DOUTin;
output       DOUTout;


// --USER ports added here
// -- ADD USER PORTS ABOVE THIS LINE -----------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol ports, do not add to or delete
input                                     Bus2IP_Clk;
input                                     Bus2IP_Reset;
input      [0 : C_SLV_DWIDTH-1]           Bus2IP_Data;
input      [0 : C_SLV_DWIDTH/8-1]         Bus2IP_BE;
input      [0 : C_NUM_REG-1]              Bus2IP_RdCE;
input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
output                                    IP2Bus_RdAck;
output                                    IP2Bus_WrAck;
output                                    IP2Bus_Error;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

//----------------------------------------------------------------------------
// Implementation
//----------------------------------------------------------------------------

  // --USER nets declarations added here, as needed for user logic
```

```
  // Nets for user logic slave model s/w accessible register example

  wire [0:8] SPI_MOSII;
  wire       SPI_SCK;
  reg       AMP_CS,enable,AD_CONV=0;
  reg [0:2] state ;
  reg [0:7] cnt1 ;
  integer      cnt;
  wire      clk4fpga;
  reg     SPI_MOSI;
  wire [7:0] cnt2;
  wire       SPI_SS_B=1;
  wire       DAC_CS=1;
  wire       SF_CE0=1;
  wire       FPGA_INIT_B=1;
  wire       AMP_SHDN=0;
  wire       enable1;
  reg        enable2=0;
  wire en;
       reg     load_en;
       reg     [13:0]  ADCresult;
       output  [13:0]  ADC14bits;
       output                          result_load;

    reg        [0 : C_SLV_DWIDTH-1]           slv_reg0;
    reg        [0 : C_SLV_DWIDTH-1]           slv_reg1;
    wire       [0 : 1]                        slv_reg_write_sel;
    wire       [0 : 1]                        slv_reg_read_sel;
    reg        [0 : C_SLV_DWIDTH-1]           slv_ip2bus_data;
    wire                                      slv_read_ack;
    wire                                      slv_write_ack;
    integer                                   byte_index, bit_index;


  // --USER logic implementation added here

  // ------------------------------------------------------

  assign
    slv_reg_write_sel = Bus2IP_WrCE[0:1],
    slv_reg_read_sel  = Bus2IP_RdCE[0:1],
    slv_write_ack     = Bus2IP_WrCE[0] || Bus2IP_WrCE[1],
    slv_read_ack      = Bus2IP_RdCE[0] || Bus2IP_RdCE[1];


 // implement slave model register(s)
  /*always @( posedge Bus2IP_Clk )
    begin: SLAVE_REG_WRITE_PROC

      if ( Bus2IP_Reset == 1 )
        begin
          slv_reg0 <= 0;
          slv_reg1 <= 0;
        end
      else
        case ( slv_reg_write_sel )
          2'b10 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg0[bit_index] <= Bus2IP_Data[bit_index];
          2'b01 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
          default : ;
        endcase
```

```
    end // SLAVE_REG_WRITE_PROC*/

  // implement slave model register read mux
  always @( slv_reg_read_sel or slv_reg0 or slv_reg1 )
    begin: SLAVE_REG_READ_PROC

      case ( slv_reg_read_sel )
        2'b10 : slv_ip2bus_data <= slv_reg0;
        2'b01 : slv_ip2bus_data <= slv_reg1;
        default : slv_ip2bus_data <= 0;
      endcase

    end // SLAVE_REG_READ_PROC

////////////////////////////////////////// I2C components
////////////////////////////////////
       Counter2       C2(en,AD_CONV,reset,Bus2IP_Clk);
//     Counter2(en,AD_CONV,rst,clk)
       Counter34 Counter(cnt2,8'h00,1'b0,~en|AD_CONV,~clk4fpga);
//     Counter34(out,in,load,rst,clk);
       always @(negedge Bus2IP_Clk)
              begin
                     if(reset)
                            begin
                            load_en <= 1'b0;
                            ADCresult <= 14'h0000;
                            end
                     case(cnt2)
                     8'h05: load_en <= 1'b0;
                     8'h13: ADCresult[13] <= SPI_MISO;
                     8'h14: ADCresult[12] <= SPI_MISO;
                     8'h15: ADCresult[11] <= SPI_MISO;
                     8'h16: ADCresult[10] <= SPI_MISO;
                     8'h17: ADCresult[9] <= SPI_MISO;
                     8'h18: ADCresult[8] <= SPI_MISO;
                     8'h19: ADCresult[7] <= SPI_MISO;
                     8'h1a: ADCresult[6] <= SPI_MISO;
                     8'h1b: ADCresult[5] <= SPI_MISO;
                     8'h1c: ADCresult[4] <= SPI_MISO;
                     8'h1d: ADCresult[3] <= SPI_MISO;
                     8'h1e: ADCresult[2] <= SPI_MISO;
                     8'h1f: ADCresult[1] <= SPI_MISO;
                     8'h20: ADCresult[0] <= SPI_MISO;
                     8'h21: load_en <= 1'b1;
                     default: ADCresult <= ADCresult;
                     endcase
              end
       assign  ADC14bits = ADCresult;
       assign  result_load = load_en;

//////////////////////////////////////  End of I2C components
////////////////////////////////////

   clk4fpga    fpg(clk4fpga,Bus2IP_Clk);
   always @(posedge clk4fpga)
   begin
     if(reset)
         begin
              state  <= s0;
         end
    else
    begin
        case(state)
          s0:
          begin

              state <= s1;
              cnt    =8;
        end
        s1:
                      if(cnt==8)
```

```
                              begin
            AMP_CS <=1'b1;
                              state <= s2;
                              end

        s2:
        begin

               if (cnt != -1)
                              begin
                              cnt  = cnt - 1;
                              AMP_CS <=1'b0;
               SPI_MOSI <= SPI_MOSII[cnt];

               end
               else
                              begin
                              state <= s3;
                              AMP_CS <=1'b1;
                              end
               //SPI_MOSI <= SPI_MOSII[cnt];

        end

        s3 :
        begin

              enable <=1'b1;
                        state  <= s4;
              cnt1    <=8'h00;
                        AD_CONV <=1'b1;
        end


                s4:
                      begin
                       AD_CONV <=1'b0;
                      cnt1  <= cnt1 + 8'h01;
                      if(cnt1 ==8'h22)
                        begin
                        AD_CONV <=1'b1;
                        cnt1    <=8'h00;
                        end
                      end
         endcase
    end
    end
     assign     SPI_SCK  =  clk4fpga & ~AD_CONV;
     assign     SPI_MOSII = 9'b100010000;
          assign     IO9OUT   = SPI_MISO;
          assign     BUS_CLK  = DOUTin;
     assign     FPGA_CLK = SPI_SCK;
     assign        CONV     = AD_CONV;
          assign     AMP      = clk4fpga;
          assign     DOUTout  = DOUTin;

  // ------------------------------------------------------------
  // Example code to drive IP to Bus signals
  // ------------------------------------------------------------

  assign IP2Bus_Data    = slv_ip2bus_data;
  assign IP2Bus_WrAck   = slv_write_ack;
  assign IP2Bus_RdAck   = slv_read_ack;
  assign IP2Bus_Error   = 0;
endmodule
```

Start/Stop Detector:

```
`timescale 1ns / 1ps
```

```
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:    11:09:17 01/22/2008
// Design Name:
// Module Name:    SandP_Control
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module SandP_Detect(sdain, sclin, P,rst,clk);
    input sdain, sclin,rst,clk;
    output P;
         wire S0,S1,a,rstclk;//, a,S0,S1;
         reg Q;
         wire  rst2,rst1;
        assign rstclk=sdain^(rst&clk);

        reg_1bit       r1(rst1,rst,1'b1,1'b0,clk);
        reg_1bit       r2(rst2,rst1,1'b1,1'b0,~clk);
        toggle  T(a,sclin,rst2,rstclk);
        reg_1bit_n start0(S0, a ,sclin, rst2 , rstclk);
        reg_1bit_p stop0(S1, a ,sclin, rst2 , rstclk);
        assign P=~(S0^S1);
endmodule
```

Toggle Register:

```
module toggle(out,load,clear, clk);
        input                  load;
        input       clear;   /* reset input */
        input       clk;    /* clock input */
        output         out;    /* output of register */
        reg       Q;      /* one bit register to hold count */
        /* State changes only on negative clock transitions */
        always @(posedge clk)
        begin
                if(clear == 1'b1) /*1  */
                        Q <= 1'b1;
                else if(load == 1'b1) /*  1*/
                        Q <= ~Q;
                else
                        Q <= Q;
        end
        assign out = Q;
endmodule
```

Data Register (or Shift Register):

```
module Data_Reg(DR_clk_out,sout,sin,data_in,data_out,load,SorPL,sclk,clk,rst);
//      Data_Reg DR(DR_clk_out,sout,sdain, 8'b00000001,data_out,1'b1,~SorPL,sclin,clk,rst);
input  [7:0]  data_in;
output [7:0]  data_out;
input         clk, sclk, sin,rst,load,SorPL;
output                  DR_clk_out,sout;
reg  [7:0] R;
wire rst1,rst2,sdatain, dclk;
wire [7:0] mux_out;
assign DR_clk_out = R[0];
mux2to1_1bit Clock(dclk, SorPL,clk, sclk);
//      reg_1bit       D1(rst2,rst,1'b1,1'b0,clk);
//      reg_1bit D2(rst1,rst2,1'b1,1'b0,~clk);
```

```
always@(posedge dclk) begin
    if(rst)begin R<=8'h00;end
    else if(load)begin
     R<= mux_out;
      end
    else   R<=R;
  end
assign data_out = R;
buf (sdatain,sin);
mux2to1_1bit Mout(sout, SorPL,1'b0, R[7]);
mux2to1_1bit M0(mux_out[0], SorPL, data_in[0], sdatain);
mux2to1_1bit M1(mux_out[1], SorPL, data_in[1], R[0]);
mux2to1_1bit M2(mux_out[2], SorPL, data_in[2], R[1]);
mux2to1_1bit M3(mux_out[3], SorPL, data_in[3], R[2]);
mux2to1_1bit M4(mux_out[4], SorPL, data_in[4], R[3]);
mux2to1_1bit M5(mux_out[5], SorPL, data_in[5], R[4]);
mux2to1_1bit M6(mux_out[6], SorPL, data_in[6], R[5]);
mux2to1_1bit M7(mux_out[7], SorPL, data_in[7], R[6]);
endmodule
```

Delay Module:

```
module Delay_2Clk(out, in, rst, clk);

output  out;
input   in,rst,clk;
reg     [1:0]cnt;
reg     tmp;
always @(posedge clk)
begin
       if(rst == 1'b1)
              begin
              cnt <= 2'b00;
              tmp <= 1'b0;
              end
       else if(cnt == 2'b10)
              begin
              tmp <= in;
              cnt <= 2'b00;
              end
       else
              cnt <= cnt + 1;
end

assign out = tmp;
endmodule
```

1-bit Register:

```
module reg_1bit(out, in,load,clear, clk);
       input          load;
      input       in;     /* set input */
       input       clear;   /* reset input */
       input       clk;    /* clock input */
       output         out;    /* output of register */
       reg       Q;      /* one bit register to hold count */
       /* State changes only on negative clock transitions */
       always @(posedge clk)
       begin
              if(clear == 1'b1) /*1  */
                     Q <= 1'b0;
              else if(load == 1'b1) /*  1*/
                     Q <= in;
              else
                     Q <= Q;
       end
       assign out = Q;
endmodule
```

Mux 2to1_1 bit:

```
module mux2to1_1bit (out, sel, in0, in1);
       input          sel; /* mux select line */
       input          in0, in1; /* inputs */
       output  out; /* output of mux   */
       reg            out; /* this should optimize out */
       /* if any of the inputs to the mux change we want
          to reevaluate the case statement              */
       always @( sel or in0 or in1 )
              case (sel)
                  1'b0 :            out <= in0;
                  1'b1 :            out <= in1;
                      /* having a default case ensures
                         that the mux will synthesize
                         as purely combinational logic */
                  default :  out <= in0;
              endcase
endmodule
```

8-bit Register:

```
module Reg_8bit(out,in,load,clk,rst);
output         [7:0]  out;
input   [7:0]  in;
input          clk, load, rst;
reg    [7:0] R;
always @(posedge clk)
              if (rst==1'b1) R<=0;
              else if(load==1'b1) R<= in;
              else   R<=R;
assign out = R;
endmodule
```

One-bit negative-edge Register:

```
module reg_1bit_n(out, in,load,clear, clk);
       input          load;
      input       in;     /* set input */
       input       clear;   /* reset input */
       input       clk;    /* clock input */
       output         out;    /* output of register */
       reg       Q;      /* one bit register to hold count */
       /* State changes only on negative clock transitions */
       always @(negedge clk)
       begin
              if(clear == 1'b1) /*  */
                     Q <= 1'b0;
              else if(load == 1'b1) /* */
                     Q <= in;
              else
                     Q <= Q;
       end
       assign out = Q;
endmodule
```

Load-Once Register:

```
module  Reg_Load_Once(out,in,load,rst,sclk,pclk);
//     Reg_Load_Once  Address1(A1,comp01,1'b1,rst|P,bit8_2,clk);
output         out;
input                 in;
input          load,rst,sclk,pclk;
wire           rstclk,rst1,rst2;
```

```
reg                     Q;
reg             cnt;
        assign rstclk = sclk^(rst&pclk);
        reg_1bit        D1(rst2,rst,1'b1,1'b0,pclk);
        reg_1bit D2(rst1,rst2,1'b1,1'b0,~pclk);

        always @(posedge rstclk)
                begin
                        if(rst1)
                                begin
                                        Q <= 1'b0;
                                        cnt <= 1'b0;
                                end
                        else if(cnt == 1'b1)
                                Q <= Q;
                        else if(load)//beginT2 is only high at posedge of bit9
                                begin
                                        Q <= in;
                                        cnt <= cnt + 1;
                                end
                        else
                                Q <= Q;

                end
assign out = Q;
endmodule
```

5-bit Counter:

```
module Counter_5bit(out, in,en, load, clear, clk, fpga_clk);
//      Add_counter(ADDC_out, 5'b00001,~P, Add_cou_ld,P/*|(MACK&bit9)*/, sclin^((rst|P)&clk));
           input   [4:0]   in;
                input  load,en;
                input clear;
                input  fpga_clk,clk;
                output  [4:0]   out;
                reg    [4:0]   Q;
        reg clear1;
                wire    rst1,rst2;
                // Delay P until sclin goes low so that after P going low 1 clk cycle,
                // the counter start counting up if there is a sclin
        reg_1bit        D1(rst2,clear,1'b1,1'b0,fpga_clk);
        reg_1bit D2(rst1,rst2,1'b1,1'b0,~fpga_clk);
                always @(posedge clk)
                begin
                if(rst1 == 1'b1)                Q <= 5'b00000;
      else if (load == 1'b1) Q <= in;
                /*else if(en)
                                                Q <= Q  + 1'b1 ;                */
                else
                        Q <= Q + 1'b1;
                end
                assign out = Q;
        endmodule
```

Four-to-One eight-bit Multiplexer:

```
module mux4to1_8bit (out, sel, in0, in1,in2,in3);
        input           [1:0]   sel; /* mux select line */
        input    [7:0]  in0, in1,in2,in3; /* inputs */
        output  [7:0] out; /* output of mux   */
        reg     [7:0] out; /* this should optimize out */
        /* if any of the inputs to the mux change we want
           to reevaluate the case statement             */
        always @( sel or in0 or in1 )
                case (sel)
                    2'b00 :             out <= in0;
                    2'b01 :             out <= in1;
```

```
                     2'b10 :              out <= in2;
                     2'b11 :              out <= in3;
                    /* having a default case ensures
                       that the mux will synthesize
                       as purely combinational logic */
                default :  out <= in0;
            endcase
endmodule
```

Counter2:

```
module Counter2(en,AD_CONV,rst,clk);
        output  en;
        input   AD_CONV,rst,clk;
        wire    rst2,rst1,rstclk;
        reg     [7:0]   Q;
        reg     temp;

        assign rstclk = AD_CONV^(rst&clk);
        reg_1bit       d1(rst1,rst,1'b1,1'b0,clk);
        reg_1bit       d2(rst2,rst1,1'b1,1'b0,~clk);
        always @(posedge rstclk)
                begin
                        if(rst2)
                                begin
                                        Q <= 8'h00;
                                        temp <= 1'b0;
                                end
                        else if(Q == 8'h01)
                                begin
                                        Q <= Q;
                                        temp <= 1'b1;
                                end
                        else
                                Q <= Q + 1;
                end
        assign en = temp;
endmodule
```

Counter34:

```
module Counter34(out,in,load,rst,clk);
        output  [7:0]   out;
        input   load,rst,clk;
        input   [7:0]   in;
        reg     [7:0]   Q;

        always @(posedge clk)
                begin
                        if(rst)
                                Q <= 8'h00;
                        else if(load)
                                Q <= in;
                        else
                                Q <= Q + 1;
                end
        assign out = Q;
endmodule
```

Clock for FPGA:

```
module clk4fpga(clk4fpga,Bus2IP_Clk);

    input          Bus2IP_Clk;
    output         clk4fpga;
```

```
    reg           enable=1'b1,state= 1'b0;
    reg   [3:0]   clk_cnt= 4'h0 ;
        reg           clk4fpga=1'b0;


    parameter s0  = 0;
    parameter s1  = 1;



      always @ (posedge Bus2IP_Clk)
      begin

               clk_cnt <= clk_cnt + 1;

     case(state)
     s0:
         begin
       clk4fpga <= 1'b0;
        if(clk_cnt == 4'h2)
         begin
          state<= s1;
          clk_cnt <=4'h0;
         end
     end

         s1:
         begin
       clk4fpga <= 1'b1;
        if(clk_cnt == 4'h2)
         begin
          state<= s0;
          clk_cnt <=4'h0;
         end
               end
               /*default:
                begin
           //clk_cnt <= 4'h0;
                       //state <= s0;
       end*/
       endcase
   end
 endmodule
```

2-bit Counter:

```
module Counter_2bit(cnt,out, rst, sclk, pclk);
//     Counter(cnt2,bit8_1load, rst, bit8_2, clk);
               input rst;
               input  pclk,sclk;
               output   out;
               output  [1:0]  cnt;
               reg    [1:0]   Q;
               reg    outp;
               wire                          rst1,rst2,clk1;
       mux2to1_1bit   muxclk(clk1,rst,sclk,pclk);
       reg_1bit       D1(rst2,rst,1'b1,1'b0,pclk);
       reg_1bit D2(rst1,rst2,1'b1,1'b0,~pclk);
       always @(posedge clk1)
       begin
               if(rst1 == 1'b1)
                       begin
                               Q <= 2'b00;
                               outp <= 1'b0;
                       end
//             else if(Q == 2'b00)
//                     begin
//                             outp <= 1'b1;
//                             Q <= Q+1;
//                     end
```

```
        else if(Q == 2'b11)
                begin
                        outp <= 1'b1;
                        Q <= 2'b01;
                end
        else if(Q==2'b01)
                begin
                        outp <= 1'b0;
                        Q <= Q+1;
                end
        else
                Q <= Q + 1;

end
        assign out = outp;
        assign cnt = Q;
endmodule
```

APPENDIX B

Verilog Code for I2C Master

I2C Master:

```
module I2C_Master(data_out,beginT2,TxRx,bit8_1,cs,WR,address,resend,P,NoOfLoc
,NoOfBytes_ready,No82
,No83,No84,No85,test,sdatest,sclin,sdain,sclout,sdaout,ext_data_out,Start_Byte,ext_data_in
,rst,INT,Tx_nRx,fpga_clk);
        input NoOfBytes_ready,cs,WR,resend,fpga_clk,rst,sdain,sclin;
        input   [1:0]                   address;
        input           [31:0]  No82,No83,No84,No85;
        input   [7:0]   Start_Byte,ext_data_in;
        output [7:0]   data_out,ext_data_out;
        wire    [7:0]    DR_in ,ADD1_out,ADD2_out, BB_in, BB_out;
        wire     [4:0]  ADDC_out;
        output bit8_1,TxRx,beginT2,P,INT, Tx_nRx, sdaout,sclout,sdatest;
        output [31:0] test;
        wire STOP,ADD1_load,No_Ack,DR_clk_out,TxRx1;
        wire A1,A2,S,P1;
        wire bit8_4,bit8_3,bit8_2,bit9_4,bit9_3,bit9_2,bit9_1,bit1,bit7,bit8,bit9,bit17,bit18,
             P, ACK,ACK1,ACK_in,MACK;
        wire beginT, SorPL, SorPL0, Counter0;
        wire rstclk0,rstclk14,sclout1,sout,sout1, soutinv, clkSout, SDA_MUX_SEL,
sdaout1,sdaout2,sdaoutg, sdaoutmux;
        wire    No82_1,No83_1,No84_1,No85_1;
        wire    [31:0]  NoOfBytes;
        output  [7:0]           NoOfLoc;


        wire    [9:0]   cnt;
        wire
        MACK1,cnttest,sclout0,scl_select2,scl_select1,scl_hold_sel,clkSout2,clkSout1,sclout4,s
clout3,sclout2,rstclk1,First_byte,MorS_sel,sdaout3,sda_select,scl_select,C10_en;
        reg     C10_reset;
        wire resend1,sdaoutg_sel,sclout1_delay,beginT1,A1_ld1,TxRx2,TxRx3, TxRx4,
MACK_detect_in, MACK_detect1,clk;
        reg     Go,go_rst,MACK_detect;
        parameter N = 8; //Number of bytes to send
//      Clk_Div64 Div(clk,rst,fpga_clk);
        assign clk = fpga_clk;

        assign sdatest = rst|P;
        reg_1bit        GoReg(resend1,resend,1'b1,Go,clk);
        always @(negedge sclout1_delay)
                begin
                        if(rst|go_rst)
                                begin
                                go_rst <= 1'b0;

                                Go <= 1'b0;
                                end
                        else if(resend1)
                                begin
                                        Go <= resend1;
                                        go_rst <= 1'b1;

                                end
                end

        SandP_Detect  SPDET(sdain, sclin, P,rst,clk); // Start-stop bit register
        wire    scl_rls_sel;
////////////////////////////////// Master Components
//////////////////////////////////////////////////
//      Rst signal will be rst|Go|WR|~cs, so if Go, reset, if WR, reset, if not chip select,
reset.
```

```
        SandP_Control
SPCON(STOP,MACK1,Go,NoOfLoc,NoOfBytes_ready,No82,No83,No84,No85,No_Ack,MorS_sel,sdaout3,sda_s
elect,scl_rls_sel ,scl_hold_sel,scl_select1,scl_select,data_out,sdain,sclout4,sclin,TxRx,
rst|Go|~cs,rst,P,sclout1_delay,clk);
//
        (MACK1,Go,NoOfLoc,No82,No83,No84,No85,No_Ack,master_slave_sel,sda_out_start,sda_select
,scl_rls_sel ,scl_hold_sel,scl_select1,scl_select,sdain, sclout1,sclin,
rst,hard_rst,P,sclout1_delay,clk);
        reg_1bit
        Rteyt8(scl_select2,scl_select1,1'b1,rst|P,~sclout1_delay);
        mux2to1_1bit   sclmux_final(sclout,scl_hold_sel,sclout0,1'b1);

        mux2to1_1bit   sclmux(sclout0,scl_select&scl_rls_sel,1'b0,~sclout1);
        mux2to1_1bit   SorMmux(sdaout4,MorS_sel,~First_byte,sdaout2);//MorS = 0: master, 1:
slave mode
        mux2to1_1bit   FinalSdaMux(sdaout,sda_select,sdaout3,sdaout4);//select start, stop
signal
        Delay_2Clk             delay2(sclout3,sclout1,rst,~clk);
        Delay_2Clk             delay3(sclout4,sclout3,rst,clk);
        Clk_Div512     Div512(sclout1,rst,clk);
        Clk_Div1024 Div1024(sclout2,rst,clk);
        reg_1bit  R6(sclout1_delay,sclout1,1'b1,rst,sclout2);//delay 1/4 sclout1 clk cycle
//                             Clk_Div512(div_clk,rst,sclk);
        Shift_reg      Sreg(First_byte,Start_Byte,rst|Go,sclin,sclout1,clk);
//      Shift_reg(out,in,rst,scl,clk);

//      Reg_8bit_Load_Once
        NoOfRAMLocations(cnttest,NoOfLoc,data_out,No82,TxRx1,beginT2,rst|P,bit8_4,clk);
        //
        Reg_8bit_Load_Once(cnt,out,in,in1,TxRx,load,rst,sclk,pclk);
//      assign  NoOfBytes = NoOfLoc;//NoOfLoc;
//      mux2to1_1bit   mux1(No82_1,TxRx1,NoOfBytes,No82);
//      mux2to1_1bit   mux2(No83_1,TxRx1,NoOfBytes + 1'h1,No83);
//      mux2to1_1bit   mux3(No84_1,TxRx1,NoOfBytes + 2'h2,No84);
//      mux2to1_1bit   mux4(No85_1,TxRx1,NoOfBytes + 2'h3,No85);



////////////////////////////   End of Master components ////////////////////////////////

//      assign INT=ACK|RD;
//      assign sclout=WR;
// bus buffer multiplexer.
//      mux_4to1_8bit BusBufferMux(BB_in, address, data_out,ext_data_out ,ADD1_out, ADD2_out);
//                DR(sout, sin, data_in,data_out,load,SorPL,sclk,clk,rst)
        Data_Reg DataReg(DR_clk_out,sout,sdain, 8'b00000011
,data_out,1'b1,~SorPL,sclin,clk,rst|P);
        Delay_2Clk ClkDelay(bit8_1,bit8,rst,clk);
        Delay_2Clk ClkDelay1(bit8_2,bit8_1,rst,~clk);
        Delay_2Clk ClkDelay2(bit8_3,bit8_2,rst,clk);
        Delay_2Clk ClkDelay3(bit8_4,bit8_3,rst,~clk);

// Serial or Paralell load signal
//      assign SorPL0 = TxRx&beginT&~P&bit9&sclin;
        assign SorPL = ~bit9_4&TxRx&beginT&~P&bit9;
//beginT=A1&A2: When addresses are equal, start transfering data
// Serial or Paralell load register
        Delay_2Clk delay21(bit9_1,bit9,rst,~clk);
        Delay_2Clk delay31(bit9_2,bit9_1,rst,clk);
        Delay_2Clk delay41(bit9_3,bit9_2,rst,~clk);
        Delay_2Clk delay51(bit9_4,bit9_3,rst,clk); // delay bit9, make sure there is no extra
value for
//      SorPL
//      SorPL0 = 1 if transmit, 0 if receive (or serial)
//      assign SorPL0 = TxRx&beginT&ACK&~clkSout;
// Serial or Paralell load register
//      reg_1bit SorPL_Reg(SorPL, SorPL0, 1'b1, rst, ~clk);
        Delay_2Clk D3(clkSout1,clkSout,rst,clk);
        Delay_2Clk D4(clkSout2,clkSout,rst,~clk);//make sure that it is delayed

        reg_1bit SDA_Buffer1_Neg(sout1, sout, 1'b1, rst, ~clkSout2);
//delay for half sclin cycle and put the data out on negedge
//that makes the data changed only in when sclin is 0 and data is stable when sclin is high.
```

```
        reg_1bit Clk_Sync(clkSout, sclin, 1'b1, rst, clk);
//sync clin with the internal clk
//Serial data out multiplexer.(out, sel, in0, in1)
        mux2to1_1bit SDA_Mux1(sout2,SDA_MUX_SEL , 1'b1, sout1);
        assign SDA_MUX_SEL = beginT&TxRx ;//TxRx = 1: transfer data; TxRx = 0, receive data
//if address is correct and want to send data --> select sout1 at the output of the mux
//if not, select logic high at the output of the mux.
        assign sdaoutg_sel = MACK1|MACK_detect|/*(bit9&~sclin)|*/P;
        mux2to1_1bit SDA_Mux2(sdaoutg,sdaoutg_sel,sdaout1,1'b0);
//      Define times to release the sda line:
/*      + At negedge of 9th clk, release the line if there is no MACK (MACK1 = 1'b1) to detect
stop transfering
        data: MACK1
        + MACK_detect high for the transmitter to release the sdaline at the negedge of 8th
bit, so that the
        receiver can ACK the transmitter at the 9th clk
        + Right after negedge of bit 9th, release the line (because if receiver, then I
already sent the
        ACK at the posedge of 9th bit, if transmitter, I dont want to use sdaline until bit
1??? Correct?
        + Obviously, during stop, I dont want to use the line.
*/
        assign sdaoutmux = SDA_MUX_SEL&(bit8|bit9)&ACK;
//Serial data out register.(out, sel, in0, in1)
//      reg_1bit SDA_OUT_REG(sdaout2, sdaoutg, 1'b1, rst, clk);
        Delay_2Clk delay(sdaout2,sdaoutg,rst,clk);
// Bus Buffer REG
        Reg_8bit BB(BB_out,BB_in,BB_load, clk, rst);

        assign BB_load = bit8&beginT&~TxRx;
        assign ext_data_out=BB_out;

// ADD REG
// Right now, just write the address to addr reg, and leave it there, Master is just a
master, not a slave.
   Reg_8bit Add_reg1(ADD1_out,8'h05,ADD1_load,clk,rst);
//   Reg_8bit Add_reg2(ADD2_out,ext_data_in,ADD2_load,clk,rst);

        assign ADD1_load= address[1]&~address[0]&WR;
//      assign ADD2_load= address[1]& address[0]&WR;

// COMPARATOR REG
//      No comparator, just load A1 at the bit 8, sothat we can remain the timing constrain as
in the Slave.
   reg_1bit Address1(A1, 1'b1,bit8, rst|P, ~sclout1); //always load 1 to A1, that makes the
master work
//      reg_1bit Aadd2(A2, comp02,A2_ld, rst|P, clk);

/////TxRxTxRxTxRxTxRxTxRxTxRxTxRx TxRx from here
TxRxTxRxTxRxTxRxTxRxTxRxTxRxTxRx//////////////////////////////////////////


// Transfer Receive REG
        reg_1bit       beginT_delay(beginT1,beginT,1'b1,rst,sclin);
        reg_1bit       beginT_delay1(beginT2,beginT1,1'b1,rst,clk);

        reg_1bit_loadonce TxorRx1(TxRx1, ~data_out[0],bit8_2, rst|P, clk);
        reg_1bit_n TxorRx3(TxRx3, TxRx1,1'b1, rst, sclin);
        reg_1bit TxorRx(TxRx, TxRx1,1'b1, rst, sclin);
//       reg_1bit_n(out, in,load,clear, clk);
        reg_1bit_n TxorRx2(TxRx2, TxRx,1'b1, rst, sclin);
        reg_1bit TxorRx4(TxRx4, TxRx2,1'b1, rst, sclin);

//      TxRx4 is high only after the first 9 bits of the address, or only high at the begining
//      of 2nd byte
//      Delay TxRx for another cycle
//      Delay TxRx for one cycle, TxRx will be high at the negedge of 9th bit, that makes the
// A1_ld must be long enough to make sure that it will be high when there is a posedge of
sclin
// Detect 8th bit to see if transmit or receive, if TxRx=1, then the server wants to read
from the salve
// otherwise, the server wants to send data to the slave
        assign Tx_nRx=TxRx&~P;
```

```
// A1 and A2 load
        mux2to1_1bit A1_ld_mux(A1_ld, bit8, 1'b0, ~beginT);
// A1 load is used to load the comparator reg. It happens only at the first 8th bit of sdain

        mux2to1_1bit A2_ld_mux(A2_ld, bit17, 1'b0, ~beginT);
// Unable to detect changes in direction, once beginT is high, it will be high forever
because A1_ld
// is not high again, A1 and A2 keep the same value forever
// I want to change A2_ld, I want to extend it longer
        //reg_1bit regA1Load(beginT, A1, 1'b1,rst, sclin);
        assign beginT=A1;

//SCL Counter 5-bit
        Counter_5bit Add_counter(ADDC_out, 5'b00001,~P, Add_cou_ld,rst|P/*|(MACK&bit9)*/,
sclin^((rst|P)&clk),clk);
//Counter_5bit(out, in,en, load, clear, clk, fpga_clk);
//use load signal to reset the counter, whenenver reaches bit 9 or bit 18, reset counter. Is
this correct?
//should load when there is an ACK (either Master or Slave ACK)! Should Clear when the stop
signal is high
//clock for the counter is just the sclin.
        assign Add_cou_ld= (bit9&beginT&~P);//|(bit1&~beginT);
// Address Comparators
/*      No Comparator
        assign comp01=(~(ADD1_out[0]^data_out[1]|ADD1_out[1]^data_out[2]
                |ADD1_out[2]^data_out[3]|ADD1_out[3]^data_out[4]|ADD1_out[4]^data_out[5]
                |ADD1_out[5]^data_out[6]|ADD1_out[6]^data_out[7]));
        assign
comp02=(~(ADD2_out[0]^data_out[0]|ADD2_out[1]^data_out[1]|ADD2_out[2]^data_out[2]

        |ADD2_out[3]^data_out[3]|ADD2_out[4]^data_out[4]|ADD2_out[5]^data_out[5]
                                        |ADD2_out[6]^data_out[6]|ADD2_out[7]^data_out[7]));
*/
        assign bit1=ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&~ADDC_out[4];
        assign bit7=ADDC_out[0]&ADDC_out[1]&ADDC_out[2]&~ADDC_out[3]&~ADDC_out[4];

        assign bit8=~ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&ADDC_out[3]&~ADDC_out[4];
// assign bit16=~ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&ADDC_out[4];
        assign bit9=ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&ADDC_out[3]&~ADDC_out[4];
   assign bit17=ADDC_out[0]&~ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&ADDC_out[4];
        assign bit18=~ADDC_out[0]&ADDC_out[1]&~ADDC_out[2]&~ADDC_out[3]&ADDC_out[4];
        assign Counter0=ADDC_out[0]|ADDC_out[1]|ADDC_out[2]|ADDC_out[3]|ADDC_out[4];
// ACKNOWLEDGE
                                                //(out, sel, in0, in1)
mux2to1_1bit ACK_MUX(sdaout1, ACK, soutinv, 1'b1);
/* Question: When should I send ACK?
        Send ack only in transfering mode and send out at bit 9*/
not (soutinv,sout2);

reg_1bit ACK_reg0(ACK1,ACK_in,1'b1, rst, clk);
reg_1bit_nCLK ACK_reg(ACK,ACK1,1'b1, rst|bit9_4, sclin/*clkSout*/, clk);
// Load ACK1 to ACK during negedge of bit 8
//reg_1bit_nCLK (out,in,load,clear, sclk,pclk);
// when rst = 0, clk = clkSout ( or sclin afer being synced with clk), load at the negedge of
the sclin
// that guarantees that the sda will not be pulled down when sclin is still high.
assign ACK_in= bit8&~TxRx4&No_Ack;//|(A2&bit17);
//      For the slave: ACK_in: If address is correct and in receive mode, send ACK. And send
ACK only for the first byte in
//      transfering mode, send ACK at the end of every byte in the receiving mode.

//      For the MASTER:
//      +       Donot send ACK for the first byte in any case
//      +       If transmit, detect ack from slave (that's what MACK1 already did)
//      +       If receive, send ACK after the first byte ANDDDDDD: Do not send ack at the
last byte (No_Ack)
//      if the output of the counter counts to 8 and if address is ok --> ACK? No, should have
more
//      logic to check for ACK condition. Add busy signal to the input of the I2C, and there
should be
//      TxRx there, the slave send ACK only when it is in the receiving status
//      Master acnoligment register// Master acnoligment register
//      assign MACK_detect = A1&TxRx3&TxRx2;
```

```
//      reg_1bit MACK_detect_reg0(MACK_detect1,MACK_detect_in,1'b1, rst, clk);
//      reg_1bit_nCLK MACK_detect_reg(MACK_detect,MACK_detect1,1'b1, rst, sclin/*clkSout*/,
clk);

//      MACK_detect high for the transmitter to release the sdaline at the negedge of 8th bit,
so that the
//      receiver can ACK the transmitter at the 9th clk
always @(negedge sclin)
        if(rst)
                MACK_detect <= 1'b0;
        else if(bit8&TxRx2)//that means there is no MACK_detect at the start byte
                MACK_detect <= 1'b1;
        else
                MACK_detect <= 1'b0;

        reg_1bit_loadonce MACK_reg(MACK,sdain,~P&TxRx3&bit9_4,rst|P|bit7, clk);
//      reg_1bit_loadonce                               (out,in,load,rst,sclk,pclk);mack is
loaded EVERY bit9_4, so, use bit 7 to reset
// dont use bit one, because of extra sclin posedge and negedge from the master, it makes
extra bit 1 :D
//      Why do you need TxRx4? Because the master always sends the first byte. and the slave
always Ack 1st byte.
//      so, for slave, dont need to detect the first MACK, but for master, we need to detect
that. Thus, donot
//      use RxTx4. Use TxRx3 instead, TxRx3 is high at negedge of 8th bit.
reg_1bit_n      MACK_reg1(MACK1,MACK,1'b1,rst,sclin);
// Delay MACK for hafl clk cycle, for a transmitter stop transfering data detection
// Detect master ACK. For simulation purpose, input should be "sdain&~sdaout"


//      reg_1bit MACK_reg(MACK,sdain,/*SorPL0&sclin*/~P&TxRx3&bit9&sclin, rst|P|bit1, clk);
//
////    reg_1bit_n      MACK_reg1(MACK1,MACK,1'b1,rst,sclin);
////    The above instruction may cause MACK1 is always 1, must be carefull
//      assign rst_clk = sclin^(rst&clk);
//      always@(negedge rst_clk)
//              begin
//                      if(rst)
//                              MACK1 <= 1'b0;
//                      else
//                              MACK1 <= MACK;
//              end


//      MACK1 is actually received ACK for both master or slave in transfering mode
// Delay MACK for hafl clk cycle, for a transmitter stop transfering data detection. At any
time, If no
//      MACK, release the sda line. That's true for both master and slave.
// Detect master ACK. For simulation purpose, input should be "sdain&~sdaout"
assign test[0] = beginT1;//b6
assign test[1] = beginT2|P;//E7
assign test[2] = beginT;//F7
assign test[4] = sout;//a10
assign test[5] = ~First_byte;//a11
assign test[6] = MorS_sel;//a12
assign test[7] = MorS_sel;//a13
assign test[8] = sdaout3;//a14
assign test[9] = P;//a15
assign test[10] = sdaout1;//a16
assign test[11] = sdaout2;//a17
assign test[12] = rstclk14;//a18
assign test[13] = data_out[0];
assign test[14] = data_out[1];
assign test[15] = data_out[2];
assign test[16] = data_out[3];
assign test[17] = data_out[4];
assign test[18] = data_out[5];
assign test[19] = data_out[6];
assign test[20] = DR_clk_out;
endmodule
```

Buffer 96x8:

```
module buffer(out,data_in,load,RnW,rst,P,Pclk,rd_clk,clk);
//buffer b(buf_out,data_out,beginT2|P,TxRx|P,rst,P,Bus2IP_Clk,slv_reg_read_sel[4],bit8_1);
output  [7:0]   out;
input           [7:0]   data_in;
input                           rd_clk,P,rst,Pclk,load,RnW,clk;
reg             [7:0]   R       [95:0];
wire            [7:0]   mux_out [95:0];
integer i,k;
reg             [7:0]   tmp,cnt;
wire                            rd_clk4,rd_clk3,rd_clk2,rd_clk1,rst1,rst2,clk1;
//Delay rd_clk or read reg_slv4 to make sure read correct data before changing data.
        reg_1bit        D3(rd_clk1,rd_clk,1'b1,1'b0,Pclk);
        reg_1bit D4(rd_clk2,rd_clk1,1'b1,1'b0,~Pclk);
        reg_1bit        D5(rd_clk3,rd_clk2,1'b1,1'b0,Pclk);
        reg_1bit D6(rd_clk4,rd_clk3,1'b1,1'b0,~Pclk);

mux2to1_1bit   muxclk(clk1,rst|P,clk,(rst&Pclk)|rd_clk4);//after reset, if stop, rd_clk will
be selected
        reg_1bit        D1(rst2,rst,1'b1,1'b0,Pclk);
        reg_1bit D2(rst1,rst2,1'b1,1'b0,~Pclk);
always @(posedge clk1) begin
                if(rst1)
                        begin
                                cnt <= 8'b11111111;
                                for(i = 0; i <96; i = i +1)
                                R[i] <= 8'b00000000;
                        end
                else if(load)
                        begin
                                for(i = 0; i <96; i = i +1)
                                        R[i] <= mux_out[i];
                                if(~RnW)
                                        cnt <= cnt + 1'b1;//if write, increase cnt to point to
top of the buffer
                        end


                else
                                for(i = 0; i <96; i = i +1)
                                R[i] <= R[i];
  end

mux2to1_8bit Mout(out, RnW,8'hzz, R[cnt]);//If read, read from top of buffer and shift the
other location
// to top of buffer.
mux2to1_8bit M0(mux_out[0], RnW, data_in, R[cnt]);//If read, feed top of buffer back to entry
of buffer

assign mux_out[1] = R[0];
assign mux_out[2] = R[1];
assign mux_out[3] = R[2];
assign mux_out[4] = R[3];
assign mux_out[5] = R[4];
assign mux_out[6] = R[5];
assign mux_out[7] = R[6];
assign mux_out[8] = R[7];
assign mux_out[9] = R[8];
assign mux_out[10] = R[9];
assign mux_out[11] = R[10];
assign mux_out[12] = R[11];
assign mux_out[13] = R[12];
assign mux_out[14] = R[13];
assign mux_out[15] = R[14];
assign mux_out[16] = R[15];

assign mux_out[17] = R[16];
assign mux_out[18] = R[17];
```

```
assign mux_out[19] = R[18];
assign mux_out[20] = R[19];
assign mux_out[21] = R[20];
assign mux_out[22] = R[21];
assign mux_out[23] = R[22];
assign mux_out[24] = R[23];
assign mux_out[25] = R[24];
assign mux_out[26] = R[25];
assign mux_out[27] = R[26];
assign mux_out[28] = R[27];
assign mux_out[29] = R[28];
assign mux_out[30] = R[29];
assign mux_out[31] = R[30];
assign mux_out[32] = R[31];

assign mux_out[33] = R[32];
assign mux_out[34] = R[33];
assign mux_out[35] = R[34];
assign mux_out[36] = R[35];
assign mux_out[37] = R[36];
assign mux_out[38] = R[37];
assign mux_out[39] = R[38];
assign mux_out[40] = R[39];
assign mux_out[41] = R[40];
assign mux_out[42] = R[41];
assign mux_out[43] = R[42];
assign mux_out[44] = R[43];
assign mux_out[45] = R[44];
assign mux_out[46] = R[45];
assign mux_out[47] = R[46];
assign mux_out[48] = R[47];

assign mux_out[49] = R[48];
assign mux_out[50] = R[49];
assign mux_out[51] = R[50];
assign mux_out[52] = R[51];
assign mux_out[53] = R[52];
assign mux_out[54] = R[53];
assign mux_out[55] = R[54];
assign mux_out[56] = R[55];
assign mux_out[57] = R[56];
assign mux_out[58] = R[57];
assign mux_out[59] = R[58];
assign mux_out[60] = R[59];
assign mux_out[61] = R[60];
assign mux_out[62] = R[61];
assign mux_out[63] = R[62];
assign mux_out[64] = R[63];

assign mux_out[65] = R[64];
assign mux_out[66] = R[65];
assign mux_out[67] = R[66];
assign mux_out[68] = R[67];
assign mux_out[69] = R[68];
assign mux_out[70] = R[69];
assign mux_out[71] = R[70];
assign mux_out[72] = R[71];
assign mux_out[73] = R[72];
assign mux_out[74] = R[73];
assign mux_out[75] = R[74];
assign mux_out[76] = R[75];
assign mux_out[77] = R[76];
assign mux_out[78] = R[77];
assign mux_out[79] = R[78];
assign mux_out[80] = R[79];

assign mux_out[81] = R[80];
assign mux_out[82] = R[81];
assign mux_out[83] = R[82];
assign mux_out[84] = R[83];
assign mux_out[85] = R[84];
assign mux_out[86] = R[85];
assign mux_out[87] = R[86];
```

```
assign mux_out[88] = R[87];
assign mux_out[89] = R[88];
assign mux_out[90] = R[89];
assign mux_out[91] = R[90];
assign mux_out[92] = R[91];
assign mux_out[93] = R[92];
assign mux_out[94] = R[93];
assign mux_out[95] = R[94];
endmodule
```

SandP_Control:

```
module SandP_Control(STOP,MACK1,Go,NoOfLoc,NoOfBytes_ready,No82,No83,No84,No85,No_Ack
,master_slave_sel,sda_out_start,sda_select,scl_rls_sel
,scl_hold_sel,scl_select1,scl_select,data_out,sdain, sclout1,sclin,
TxRx,rst,hard_rst,P,sclout1_delay,clk);
    //
        SPCON(No_Ack,MorS_sel,sdaout3,sda_select,scl_hold_sel,scl_select,sdain,sclout4,sclin,
rst,P,sclout1_delay,clk);
         input NoOfBytes_ready,MACK1,hard_rst,Go,sclout1_delay,P,sclout1,sdain, sclin,rst,clk;
    output scl_select1,scl_rls_sel ,No_Ack,master_slave_sel, sda_out_start,sda_select,
        scl_select;
        output  STOP,scl_hold_sel;//remember to change the functioncall for sapcontrol,
scl_hold_sel hold scl line
       input   [7:0]   data_out;
        reg scl_rls_sel ,scl_hold_sel,sda_out1, sda_select, scl_select;
        wire
rst4,rst3,rst1,rst2,rstclk2,rstclk3,rstclk4,rstclk5,rstclk6,rstclk7,rstclk8,rstclk9,rstclk10,
rstclk11,rstclk12,rstclk13,rstclk14,rstclk0,bit11,a,rstclk1;
        reg STOP,scl_select1,test,No_Ack;
        reg    [31:0]  cnt;//Number of bytes in on transactions (from start to stop)
        input  [31:0]  No82,No83,No84,No85;
       reg             [31:0]  Q;//No Of Bytes
       output [7:0] NoOfLoc;
       reg    [7:0]   NoOfLoc;
       input TxRx;



       assign rstclk0=sclout1_delay^(rst&clk);//Use this clk for the 83th clk because sclin
only counts up to 82

//     and then reset.

//     Delay rst for 1 clk cycle, make sure that there is no uneccessary posedge, which may
ruins
//     the initial conditions of the circuit
//     always @(posedge clk)
//             if(rst)
//                     rst1 <= rst;
//             else
//                     rst1 <= 1'b0;
       reg_1bit       D1(rst2,rst,1'b1,1'b0,clk);
       reg_1bit D2(rst1,rst2,1'b1,1'b0,~clk);
       reg_1bit       D3(rst3,rst1,1'b1,1'b0,clk);
       reg_1bit D4(rst4,rst3,1'b1,1'b0,~clk);
//     Delay_2Clk(out, in, rst, clk);
       always @(posedge sclout1_delay)
              begin
                     if(rst)
                            begin
                                   cnt <= 32'h00000000;
                                   scl_rls_sel <= 1'b1;
                            end
                     else if((sclin == 1'b0)|(cnt == Q + 3'b100))
//At 85th bit, do not keep counting up until another reset
                            begin
                                   scl_rls_sel <= 1'b0;
//release the scl line so that the slave can pull it low.
                                   cnt <= cnt;
// detect holding time of the sclin, that's why we need slcout1_delay
```

```
                        end
                    else
                        begin
                            cnt <= cnt + 1;
                            scl_rls_sel <= 1'b1;
// after holding the sclline, the slave release the scl to make scl = 1
                        end
            end
        always @(negedge sclout1)
            begin
                    if(hard_rst)
//Only a hard reset can reset this signal. If there is no ack, stop!!
                        STOP <= 1'b0;
                    else if(MACK1)
                        STOP <= 1'b1;
                    else if(rst1|STOP)
                        begin
                            scl_select1 <= 1'b0;
                            sda_out1 <= 1'b0;
                            sda_select <= 1'b0;
                            scl_hold_sel <= 1'b0;
                            NoOfLoc <= 8'h00;
                            Q <= 32'h00000025;
//by default, receive 3 bytes at least from the slave.
                            scl_select1 <= 1'b0;
                            scl_select <= 1'b0;
                            No_Ack <= 1'b1;


                        end


                    else if(cnt == 32'h00000001)
                        begin
                            sda_out1 <= 1'b1;

                        end
                    else if(cnt == 32'h00000002)
                        begin
                            sda_select <= 1'b1;// select actual sda
                            scl_select <= 1'b1;// select actual scl
                        end
                    else if(cnt == 32'h00000007)
//make sure there is no ack at the first byte
                        No_Ack <= 1'b0;
                    else if(cnt == 32'h0000000C)
                        begin
                            No_Ack <= 1'b1;
                            if(TxRx)//TxRx = 1 -> write data to the slave, master
decides NoOfbytes
                                Q <= No82;
                        end
                    else if(cnt == 32'h00000013)//at bit 18th, load the content of DR to
decide NoOfBytes
                        begin
                            if(~TxRx)
                                begin
                                    NoOfLoc[4:0] <= data_out[6:2];
//Slave decides number of bytes to send to master.
                                    if(~NoOfBytes_ready)
//MB did not finish calculate no of bytes yet, this signal received
//from the same s/w Reg as the No82
                                    scl_hold_sel <= 1'b1 ;
//hold scl line until receive number of bytes calculated from MB
                                    else
                                        begin
                                            scl_hold_sel <= 1'b0;
                                            Q <= No82;
//Receive No of bytes from MB, in the same s/w Reg.
                                        end
//this signal also makes the counter stop counting, because scl line is low, and also
//makes scl_rls_sel = 0, or select 0 at output of the first scl mux (or release scl),
//but scl_hold_sel is the outter mux, and the scl line is still low.
```

```
//Because counter stop counting -> stay in this condition

                                        end
                        end

                else if(cnt == Q -1) //82, donot send ACK: (NoOfBytes+1)*9 + 1
                        No_Ack <= 1'b0;
                else if(cnt == Q) //82, donot send ACK: (NoOfBytes+1)*9 + 1
                        No_Ack <= 1'b0;
                else if(cnt == Q + 1)//83, select stop signal: (NoOfBytes+1)*9 + 2
                        begin
                        sda_select <= 1'b0;

                        end
                else if(cnt == Q + 2'b10)//84, select stop signal: (NoOfBytes+1)*9 + 2
                        begin
                        scl_select <= 1'b0;

                        end
                else if(cnt == Q + 2'b11)//85, select stop signal
                        begin
                                No_Ack <= 1'b1;
                                sda_out1 <= 1'b0;
                        end

                else
                        begin
                                sda_select <= sda_select;
                                scl_select <= scl_select;
                        end
        end
        reg_1bit       MorS(master_slave_sel,1'b1,bit11,P|rst,~rstclk0);
//after bit 11, the master will operate exact the
// same as the slave. If master_slave_sel = 0 select master, 1 select slave mode.
        assign bit11 = ~cnt[7]&~cnt[6]&~cnt[5]&~cnt[4]&cnt[3]&~cnt[2]&cnt[1]&cnt[0];
// 00001011
        assign sda_out_start = sda_out1;

endmodule
```

Clock Divider:

```
module Clk_Div1024(div_clk,rst,sclk);
input sclk,rst;
output  div_clk;
reg     [7:0]   cnt;
reg     tmp;
always @(posedge sclk)
begin
        if(rst == 1'b1)
                begin
                        cnt <= 8'h00;
                        tmp <= 1'b0;
                end
        else if(cnt == 8'b01111111)
                begin
                        tmp <= ~tmp;
                        cnt <= 8'h00;
                end
        else
                cnt <= cnt + 1'b1;

end

assign div_clk = tmp;
endmodule

module Clk_Div512(div_clk,rst,clk);
input clk,rst;
output  div_clk;
reg     [7:0]   cnt;
reg     tmp;
```

```
always @(posedge clk)
begin
	if(rst == 1'b1)
		begin
			cnt <= 5'h00;
			tmp <= 1'b0;
		end
	else if(cnt == 8'b11111111)
		begin
			tmp <= ~tmp;
			cnt <= 8'h00;
		end
	else
		cnt <= cnt + 1'b1;

end

assign div_clk = tmp;
endmodule
```

Delay module:

```
module Delay_2Clk_8bits(out, in, rst, clk);


output  [7:0]   out;

input           [7:0]   in;

input                           rst,clk;

reg     [1:0] cnt;

reg     [7:0] tmp;

always @(posedge clk)

begin

	if(rst == 1'b1)

		begin

		cnt <= 2'b00;

		tmp <= 1'b0;

		end

	else if(cnt == 2'b10)

		begin

		tmp <= in;

		cnt <= 2'b00;

		end

	else

		cnt <= cnt + 1;

end


assign out = tmp;

endmodule
```

```
module Delay_8Clk(out, in, rst, clk);


output  out;
input   in,rst,clk;
reg     [2:0]cnt;
reg     tmp;
always @(posedge clk)
begin
        if(rst == 1'b1)
                begin
                cnt <= 3'b000;
                tmp <= 1'b0;
                end
        else if(cnt == 3'b111)
                tmp <= in;
        else
                cnt <= cnt + 1;
end

assign out = tmp;
endmodule
```

One-bit positive-edge Register:

```
module reg_1bit_p(out, in,load,clear, clk);
        input           load;
      input       in;     /* set input */
        input       clear;   /* reset input */
        input       clk;    /* clock input */
        output          out;    /* output of register */
        reg       Q;      /* one bit register to hold count */
        /* State changes only on negative clock transitions */
        always @(posedge clk)
        begin
                if(clear == 1'b1) /*  */
                        Q <= 1'b0;
```

```
		else if(load == 1'b1) /* */
			Q <= in;
		else
			Q <= Q;
	end
	assign out = Q;
endmodule
```

Shift Register:

```
module Shift_reg(out,in,rst,scl,sclout1,clk);
//			Sreg(First_byte,8'h02,rst|P,sclin,sclout1,clk);
output	out;
input	[7:0]	in;
input	sclout1,scl,rst,clk;
reg	rst1;
reg	[8:0]	a;
wire	rstclk1;
	assign rstclk1=scl^(rst&clk);

//	Delay rst for 1 clk cycle, make sure that there is no uneccessary posedge, which may
ruins
//	the initial conditions of the circuit
	always @(posedge clk)
		if(rst)
			rst1 <= rst;
		else
			rst1 <= 1'b0;
	always @(negedge rstclk1) // do at negedge of sclin, so data is available before the
rising edge
//	and do it after the first bit so that there will be no missing data
	if (rst1)
		begin
			a[0] <= in[0];
			a[1] <= in[1];
			a[2] <= in[2];
			a[3] <= in[3];
			a[4] <= in[4];
```

```
                        a[5] <= in[5];
                        a[6] <= in[6];
                        a[7] <= in[7];
                end
        else
                begin

                        a[8] <= a[7];
                        a[7] <= a[6];
                        a[6] <= a[5];
                        a[5] <= a[4];
                        a[4] <= a[3];
                        a[3] <= a[2];
                        a[2] <= a[1];
                        a[1] <= a[0];
                end
assign out = a[8]; //1st negedge after rst: a[7]->a8;2nd: a[6]->a8
endmodule
```

APPENDIX C

Code for I2C Microcontroller

C Program

```
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "xstatus.h"
#include "xuartlite.h"

/************************** Constant Definitions *****************************/

/*
 * The following constants map to the XPAR parameters created in the
 * xparameters.h file. They are defined here such that a user can easily
 * change all the needed parameters in one place.
 */
#define UARTLITE_DEVICE_ID          XPAR_RS232_DCE_DEVICE_ID

/*
 * The following constant controls the length of the buffers to be sent
 * and received with the UartLite, this constant must be 16 bytes or less since
 * this is a single threaded non-interrupt driven example such that the
 * entire buffer will fit into the transmit and receive FIFOs of the UartLite.
 */
#define TEST_BUFFER_SIZE 16
/************************** Function Prototypes ******************************/

XStatus UartLitePolledExample(Xuint16 DeviceId);

/************************** Variable Definitions *****************************/

XUartLite UartLite;                   /* Instance of the UartLite Device */
/*
 * The following buffers are used to send and receive data
 * with the UartLite.
 */
Xuint8 SendBuffer[TEST_BUFFER_SIZE];    /* Buffer for Transmitting Data */
Xuint8 RecvBuffer[TEST_BUFFER_SIZE];    /* Buffer for Receiving Data */



volatile unsigned int *I2C_Master = (unsigned int *) 0xcae00000 ;
//=====================================================
int main (void) {
       unsigned int i,j,k;
       unsigned int temp = 0x00000000,temp1 = 0x00000000;
       unsigned int SentCount;
   unsigned int ReceivedCount;
   int Index;
       Xuint32       abc,buffer[96];
       Xuint8 buffer1[96];
           XStatus Status;

               temp = 0x00000000;
               temp1 = 0x00000000;
               while(*(I2C_Master + 0x2) == 0xffffffff)//P = 0, started. do nothing
                       {;}
               for(Index = 0;Index <16;Index++)
                       {
                       RecvBuffer[Index] = 0;
                       }

               Status = XUartLite_Initialize(&UartLite, UARTLITE_DEVICE_ID);

               while(RecvBuffer[0] == 0)
```

```
                {
                ReceivedCount = XUartLite_Recv(&UartLite, RecvBuffer, 1);
//              *(I2C_Master) = 0x00000001;//chipsel is low, do not select master.
                }
               *(I2C_Master) = 0x00000201 + RecvBuffer[0]*2;//Read from Slave 1, reg0 is for
sending control signal to the master

               *(I2C_Master) = 0x00000301 + RecvBuffer[0]*2; //resend
               *(I2C_Master) = 0x00000301 + RecvBuffer[0]*2;
               *(I2C_Master) = 0x00000301 + RecvBuffer[0]*2;
               *(I2C_Master) = 0x00000301 + RecvBuffer[0]*2;
               *(I2C_Master) = 0x00000301 + RecvBuffer[0]*2; //resend

               *(I2C_Master) = 0x00000201 + RecvBuffer[0]*2;//chipsel is always high
               //Now we can read NoOfLoc from slv_reg3:
               *(I2C_Master + 0x1) = 0x00000000;
               while(*(I2C_Master + 0x3)==0x00000000)
                       {;}

                       for(i=0;i<3;i++)
                               temp = temp + *(I2C_Master + 0x3) + 0x1;//*(I2C_Master + 0x3) =
5'b11111 = Dec31
                       temp = temp + 0x1;
                       for(i=0;i<9;i++)
                               temp1 = temp1 + temp;
                       *(I2C_Master + 0x1) = temp1 +0x1;
                       while(*(I2C_Master + 0x1) == 0)
                               {;}
                       *(I2C_Master) = 0x00010201+ RecvBuffer[0]*2;//Signal the master that MB
finishes calculating No Of Bytes
                       while(*(I2C_Master + 0x2) == 0xffffffff)//if I2C does not finish yet,
wait.
                       {;}
                       //finish? start getting data back.
                       for(i=0;i<96;i++)
                               buffer[i] = 0;
                       for(i=0;i<96;i++)
                       {
                       buffer[i] = *(I2C_Master + 0x4) ;//Now, data is stored in this buffer.

                       }

                       for(i=0;i<96;i++)
                       {
                       ;//delay
                       }
                       for(i=0;i<96;i++)
                       {
                       buffer1[i] = (Xuint8)buffer[i];//casting.
                       }
                       for(i = 0; i <6;i++)
                       {
                               for (Index = 0; Index < TEST_BUFFER_SIZE; Index++)
                               {
                                       SendBuffer[Index] = buffer1[Index +
i*16];//buffer[Index];

                               }

                               SentCount = XUartLite_Send(&UartLite, SendBuffer,
TEST_BUFFER_SIZE);//this returns no of bytes sent
                               //use XUartLite_Send to send data in SendBuffer
                               while (XUartLite_IsSending(&UartLite))
                               {;}//wait until finish sending first buffer to send second
buffer
       //              for(Index = 0;Index <16;Index++)
       //              {
       //              RecvBuffer[Index] = 0;
       //              }

                       }

}
```

The Wrapper (master5.vhd)

```
------------------------------------------------------------------------------
-- master5.vhd - entity/architecture pair
------------------------------------------------------------------------------
-- IMPORTANT:
-- DO NOT MODIFY THIS FILE EXCEPT IN THE DESIGNATED SECTIONS.
--
-- SEARCH FOR --USER TO DETERMINE WHERE CHANGES ARE ALLOWED.
--
-- TYPICALLY, THE ONLY ACCEPTABLE CHANGES INVOLVE ADDING NEW
-- PORTS AND GENERICS THAT GET PASSED THROUGH TO THE INSTANTIATION
-- OF THE USER_LOGIC ENTITY.
------------------------------------------------------------------------------
--
-- ***************************************************************************
-- ** Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.            **
-- **                                                                       **
-- ** Xilinx, Inc.                                                          **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"         **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
-- ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
-- ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE               **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR        **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF       **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       **
-- ** FOR A PARTICULAR PURPOSE.                                             **
-- **                                                                       **
-- ***************************************************************************
--
------------------------------------------------------------------------------
-- Filename:          master5.vhd
-- Version:           1.00.a
-- Description:       Top level design, instantiates library components and user logic.
-- Date:              Sat Nov 08 18:07:31 2008 (by Create and Import Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------
-- Naming Conventions:
--   active low signals:                    "*_n"
--   clock signals:                         "clk", "clk_div#", "clk_#x"
--   reset signals:                         "rst", "rst_n"
--   generics:                              "C_*"
--   user defined types:                    "*_TYPE"
--   state machine next state:              "*_ns"
--   state machine current state:           "*_cs"
--   combinatorial signals:                 "*_com"
--   pipelined or register delay signals:   "*_d#"
--   counter signals:                       "*cnt*"
--   clock enable signals:                  "*_ce"
--   internal version of output port:       "*_i"
--   device pins:                           "*_pin"
--   ports:                                 "- Names begin with Uppercase"
--   processes:                             "*_PROCESS"
--   component instantiations:              "<ENTITY_>I_<#|FUNC>"
------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
use proc_common_v2_00_a.ipif_pkg.all;

library plbv46_slave_single_v1_00_a;
```

```
use plbv46_slave_single_v1_00_a.plbv46_slave_single;

------------------------------------------------------------------------------
-- Entity section
------------------------------------------------------------------------------
-- Definition of Generics:
--   C_BASEADDR                   -- PLBv46 slave: base address
--   C_HIGHADDR                   -- PLBv46 slave: high address
--   C_SPLB_AWIDTH                -- PLBv46 slave: address bus width
--   C_SPLB_DWIDTH                -- PLBv46 slave: data bus width
--   C_SPLB_NUM_MASTERS           -- PLBv46 slave: Number of masters
--   C_SPLB_MID_WIDTH             -- PLBv46 slave: master ID bus width
--   C_SPLB_NATIVE_DWIDTH         -- PLBv46 slave: internal native data bus width
--   C_SPLB_P2P                   -- PLBv46 slave: point to point interconnect scheme
--   C_SPLB_SUPPORT_BURSTS        -- PLBv46 slave: support bursts
--   C_SPLB_SMALLEST_MASTER       -- PLBv46 slave: width of the smallest master
--   C_SPLB_CLK_PERIOD_PS         -- PLBv46 slave: bus clock in picoseconds
--   C_FAMILY                     -- Xilinx FPGA family
--
-- Definition of Ports:
--   SPLB_Clk                     -- PLB main bus clock
--   SPLB_Rst                     -- PLB main bus reset
--   PLB_ABus                     -- PLB address bus
--   PLB_UABus                    -- PLB upper address bus
--   PLB_PAValid                  -- PLB primary address valid indicator
--   PLB_SAValid                  -- PLB secondary address valid indicator
--   PLB_rdPrim                   -- PLB secondary to primary read request indicator
--   PLB_wrPrim                   -- PLB secondary to primary write request indicator
--   PLB_masterID                 -- PLB current master identifier
--   PLB_abort                    -- PLB abort request indicator
--   PLB_busLock                  -- PLB bus lock
--   PLB_RNW                      -- PLB read/not write
--   PLB_BE                       -- PLB byte enables
--   PLB_MSize                    -- PLB master data bus size
--   PLB_size                     -- PLB transfer size
--   PLB_type                     -- PLB transfer type
--   PLB_lockErr                  -- PLB lock error indicator
--   PLB_wrDBus                   -- PLB write data bus
--   PLB_wrBurst                  -- PLB burst write transfer indicator
--   PLB_rdBurst                  -- PLB burst read transfer indicator
--   PLB_wrPendReq                -- PLB write pending bus request indicator
--   PLB_rdPendReq                -- PLB read pending bus request indicator
--   PLB_wrPendPri                -- PLB write pending request priority
--   PLB_rdPendPri                -- PLB read pending request priority
--   PLB_reqPri                   -- PLB current request priority
--   PLB_TAttribute               -- PLB transfer attribute
--   Sl_addrAck                   -- Slave address acknowledge
--   Sl_SSize                     -- Slave data bus size
--   Sl_wait                      -- Slave wait indicator
--   Sl_rearbitrate               -- Slave re-arbitrate bus indicator
--   Sl_wrDAck                    -- Slave write data acknowledge
--   Sl_wrComp                    -- Slave write transfer complete indicator
--   Sl_wrBTerm                   -- Slave terminate write burst transfer
--   Sl_rdDBus                    -- Slave read data bus
--   Sl_rdWdAddr                  -- Slave read word address
--   Sl_rdDAck                    -- Slave read data acknowledge
--   Sl_rdComp                    -- Slave read transfer complete indicator
--   Sl_rdBTerm                   -- Slave terminate read burst transfer
--   Sl_MBusy                     -- Slave busy indicator
--   Sl_MWrErr                    -- Slave write error indicator
--   Sl_MRdErr                    -- Slave read error indicator
--   Sl_MIRQ                      -- Slave interrupt indicator
------------------------------------------------------------------------------

entity master5 is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE ---------------
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE ---------------

    -- DO NOT EDIT BELOW THIS LINE ---------------------
    -- Bus protocol parameters, do not add to or delete
```

```
  C_BASEADDR                     : std_logic_vector     := X"FFFFFFFF";
  C_HIGHADDR                     : std_logic_vector     := X"00000000";
  C_SPLB_AWIDTH                  : integer              := 32;
  C_SPLB_DWIDTH                  : integer              := 128;
  C_SPLB_NUM_MASTERS             : integer              := 8;
  C_SPLB_MID_WIDTH               : integer              := 3;
  C_SPLB_NATIVE_DWIDTH           : integer              := 32;
  C_SPLB_P2P                     : integer              := 0;
  C_SPLB_SUPPORT_BURSTS          : integer              := 0;
  C_SPLB_SMALLEST_MASTER         : integer              := 32;
  C_SPLB_CLK_PERIOD_PS           : integer              := 10000;
  C_FAMILY                       : string               := "virtex5"
  -- DO NOT EDIT ABOVE THIS LINE ----------------------
);
port
(
  -- ADD USER PORTS BELOW THIS LINE ------------------
     rste: in  std_logic;
     WRe: in  std_logic;
     RDe: in  std_logic;
     sdaine: in  std_logic;
     scline: in  std_logic;
     addresse: in  std_logic_vector(0 to 1);
     Ext_data_ine: in  std_logic_vector(0 to 7);
     INTe: out std_logic;
     Tx_nRxe: out std_logic;
     sdaoute: out std_logic;
     scloute: out std_logic;
     sdateste: out std_logic;
     teste: out std_logic_vector(0 to 31);
     --USER ports added here
  -- ADD USER PORTS ABOVE THIS LINE ------------------

  -- DO NOT EDIT BELOW THIS LINE ----------------------
  -- Bus protocol ports, do not add to or delete
  SPLB_Clk                       : in  std_logic;
  SPLB_Rst                       : in  std_logic;
  PLB_ABus                       : in  std_logic_vector(0 to 31);
  PLB_UABus                      : in  std_logic_vector(0 to 31);
  PLB_PAValid                    : in  std_logic;
  PLB_SAValid                    : in  std_logic;
  PLB_rdPrim                     : in  std_logic;
  PLB_wrPrim                     : in  std_logic;
  PLB_masterID                   : in  std_logic_vector(0 to C_SPLB_MID_WIDTH-1);
  PLB_abort                      : in  std_logic;
  PLB_busLock                    : in  std_logic;
  PLB_RNW                        : in  std_logic;
  PLB_BE                         : in  std_logic_vector(0 to C_SPLB_DWIDTH/8-1);
  PLB_MSize                      : in  std_logic_vector(0 to 1);
  PLB_size                       : in  std_logic_vector(0 to 3);
  PLB_type                       : in  std_logic_vector(0 to 2);
  PLB_lockErr                    : in  std_logic;
  PLB_wrDBus                     : in  std_logic_vector(0 to C_SPLB_DWIDTH-1);
  PLB_wrBurst                    : in  std_logic;
  PLB_rdBurst                    : in  std_logic;
  PLB_wrPendReq                  : in  std_logic;
  PLB_rdPendReq                  : in  std_logic;
  PLB_wrPendPri                  : in  std_logic_vector(0 to 1);
  PLB_rdPendPri                  : in  std_logic_vector(0 to 1);
  PLB_reqPri                     : in  std_logic_vector(0 to 1);
  PLB_TAttribute                 : in  std_logic_vector(0 to 15);
  Sl_addrAck                     : out std_logic;
  Sl_SSize                       : out std_logic_vector(0 to 1);
  Sl_wait                        : out std_logic;
  Sl_rearbitrate                 : out std_logic;
  Sl_wrDAck                      : out std_logic;
  Sl_wrComp                      : out std_logic;
  Sl_wrBTerm                     : out std_logic;
  Sl_rdDBus                      : out std_logic_vector(0 to C_SPLB_DWIDTH-1);
  Sl_rdWdAddr                    : out std_logic_vector(0 to 3);
  Sl_rdDAck                      : out std_logic;
  Sl_rdComp                      : out std_logic;
  Sl_rdBTerm                     : out std_logic;
```

```
    Sl_MBusy                       : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MWrErr                      : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MRdErr                      : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1);
    Sl_MIRQ                        : out std_logic_vector(0 to C_SPLB_NUM_MASTERS-1)
    -- DO NOT EDIT ABOVE THIS LINE --------------------
  );

  attribute SIGIS : string;
  attribute SIGIS of SPLB_Clk      : signal is "CLK";
  attribute SIGIS of SPLB_Rst      : signal is "RST";

end entity master5;

------------------------------------------------------------------------------
-- Architecture section
------------------------------------------------------------------------------

architecture IMP of master5 is

  ---------------------------------------
  -- Array of base/high address pairs for each address range
  ---------------------------------------
  constant ZERO_ADDR_PAD                 : std_logic_vector(0 to 31) := (others => '0');
  constant USER_SLV_BASEADDR             : std_logic_vector     := C_BASEADDR;
  constant USER_SLV_HIGHADDR             : std_logic_vector     := C_HIGHADDR;

  constant IPIF_ARD_ADDR_RANGE_ARRAY     : SLV64_ARRAY_TYPE     :=
    (
      ZERO_ADDR_PAD & USER_SLV_BASEADDR,  -- user logic slave space base address
      ZERO_ADDR_PAD & USER_SLV_HIGHADDR   -- user logic slave space high address
    );

  ---------------------------------------
  -- Array of desired number of chip enables for each address range
  ---------------------------------------
  constant USER_SLV_NUM_REG              : integer              := 16;
  constant USER_NUM_REG                  : integer              := USER_SLV_NUM_REG;

  constant IPIF_ARD_NUM_CE_ARRAY         : INTEGER_ARRAY_TYPE   :=
    (
      0  => pad_power2(USER_SLV_NUM_REG)  -- number of ce for user logic slave space
    );

  ---------------------------------------
  -- Ratio of bus clock to core clock (for use in dual clock systems)
  -- 1 = ratio is 1:1
  -- 2 = ratio is 2:1
  ---------------------------------------
  constant IPIF_BUS2CORE_CLK_RATIO       : integer              := 1;

  ---------------------------------------
  -- Width of the slave data bus (32 only)
  ---------------------------------------
  constant USER_SLV_DWIDTH               : integer              := C_SPLB_NATIVE_DWIDTH;

  constant IPIF_SLV_DWIDTH               : integer              := C_SPLB_NATIVE_DWIDTH;

  ---------------------------------------
  -- Index for CS/CE
  ---------------------------------------
  constant USER_SLV_CS_INDEX             : integer              := 0;
  constant USER_SLV_CE_INDEX             : integer              :=
calc_start_ce_index(IPIF_ARD_NUM_CE_ARRAY, USER_SLV_CS_INDEX);

  constant USER_CE_INDEX                 : integer              := USER_SLV_CE_INDEX;

  ---------------------------------------
  -- IP Interconnect (IPIC) signal declarations
  ---------------------------------------
  signal ipif_Bus2IP_Clk                : std_logic;
  signal ipif_Bus2IP_Reset              : std_logic;
  signal ipif_IP2Bus_Data               : std_logic_vector(0 to IPIF_SLV_DWIDTH-1);
  signal ipif_IP2Bus_WrAck              : std_logic;
```

```
  signal ipif_IP2Bus_RdAck             : std_logic;
  signal ipif_IP2Bus_Error             : std_logic;
  signal ipif_Bus2IP_Addr              : std_logic_vector(0 to C_SPLB_AWIDTH-1);
  signal ipif_Bus2IP_Data              : std_logic_vector(0 to IPIF_SLV_DWIDTH-1);
  signal ipif_Bus2IP_RNW               : std_logic;
  signal ipif_Bus2IP_BE                : std_logic_vector(0 to IPIF_SLV_DWIDTH/8-1);
  signal ipif_Bus2IP_CS                : std_logic_vector(0 to
((IPIF_ARD_ADDR_RANGE_ARRAY'length)/2)-1);
  signal ipif_Bus2IP_RdCE              : std_logic_vector(0 to
calc_num_ce(IPIF_ARD_NUM_CE_ARRAY)-1);
  signal ipif_Bus2IP_WrCE              : std_logic_vector(0 to
calc_num_ce(IPIF_ARD_NUM_CE_ARRAY)-1);
  signal user_Bus2IP_RdCE              : std_logic_vector(0 to USER_NUM_REG-1);
  signal user_Bus2IP_WrCE              : std_logic_vector(0 to USER_NUM_REG-1);
  signal user_IP2Bus_Data              : std_logic_vector(0 to USER_SLV_DWIDTH-1);
  signal user_IP2Bus_RdAck             : std_logic;
  signal user_IP2Bus_WrAck             : std_logic;
  signal user_IP2Bus_Error             : std_logic;

  ---------------------------------------
  -- Component declaration for verilog user logic
  ---------------------------------------
  component user_logic is
    generic
    (
      -- ADD USER GENERICS BELOW THIS LINE ---------------
      --USER generics added here
      -- ADD USER GENERICS ABOVE THIS LINE ---------------

      -- DO NOT EDIT BELOW THIS LINE ---------------------
      -- Bus protocol parameters, do not add to or delete
      C_SLV_DWIDTH                   : integer              := 32;
      C_NUM_REG                      : integer              := 16
      -- DO NOT EDIT ABOVE THIS LINE ---------------------
    );
    port
    (
      -- ADD USER PORTS BELOW THIS LINE ------------------
       rst: in  std_logic;
       WR: in  std_logic;
       RD: in  std_logic;
       sdain: in  std_logic;
       sclin: in  std_logic;
       address: in  std_logic_vector(0 to 1);
       Ext_data_in: in  std_logic_vector(0 to 7);
       INT: out std_logic;
       Tx_nRx: out std_logic;
       sdaout: out std_logic;
       sclout: out std_logic;
       sdatest: out std_logic;
       test: out std_logic_vector(0 to 31);
      --USER ports added here
      -- ADD USER PORTS ABOVE THIS LINE ------------------

      -- DO NOT EDIT BELOW THIS LINE ---------------------
      -- Bus protocol ports, do not add to or delete
      Bus2IP_Clk                     : in  std_logic;
      Bus2IP_Reset                   : in  std_logic;
      Bus2IP_Data                    : in  std_logic_vector(0 to C_SLV_DWIDTH-1);
      Bus2IP_BE                      : in  std_logic_vector(0 to C_SLV_DWIDTH/8-1);
      Bus2IP_RdCE                    : in  std_logic_vector(0 to C_NUM_REG-1);
      Bus2IP_WrCE                    : in  std_logic_vector(0 to C_NUM_REG-1);
      IP2Bus_Data                    : out std_logic_vector(0 to C_SLV_DWIDTH-1);
      IP2Bus_RdAck                   : out std_logic;
      IP2Bus_WrAck                   : out std_logic;
      IP2Bus_Error                   : out std_logic
      -- DO NOT EDIT ABOVE THIS LINE ---------------------
    );
  end component user_logic;

begin

  ---------------------------------------
```

```
-- instantiate plbv46_slave_single
------------------------------------------
PLBV46_SLAVE_SINGLE_I : entity plbv46_slave_single_v1_00_a.plbv46_slave_single
  generic map
  (
    C_ARD_ADDR_RANGE_ARRAY         => IPIF_ARD_ADDR_RANGE_ARRAY,
    C_ARD_NUM_CE_ARRAY             => IPIF_ARD_NUM_CE_ARRAY,
    C_SPLB_P2P                     => C_SPLB_P2P,
    C_BUS2CORE_CLK_RATIO           => IPIF_BUS2CORE_CLK_RATIO,
    C_SPLB_MID_WIDTH               => C_SPLB_MID_WIDTH,
    C_SPLB_NUM_MASTERS             => C_SPLB_NUM_MASTERS,
    C_SPLB_AWIDTH                  => C_SPLB_AWIDTH,
    C_SPLB_DWIDTH                  => C_SPLB_DWIDTH,
    C_SIPIF_DWIDTH                 => IPIF_SLV_DWIDTH,
    C_FAMILY                       => C_FAMILY
  )
  port map
  (
    SPLB_Clk                       => SPLB_Clk,
    SPLB_Rst                       => SPLB_Rst,
    PLB_ABus                       => PLB_ABus,
    PLB_UABus                      => PLB_UABus,
    PLB_PAValid                    => PLB_PAValid,
    PLB_SAValid                    => PLB_SAValid,
    PLB_rdPrim                     => PLB_rdPrim,
    PLB_wrPrim                     => PLB_wrPrim,
    PLB_masterID                   => PLB_masterID,
    PLB_abort                      => PLB_abort,
    PLB_busLock                    => PLB_busLock,
    PLB_RNW                        => PLB_RNW,
    PLB_BE                         => PLB_BE,
    PLB_MSize                      => PLB_MSize,
    PLB_size                       => PLB_size,
    PLB_type                       => PLB_type,
    PLB_lockErr                    => PLB_lockErr,
    PLB_wrDBus                     => PLB_wrDBus,
    PLB_wrBurst                    => PLB_wrBurst,
    PLB_rdBurst                    => PLB_rdBurst,
    PLB_wrPendReq                  => PLB_wrPendReq,
    PLB_rdPendReq                  => PLB_rdPendReq,
    PLB_wrPendPri                  => PLB_wrPendPri,
    PLB_rdPendPri                  => PLB_rdPendPri,
    PLB_reqPri                     => PLB_reqPri,
    PLB_TAttribute                 => PLB_TAttribute,
    Sl_addrAck                     => Sl_addrAck,
    Sl_SSize                       => Sl_SSize,
    Sl_wait                        => Sl_wait,
    Sl_rearbitrate                 => Sl_rearbitrate,
    Sl_wrDAck                      => Sl_wrDAck,
    Sl_wrComp                      => Sl_wrComp,
    Sl_wrBTerm                     => Sl_wrBTerm,
    Sl_rdDBus                      => Sl_rdDBus,
    Sl_rdWdAddr                    => Sl_rdWdAddr,
    Sl_rdDAck                      => Sl_rdDAck,
    Sl_rdComp                      => Sl_rdComp,
    Sl_rdBTerm                     => Sl_rdBTerm,
    Sl_MBusy                       => Sl_MBusy,
    Sl_MWrErr                      => Sl_MWrErr,
    Sl_MRdErr                      => Sl_MRdErr,
    Sl_MIRQ                        => Sl_MIRQ,
    Bus2IP_Clk                     => ipif_Bus2IP_Clk,
    Bus2IP_Reset                   => ipif_Bus2IP_Reset,
    IP2Bus_Data                    => ipif_IP2Bus_Data,
    IP2Bus_WrAck                   => ipif_IP2Bus_WrAck,
    IP2Bus_RdAck                   => ipif_IP2Bus_RdAck,
    IP2Bus_Error                   => ipif_IP2Bus_Error,
    Bus2IP_Addr                    => ipif_Bus2IP_Addr,
    Bus2IP_Data                    => ipif_Bus2IP_Data,
    Bus2IP_RNW                     => ipif_Bus2IP_RNW,
    Bus2IP_BE                      => ipif_Bus2IP_BE,
    Bus2IP_CS                      => ipif_Bus2IP_CS,
    Bus2IP_RdCE                    => ipif_Bus2IP_RdCE,
    Bus2IP_WrCE                    => ipif_Bus2IP_WrCE
```

```
    );

  ---------------------------------------
  -- instantiate User Logic
  ---------------------------------------
  USER_LOGIC_I : component user_logic
    generic map
    (
      -- MAP USER GENERICS BELOW THIS LINE ---------------
      --USER generics mapped here
      -- MAP USER GENERICS ABOVE THIS LINE ---------------

      C_SLV_DWIDTH                   => USER_SLV_DWIDTH,
      C_NUM_REG                      => USER_NUM_REG
    )
    port map
    (
      -- MAP USER PORTS BELOW THIS LINE ------------------
       rst => rste,
       WR => WRe,
       RD => RDe,
       sdain => sdaine,
       sclin => scline,
       address => addresse,
       Ext_data_in => Ext_data_ine,
       INT => INTe,
       Tx_nRx => Tx_nRxe,
       sdaout => sdaoute,
       sclout => scloute,
       sdatest => sdateste,
       test => teste,

      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------

      Bus2IP_Clk                     => ipif_Bus2IP_Clk,
      Bus2IP_Reset                   => ipif_Bus2IP_Reset,
      Bus2IP_Data                    => ipif_Bus2IP_Data,
      Bus2IP_BE                      => ipif_Bus2IP_BE,
      Bus2IP_RdCE                    => user_Bus2IP_RdCE,
      Bus2IP_WrCE                    => user_Bus2IP_WrCE,
      IP2Bus_Data                    => user_IP2Bus_Data,
      IP2Bus_RdAck                   => user_IP2Bus_RdAck,
      IP2Bus_WrAck                   => user_IP2Bus_WrAck,
      IP2Bus_Error                   => user_IP2Bus_Error
    );

  ---------------------------------------
  -- connect internal signals
  ---------------------------------------
  ipif_IP2Bus_Data <= user_IP2Bus_Data;
  ipif_IP2Bus_WrAck <= user_IP2Bus_WrAck;
  ipif_IP2Bus_RdAck <= user_IP2Bus_RdAck;
  ipif_IP2Bus_Error <= user_IP2Bus_Error;

  user_Bus2IP_RdCE <= ipif_Bus2IP_RdCE(USER_CE_INDEX to USER_CE_INDEX+USER_NUM_REG-1);
  user_Bus2IP_WrCE <= ipif_Bus2IP_WrCE(USER_CE_INDEX to USER_CE_INDEX+USER_NUM_REG-1);

end IMP;
```

I2C Master Controller:

```
//----------------------------------------------------------------------------
// user_logic.vhd - module
//----------------------------------------------------------------------------
//
// ************************************************************************
// ** Copyright (c) 1995-2007 Xilinx, Inc.  All rights reserved.           **
// **                                                                      **
// ** Xilinx, Inc.                                                        **
// ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"        **
```

```
// ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND       **
// ** SOLUTIONS FOR XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE,        **
// ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,        **
// ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION           **
// ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,     **
// ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE      **
// ** FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY              **
// ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE               **
// ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR        **
// ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF       **
// ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS       **
// ** FOR A PARTICULAR PURPOSE.                                             **
// **                                                                       **
// ***************************************************************************
//
//----------------------------------------------------------------------------
// Filename:          user_logic.vhd
// Version:           1.00.a
// Description:       User logic module.
// Date:              Sat Nov 08 18:07:31 2008 (by Create and Import Peripheral Wizard)
// Verilog Standard:  Verilog-2001
//----------------------------------------------------------------------------
// Naming Conventions:
//   active low signals:                    "*_n"
//   clock signals:                         "clk", "clk_div#", "clk_#x"
//   reset signals:                         "rst", "rst_n"
//   generics:                              "C_*"
//   user defined types:                    "*_TYPE"
//   state machine next state:              "*_ns"
//   state machine current state:           "*_cs"
//   combinatorial signals:                 "*_com"
//   pipelined or register delay signals:   "*_d#"
//   counter signals:                       "*cnt*"
//   clock enable signals:                  "*_ce"
//   internal version of output port:       "*_i"
//   device pins:                           "*_pin"
//   ports:                                 "- Names begin with Uppercase"
//   processes:                             "*_PROCESS"
//   component instantiations:              "<ENTITY_>I_<#|FUNC>"
//----------------------------------------------------------------------------

module user_logic
(
  // -- ADD USER PORTS BELOW THIS LINE ---------------
  address,
  WR,
  RD,
  test,
  sdatest,
  sclin,
  sdain,
  sclout,
  sdaout,
  ext_data_in,
  rst,
  INT,
  Tx_nRx,
  // --USER ports added here
  // -- ADD USER PORTS ABOVE THIS LINE ---------------

  // -- DO NOT EDIT BELOW THIS LINE ------------------
  // -- Bus protocol ports, do not add to or delete
  Bus2IP_Clk,                     // Bus to IP clock
  Bus2IP_Reset,                   // Bus to IP reset
  Bus2IP_Data,                    // Bus to IP data bus
  Bus2IP_BE,                      // Bus to IP byte enables
  Bus2IP_RdCE,                    // Bus to IP read chip enable
  Bus2IP_WrCE,                    // Bus to IP write chip enable
  IP2Bus_Data,                    // IP to Bus data bus
  IP2Bus_RdAck,                   // IP to Bus read transfer acknowledgement
  IP2Bus_WrAck,                   // IP to Bus write transfer acknowledgement
  IP2Bus_Error                    // IP to Bus error response
  // -- DO NOT EDIT ABOVE THIS LINE ------------------
```

```
); // user_logic

// -- ADD USER PARAMETERS BELOW THIS LINE ------------
// --USER parameters added here
// -- ADD USER PARAMETERS ABOVE THIS LINE ------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol parameters, do not add to or delete
parameter C_SLV_DWIDTH                   = 32;
parameter C_NUM_REG                      = 16;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

// -- ADD USER PORTS BELOW THIS LINE -----------------
       input WR,RD,rst,sdain,sclin;
       input   [0:1]   address;

       input  [0:7]  ext_data_in;
       output INT, Tx_nRx, sdaout,sclout,sdatest;
       output [0:31] test;
  wire          [31:0]
temp1,temp,No82,No83,No84,No85;
  wire          [7:0]                                                    ext_data_out;
  wire          [7:0]                                                    NoOfLoc;
  wire                                      NoOfBytes_ready,cs;
  wire                                      temp3,resend,P;
       wire    [7:0] data_out,buf_out;
       wire    beginT2,TxRx,bit8_1;
// --USER ports added here
// -- ADD USER PORTS ABOVE THIS LINE -----------------

// -- DO NOT EDIT BELOW THIS LINE --------------------
// -- Bus protocol ports, do not add to or delete
input                                     Bus2IP_Clk;
input                                     Bus2IP_Reset;
input      [0 : C_SLV_DWIDTH-1]           Bus2IP_Data;
input      [0 : C_SLV_DWIDTH/8-1]         Bus2IP_BE;
input      [0 : C_NUM_REG-1]              Bus2IP_RdCE;
input      [0 : C_NUM_REG-1]              Bus2IP_WrCE;
output     [0 : C_SLV_DWIDTH-1]           IP2Bus_Data;
output                                    IP2Bus_RdAck;
output                                    IP2Bus_WrAck;
output                                    IP2Bus_Error;
// -- DO NOT EDIT ABOVE THIS LINE --------------------

//---------------------------------------------------------------------------
// Implementation
//---------------------------------------------------------------------------

  // --USER nets declarations added here, as needed for user logic

  // Nets for user logic slave model s/w accessible register example
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg0;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg1;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg2;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg3;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg4;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg5;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg6;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg7;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg8;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg9;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg10;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg11;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg12;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg13;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg14;
  reg        [0 : C_SLV_DWIDTH-1]           slv_reg15;
  wire       [0 : 15]                       slv_reg_write_sel;
  wire       [0 : 15]                       slv_reg_read_sel;
  reg        [0 : C_SLV_DWIDTH-1]           slv_ip2bus_data;
  wire                                      slv_read_ack;
  wire                                      slv_write_ack;
  integer                                   byte_index, bit_index;
```

```
  // --USER logic implementation added here

  // ------------------------------------------------------
  // Example code to read/write user logic slave model s/w accessible registers
  //
  // Note:
  // The example code presented here is to show you one way of reading/writing
  // software accessible registers implemented in the user logic slave model.
  // Each bit of the Bus2IP_WrCE/Bus2IP_RdCE signals is configured to correspond
  // to one software accessible register by the top level template. For example,
  // if you have four 32 bit software accessible registers in the user logic,
  // you are basically operating on the following memory mapped registers:
  //
  //    Bus2IP_WrCE/Bus2IP_RdCE   Memory Mapped Register
  //                      "1000"   C_BASEADDR + 0x0
  //                      "0100"   C_BASEADDR + 0x4
  //                      "0010"   C_BASEADDR + 0x8
  //                      "0001"   C_BASEADDR + 0xC
  //
  // ------------------------------------------------------

  assign
    slv_reg_write_sel = Bus2IP_WrCE[0:15],
    slv_reg_read_sel  = Bus2IP_RdCE[0:15],
    slv_write_ack     = Bus2IP_WrCE[0] || Bus2IP_WrCE[1] || Bus2IP_WrCE[2] || Bus2IP_WrCE[3]
|| Bus2IP_WrCE[4] || Bus2IP_WrCE[5] || Bus2IP_WrCE[6] || Bus2IP_WrCE[7] || Bus2IP_WrCE[8] ||
Bus2IP_WrCE[9] || Bus2IP_WrCE[10] || Bus2IP_WrCE[11] || Bus2IP_WrCE[12] || Bus2IP_WrCE[13] ||
Bus2IP_WrCE[14] || Bus2IP_WrCE[15],
    slv_read_ack      = Bus2IP_RdCE[0] || Bus2IP_RdCE[1] || Bus2IP_RdCE[2] || Bus2IP_RdCE[3]
|| Bus2IP_RdCE[4] || Bus2IP_RdCE[5] || Bus2IP_RdCE[6] || Bus2IP_RdCE[7] || Bus2IP_RdCE[8] ||
Bus2IP_RdCE[9] || Bus2IP_RdCE[10] || Bus2IP_RdCE[11] || Bus2IP_RdCE[12] || Bus2IP_RdCE[13] ||
Bus2IP_RdCE[14] || Bus2IP_RdCE[15];

  // implement slave model register(s)
  always @( posedge Bus2IP_Clk )
    begin: SLAVE_REG_WRITE_PROC
                      for(bit_index = 0;bit_index <= 31;bit_index = bit_index + 1)
                              slv_reg2[bit_index] <= ~P;

         slv_reg3[31] <= NoOfLoc[0];//slv_reg3 is for reading data from userlogic, used for
NoOfLoc
         slv_reg3[30] <= NoOfLoc[1];
         slv_reg3[29] <= NoOfLoc[2];
         slv_reg3[28] <= NoOfLoc[3];
         slv_reg3[27] <= NoOfLoc[4];
         slv_reg3[26] <= NoOfLoc[5];
         slv_reg3[25] <= NoOfLoc[6];
         slv_reg3[24] <= NoOfLoc[7];
         slv_reg3[0:23] <= 24'h000000;
      if ( Bus2IP_Reset == 1 )
        begin
          slv_reg0 <= 0;
          slv_reg1 <= 0;
          slv_reg2 <= 0;
          slv_reg3 <= 0;
          slv_reg4 <= 0;
          slv_reg5 <= 0;
          slv_reg6 <= 0;
          slv_reg7 <= 0;
          slv_reg8 <= 0;
          slv_reg9 <= 0;
          slv_reg10 <= 0;
          slv_reg11 <= 0;
          slv_reg12 <= 0;
          slv_reg13 <= 0;
          slv_reg14 <= 0;
          slv_reg15 <= 0;
        end
      else
        case ( slv_reg_write_sel )
          16'b1000000000000000 :
```

```
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg0[bit_index] <= Bus2IP_Data[bit_index];
          16'b0100000000000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg1[bit_index] <= Bus2IP_Data[bit_index];
          16'b0010000000000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg2[bit_index] <= Bus2IP_Data[bit_index];
          16'b0001000000000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg3[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000100000000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg4[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000010000000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg5[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000001000000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg6[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000100000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg7[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000010000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg8[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000001000000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg9[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000000100000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
```

```
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg10[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000000010000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg11[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000000001000 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg12[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000000000100 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg13[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000000000010 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg14[bit_index] <= Bus2IP_Data[bit_index];
          16'b0000000000000001 :
            for ( byte_index = 0; byte_index <= (C_SLV_DWIDTH/8)-1; byte_index = byte_index+1
)
              if ( Bus2IP_BE[byte_index] == 1 )
                for ( bit_index = byte_index*8; bit_index <= byte_index*8+7; bit_index =
bit_index+1 )
                  slv_reg15[bit_index] <= Bus2IP_Data[bit_index];
          default : ;
        endcase

    end // SLAVE_REG_WRITE_PROC

  // implement slave model register read mux
  always @( slv_reg_read_sel or slv_reg0 or slv_reg1 or slv_reg2 or slv_reg3 or slv_reg4 or
slv_reg5 or slv_reg6 or slv_reg7 or slv_reg8 or slv_reg9 or slv_reg10 or slv_reg11 or
slv_reg12 or slv_reg13 or slv_reg14 or slv_reg15 )
    begin: SLAVE_REG_READ_PROC

      case ( slv_reg_read_sel )
        16'b1000000000000000 : slv_ip2bus_data <= slv_reg0;
        16'b0100000000000000 : slv_ip2bus_data <= slv_reg1;
        16'b0010000000000000 : slv_ip2bus_data <= slv_reg2;
        16'b0001000000000000 : slv_ip2bus_data <= slv_reg3;
        16'b0000100000000000 :
                               begin
                                      slv_ip2bus_data[0:23] <= 24'h000000;//if read slv4 ->
send buffer data to MB.
                                      slv_ip2bus_data[24:31] <= buf_out;
                               end
        16'b0000010000000000 : slv_ip2bus_data <= slv_reg5;
        16'b0000001000000000 : slv_ip2bus_data <= slv_reg6;
        16'b0000000100000000 : slv_ip2bus_data <= slv_reg7;
        16'b0000000010000000 : slv_ip2bus_data <= slv_reg8;
        16'b0000000001000000 : slv_ip2bus_data <= slv_reg9;
        16'b0000000000100000 : slv_ip2bus_data <= slv_reg10;
        16'b0000000000010000 : slv_ip2bus_data <= slv_reg11;
        16'b0000000000001000 : slv_ip2bus_data <= slv_reg12;
        16'b0000000000000100 : slv_ip2bus_data <= slv_reg13;
        16'b0000000000000010 : slv_ip2bus_data <= slv_reg14;
        16'b0000000000000001 : slv_ip2bus_data <= slv_reg15;
        default : slv_ip2bus_data <= 0;
      endcase
```

```
    end // SLAVE_REG_READ_PROC
//------------------Custom Components ----------------------------------------------------
---------
        assign No82 = temp1;
        assign No83 = No82 + 1;
        assign No84 = No83 + 1;
        assign No85 = No84 + 1;
        assign resend = temp3;
//      Build a buffer 96x8: Write data from DR to buffer whenever there is a bit8_1 comes
//      buffer  buffer_96x8(DR_in,data_out,beginT2,TxRx,rst|P,clk,bit8_1);
//       buffer(out,data_in,load,RnW,rst,P,Pclk,rd_clk,clk);

        buffer
buffer_96x8(buf_out,data_out,beginT2|P,TxRx|P,rst,P,Bus2IP_Clk,slv_reg_read_sel[4],bit8_1);
//      slv_reg4 will be read out if read_sel is high, use this signal as the clock signal for
reading.
//      buf_out: output of buffer
//      data_out: input of buffer, taken from DR
//      beginT2|P: load signal. Just dont wanna load DR to buffer during first (and/or second
byte, which
//      are addresses. If stop, always load, we read out buffer.
//      TxRx|P: if started, this will take whatever value of TxRx, if Stop, this will take
value 1, which
//      is READ.

        // Call the master component
        I2C_Master
        M(data_out,beginT2,TxRx,bit8_1,cs,WR,address,resend,P,NoOfLoc,NoOfBytes_ready,No82,No8
3,No84,No85,test,sdatest,sclin,sdain,sclout,sdaout,ext_data_out,temp[7:0],ext_data_in,rst,INT
,Tx_nRx,Bus2IP_Clk);
  //I2C_Master
        (data_out,beginT2,TxRx,bit8_1,cs,WR,address,resend,P,NoOfLoc,NoOfBytes_ready,No82,No83
,No84,No85,test,sdatest,sclin,sdain,sclout,sdaout,ext_data_out,Start_Byte,ext_data_in,rst,INT
,Tx_nRx,fpga_clk);

        assign temp1[0] = slv_reg1[31];
        assign temp1[1] = slv_reg1[30];
        assign temp1[2] = slv_reg1[29];
        assign temp1[3] = slv_reg1[28];
        assign temp1[4] = slv_reg1[27];
        assign temp1[5] = slv_reg1[26];
        assign temp1[6] = slv_reg1[25];
        assign temp1[7] = slv_reg1[24];
        assign temp1[8] = slv_reg1[23];
        assign temp1[9] = slv_reg1[22];
        assign temp1[10] = slv_reg1[21];
        assign temp1[11] = slv_reg1[20];
        assign temp1[12] = slv_reg1[19];
        assign temp1[13] = slv_reg1[18];
        assign temp1[14] = slv_reg1[17];
        assign temp1[15] = slv_reg1[16];
        assign temp1[16] = slv_reg1[15];
        assign temp1[17] = slv_reg1[14];
        assign temp1[18] = slv_reg1[13];
        assign temp1[19] = slv_reg1[12];
        assign temp1[20] = slv_reg1[11];
        assign temp1[21] = slv_reg1[10];
        assign temp1[22] = slv_reg1[9];
        assign temp1[23] = slv_reg1[8];
        assign temp1[24] = slv_reg1[7];
        assign temp1[25] = slv_reg1[6];
        assign temp1[26] = slv_reg1[5];
        assign temp1[27] = slv_reg1[4];
        assign temp1[28] = slv_reg1[3];
        assign temp1[29] = slv_reg1[2];
        assign temp1[30] = slv_reg1[1];
        assign temp1[31] = slv_reg1[0];

        assign temp[0] = slv_reg0[31];//for start byte
        assign temp[1] = slv_reg0[30];
        assign temp[2] = slv_reg0[29];
        assign temp[3] = slv_reg0[28];
```

```
        assign temp[4] = slv_reg0[27];
        assign temp[5] = slv_reg0[26];
        assign temp[6] = slv_reg0[25];
        assign temp[7] = slv_reg0[24];

        assign temp[8] = slv_reg0[23];//Go bit
        assign temp[9] = slv_reg0[22];//chip select
        assign temp[10] = slv_reg0[21];
        assign temp[11] = slv_reg0[20];
        assign temp[12] = slv_reg0[19];
        assign temp[13] = slv_reg0[18];
        assign temp[14] = slv_reg0[17];
        assign temp[15] = slv_reg0[16];
        assign temp[16] = slv_reg0[15];
        assign NoOfBytes_ready = temp[16];

        assign temp3 = temp[8];//go bit
        assign cs = temp[9];
//---------------------------------End of Custom components--------------------------------
--------
  // ------------------------------------------------------------
  // Example code to drive IP to Bus signals
  // ------------------------------------------------------------

  assign IP2Bus_Data    = slv_ip2bus_data;
  assign IP2Bus_WrAck   = slv_write_ack;
  assign IP2Bus_RdAck   = slv_read_ack;
  assign IP2Bus_Error   = 0;

endmodule
```

APPENDIX D

C# Code for Serial Port Program

```
#region Namespace Inclusions
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Linq;
using System.IO.Ports;
using WindowsFormsApplication1.Properties;

#endregion
namespace WindowsFormsApplication1
{
    public enum LogMsgType { Incoming, Outgoing, Normal, Warning, Error };
    public partial class Form1 : Form
    {

        public int X, Y;

        private byte[] buffer = new byte[96];
        private byte[] buffer1 = new byte[96];
        private int[] ADCout = new int[32];
        private int nCount1,nCount = 0;
        private int test = 0;
        string testtext = "";



        private SerialPort serialPort1 = new SerialPort();
        public Form1()
        {

            InitializeComponent();
            InitializeControlValues();
            serialPort1.DataReceived += new
            SerialDataReceivedEventHandler(port_DataReceived);
            Rectangle r = new Rectangle(10, 15, 146, 152);


        }
        private Color[] LogMsgTypeColor = { Color.Blue, Color.Green, Color.Black,
Color.Orange, Color.Red };
        private void button1_Click(object sender, EventArgs e)
        {

            int i,k, j, l, n,o;



            //bit operation
            n = 0;
            for (i = 0; i < nCount1 / 3; i++)
            {
                k = (int)buffer[i * 3 + 1];//MSB of ADC output
                l = k;
                o = k & 32;
                j = (int)buffer[i * 3 + 2];
                if (o == 32)//if negative
                {
                    k = k ^ 255;//do 1st complement
                    k = k << 8;//shift left first 6 bits MSB.
                    j = j ^ 255;//do first complementary for 8-bits LSB.
```

```
                    k = k + j + 1;//do 2nd complement for 14-bits ADC output.
                    k = k & 16383;//16383 = 0011111111111111, truncate 16 bits to get 14 bits
value
                    k = -k;
                }
                else
                {
                    k = k << 8;
                    k = k + j;//add up 8-bits LSB
                    k = k & 16383;//16383 = 0011111111111111, truncate 16 bits to get 14 bits
value
                }

                ADCout[n] = k;
                n++;

            }
            DrawWaveform();
        }
        private void DrawWaveform()
        {
            int i;

            string dataText = "";

            for (i = 0; i < 32; i++)
            {

                //Displaying purpose only
                dataText += ADCout[i].ToString() + " ";
                //   stringText += Convert.ToChar(buffer[i]);



            }
            //Log(LogMsgType.Incoming, dataText);

            System.Drawing.Pen myPen;//Creat a new pen to draw
            myPen = new System.Drawing.Pen(System.Drawing.Color.Green);//select color for the
pen
            Size monitorSize = SystemInformation.PrimaryMonitorSize;//get monitor size.

            SolidBrush solidBrush = new SolidBrush(Color.White);//
            System.Drawing.Graphics formGraphics = this.CreateGraphics();
            //Graphics g = this.CreateGraphics();
            Rectangle r = new Rectangle(15, 60, 580, 260);
            Pen pen = new Pen(Color.Blue);
            formGraphics.FillRectangle(solidBrush, r);
            Pen pen2 = new Pen(Color.Black, 3);
            formGraphics.DrawRectangle(pen2, r);
            formGraphics.FillRectangle(solidBrush, r);
            formGraphics.DrawRectangle(pen2, r);

            myPen.DashStyle = System.Drawing.Drawing2D.DashStyle.Solid;
            myPen.Width = 2;
            // Declaring point1 and initializing x,y values
            Point point1 = new Point();

            // Declaring point2 without initializing x,y values
            Point point2 = new Point();
            formGraphics.DrawLine(myPen, 15,190,595,190);
            myPen = new System.Drawing.Pen(System.Drawing.Color.Red);

            for (i = 0; i < 31; i++)
            {
                point1.X = 15 + i * 15;
                point1.Y = 190 - ADCout[i]/ 100;
                point2.X = 15 + (i + 1) * 15;
                point2.Y = 190 - ADCout[i+1] / 100;
                formGraphics.DrawLine(myPen,point1 , point2 );

            }
            myPen.Dispose();
```

```
    formGraphics.Dispose();


}


private byte CharToByte(char c)
{
    if ((c >= '0') && (c <= '9'))
    {
        return (byte)(c - '0');
    }
    if ((c >= 'a') && (c <= 'f'))
    {
        return (byte)(c + 10 - 'a');
    }
    if ((c >= 'A') && (c <= 'F'))
    {
        return (byte)(c + 10 - 'A');
    }
    return 0;

}
private byte[] StringHexToBytes(string text, out bool isSuccess)
{
    byte[] ret = new byte[2];

    char c;


    ret[0] = 0;
    ret[1] = 0;

    int len = text.Length;
    if ((len < 3) || (len > 6) || (text[0] != '0') || (text[1] != 'x'))
    {
        isSuccess = false;
        return ret;
    }

    switch (len)
    {
        case 3:
            c = text[2];
            ret[1] = CharToByte(c);

            break;

        case 4:
            c = text[3];
            ret[1] = (byte)(CharToByte(c) * (byte)16 + CharToByte(text[3]));

            break;

        case 5:

            ret[1] = (byte)(CharToByte(text[3]) * 16 + CharToByte(text[4]));
            ret[0] = CharToByte(text[2]);

            break;

        case 6:
            ret[1] = (byte)(CharToByte(text[4]) * 16 + CharToByte(text[5]));
            ret[0] = (byte)(CharToByte(text[2]) * 16 + CharToByte(text[3]));
            break;

        default:
            isSuccess = false;
            return ret;
    }

    isSuccess = true;
    return ret;
```

```
        }


        private void Log(LogMsgType msgtype, string msg)
        {
            rtfTerminal.Invoke(new EventHandler(delegate
            {
                //rtfTerminal.SelectedText = string.Empty;
                //rtfTerminal.SelectionFont = new Font(rtfTerminal.SelectionFont,
FontStyle.Bold);
                //rtfTerminal.SelectionColor = LogMsgTypeColor[(int)msgtype];
                rtfTerminal.AppendText(msg);
                rtfTerminal.ScrollToCaret();
            }));
        }

        private void port_DataReceived(object sender, SerialDataReceivedEventArgs e)
        {
            string dataText = "";
            int i;
            test = test + 1;

            nCount = serialPort1.BytesToRead;

            ////byte[] buffer = new byte[nCount];
            //testtext = "nCount: " + test.ToString()+" ";
            //Log(LogMsgType.Incoming, testtext);

            serialPort1.Read(buffer1, 0, nCount);//if read a fix number --> get error while
running
            //because nCount is < 2048
            if(nCount != 0)
                for (i = 0; i < nCount; i++)
                    buffer[i+nCount1] = buffer1[i];

            nCount1 += nCount;
            //for (i = 0; i < nCount; i++)
            //{

            //    //Displaying purpose only
            //    dataText += buffer1[i].ToString() + " ";
            //    //   stringText += Convert.ToChar(buffer[i]);
            //}
            dataText += nCount1.ToString() + " ";
            Log(LogMsgType.Incoming, dataText);


        }


        private string ByteArrayToHexString(byte[] data)
        {
            StringBuilder sb = new StringBuilder(data.Length * 3);
            foreach (byte b in data)
                sb.Append(Convert.ToString(b, 16).PadLeft(2, '0').PadRight(3, ' '));
            return sb.ToString().ToUpper();
        }

        private void Form1_Load(object sender, EventArgs e)
        {

        }

        private void rtfTerminal_TextChanged(object sender, EventArgs e)
        {

        }

        private void label1_Click(object sender, EventArgs e)
        {

        }
```

```
        private void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            Close();
        }

        private void cmbPortName_SelectedIndexChanged(object sender, EventArgs e)
        {

        }


        private void label3_Click(object sender, EventArgs e)
        {

        }

        private void btnConnect_Click_1(object sender, EventArgs e)
        {
            // If the port is open, close it.
            if (serialPort1.IsOpen) serialPort1.Close();
            else
            {
                // Set the port's settings
                serialPort1.BaudRate = int.Parse(cmbBaudRate.Text);
                serialPort1.DataBits = int.Parse(cmbDataBits.Text);
                serialPort1.StopBits = (StopBits)Enum.Parse(typeof(StopBits),
cmbStopBits.Text);
                serialPort1.Parity = (Parity)Enum.Parse(typeof(Parity), cmbParity.Text);
                serialPort1.PortName = cmbPortName.Text;

                // Open the port
                serialPort1.Open();


            }
            // Change the state of the form's controls
            EnableControls();

            // If the port is open, send focus to the send data box
            if (serialPort1.IsOpen) rtfTerminal.Focus();



        }
        private void EnableControls()
        {
            // Enable/disable controls based on whether the port is open or not
            gbPortSettings.Enabled = !serialPort1.IsOpen;


            if (serialPort1.IsOpen) btnConnect.Text = "&Disconnect";
            else btnConnect.Text = "&Connect";
        }
        private void InitializeControlValues()
        {
          cmbParity.Items.Clear(); cmbParity.Items.AddRange(Enum.GetNames(typeof(Parity)));
          cmbStopBits.Items.Clear();
cmbStopBits.Items.AddRange(Enum.GetNames(typeof(StopBits)));

          cmbPortName.Items.Clear();
          foreach (string s in SerialPort.GetPortNames())
            cmbPortName.Items.Add(s);

        }
        private void SaveSettings()
        {
            Settings.Default.Save();
        }

        private void button1_Click_1(object sender, EventArgs e)
        {
            int i;
```

```
            byte[] slvAddr1 = new byte[16];
            rtfTerminal.Clear();
            nCount1 = 0;
            for (i = 0; i < 96; i++)
            {
                buffer[i] = 0;
            }
            for (i = 0; i < 32; i++)
            {
                ADCout[i] = 0;
            }
            slvAddr1[0] = 1;//address of slave 1.
            serialPort1.Write(slvAddr1, 0, 1);//Send address to the board

        }

        private void button2_Click_1(object sender, EventArgs e)
        {
            int i;
            byte[] slvAddr1 = new byte[16];
            rtfTerminal.Clear();
            nCount1 = 0;
            for (i = 0; i < 96; i++)
            {
                buffer[i] = 0;
            }
            for (i = 0; i < 32; i++)
            {
                ADCout[i] = 0;
            }
            slvAddr1[0] = 2;//address of slave 1.
            serialPort1.Write(slvAddr1, 0, 1);//Send address to the board
        }

        private void OnForm_RePaint(object sender, PaintEventArgs e)
        {


            DrawWaveform();
        }

        private void aboutToolStripMenuItem1_Click(object sender, EventArgs e)
        {
            Form frmAbout = new AboutBox1();


            frmAbout.ShowDialog();

        }
    }
}
```