Tutorial for I²C Serial Protocol

(original document written by Jon Valdez, Jared Becker at Texas Instruments)

The I²C bus is a very popular and powerful bus used for communication between a master (or multiple masters) and a single or multiple slave devices. Figure 1 illustrates how several different peripherals may share a bus which is connected to a processor through only 2 wires, which is one of the largest benefits that the I²C bus can give when compared to other interfaces. This tutorial is aimed at helping users understand how the I²C bus works. Figure 1 shows a typical I²C bus for an embedded system, where multiple slave devices are used. The microcontroller represents the I²C master, and controls the I/O expanders, various sensors, EEPROM, ADCs/DACs, and much more. All of which are controlled with only 2 pins from the master.

1. Electrical Characteristics

I²C uses an open-drain/open-collector with an input buffer on the same line, which allows a single data line to be used for bidirectional data flow.

1.1 Open-Drain for Bidirectional Communication

Open-drain refers to a type of output which can either pull the bus down to a voltage (ground, in most cases), or "release" the bus and let it be pulled up by a pull-up resistor. In the event of the bus being released by the master or a slave, the pull-up resistor (RPU) on the line is responsible for pulling the bus voltage up to the power rail. Since no device may force a high on a line, this means that the bus will never run into a communication issue where one device may try to transmit a high, and another transmits a low, causing a short (power rail to ground). I²C requires that if a master in a multi-master environment transmits a high, but see's that the line is low (another device is pulling it down), to halt communications because another device is using the bus. Push-pull interfaces do not allow for this type of freedom, which is a benefit of I²C.
Figure 2 shows a simplified view of the internal structure of the slave or master device on the SDA/SCL lines, consisting of a buffer to read input data, and a pull-down FET to transmit data. A device is only able to pull the bus line low (provide short to ground) or release the bus line (high impedance to ground) and allow the pull-up resistor to raise the voltage. This is an important concept to realize when dealing with I²C devices, since no device may hold the bus high. This property is what allows bidirectional communication to take place.

1.1.1 Open-Drain Pulling Low

As described in the previous section, the Open-Drain setup may only pull a bus low, or "release" it and let a resistor pull it high. Figure 3 shows the flow of current to pull the bus low. The logic wanting to transmit a low will activate the pull-down FET, which will provide a short to ground, pulling the line low.

1.1.2 Open-Drain Releasing Bus

When the slave or master wishes to transmit a logic high, it may only release the bus by turning off the pull-down FET. This leaves the bus floating, and the pull-up resistor will pull the voltage up to the voltage rail, which will be interpreted as a high. Figure 4 shows the flow of current through the pull-up resistor, which pulls the bus high.
2 I2C Interface

2.1 General I2C Operation

The I2C bus is a standard bidirectional interface that uses a controller, known as the master, to communicate with slave devices. A slave may not transmit data unless it has been addressed by the master. Each device on the I2C bus has a specific device address to differentiate between other devices that are on the same I2C bus. Many slave devices will require configuration upon startup to set the behavior of the device. This is typically done when the master accesses the slave's internal register maps, which have unique register addresses. A device can have one or multiple registers where data is stored, written, or read.

The physical I2C interface consists of the serial clock (SCL) and serial data (SDA) lines. Both SDA and SCL lines must be connected to VCC through a pull-up resistor. The size of the pull-up resistor is determined by the amount of capacitance on the I2C lines (for further details, refer to I2C Pull-up Resistor Calculation (SLVA689). Data transfer may be initiated only when the bus is idle. A bus is considered idle if both SDA and SCL lines are high after a STOP condition.

The general procedure for a master to access a slave device is the following:

1. Suppose a master wants to send data to a slave:
   - Master-transmitter sends a START condition and addresses the slave-receiver
   - Master-transmitter sends the requested register to write to slave transmitter
   - Master-transmitter sends data to slave-receiver
   - Master-transmitter terminates the transfer with a STOP condition

2. If a master wants to receive/read data from a slave:
   - Master-receiver sends a START condition and addresses the slave-transmitter
   - Master-receiver sends the requested register to read to slave-transmitter
   - Master-receiver receives data from the slave-transmitter
   - Master-receiver terminates the transfer with a STOP condition

*Note: sending the requested register involves multiple steps. See Section 3.2
2.1.1 START and STOP Conditions

I2C communication with this device is initiated by the master sending a START condition and terminated by the master sending a STOP condition. A high-to-low transition on the SDA line while the SCL is high defines a START condition. A low-to-high transition on the SDA line while the SCL is high defines a STOP condition.

![Figure 5. Example of START and STOP Condition](image)

2.2 Data Validity and Byte Format

One data bit is transferred during each clock pulse of the SCL. One byte is comprised of eight bits on the SDA line. A byte may either be a device address, register address, or data written to or read from a slave. Data is transferred Most Significant Bit (MSB) first. Any number of data bytes can be transferred from the master to slave between the START and STOP conditions.

**Very Important:** Data on the SDA line must remain stable during the high phase of the clock period, as changes in the data line when the SCL is high are interpreted as control commands (START or STOP).

![Figure 6. Example of Single Byte Data Transfer](image)
2.3 Acknowledge (ACK) and Not Acknowledge (NACK)

Each byte of data (including the address byte) is followed by one ACK bit from the receiver. The ACK bit allows the receiver to communicate to the transmitter that the byte was successfully received and another byte may be sent.

Before the receiver can send an ACK, the transmitter must release the SDA line. To send an ACK bit, the receiver shall pull down the SDA line during the low phase of the ACK/NACK-related clock period (period 9), so that the SDA line is stable low during the high phase of the ACK/NACK-related clock period. Setup and hold times must be taken into account.

When the SDA line remains high during the ACK/NACK-related clock period, this is interpreted as a NACK. There are several conditions that lead to the generation of a NACK:

1. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
2. During the transfer, the receiver gets data or commands that it does not understand.
3. During the transfer, the receiver cannot receive any more data bytes.
4. A master-receiver is done reading data and indicates this to the slave through a NACK.

![Diagram](image)

3 I²C Data

Data must be sent and received to or from the slave devices, but the way that this is accomplished is by reading or writing to or from registers in the slave device.

Registers are locations in the slave's memory which contain information, whether it be the configuration information, or some sampled data to send back to the master. The master must write information into these registers to instruct the slave device to perform a task.

While it is common to have registers in I²C slaves, please note that not all slave devices will have registers. Some devices are simple and contain only 1 register, which may be written directly to by sending the register data immediately after the slave address, instead of addressing a register. An example of a single-register device would be an 8-bit I²C switch, which is controlled via I²C commands. Since it has 1 bit to enable or disable a channel, there is only 1 register needed, and the master merely writes the register data after the slave address, skipping the register number.
3.1 Writing to a Slave On The I\textsuperscript{2}C Bus

To write on the \textsuperscript{2}C bus, the master will send a start condition on the bus with the slave's address, as well as the last bit (the R/W bit) set to 0, which signifies a write. After the slave sends the acknowledge bit, the master will then send the register address of the register it wishes to write to. The slave will acknowledge again, letting the master know it is ready. After this, the master will start sending the register data to the slave, until the master has sent all the data it needs to (sometimes this is only a single byte), and the master will terminate the transmission with a STOP condition.

**Note:** The data on SDA line should be set when SCL is low. Never change the data on SDA line when SCL is high because that will trigger start or stop sequence.

Figure 8 shows an example of writing a single byte to a slave register.

3.2 Reading From a Slave On The I\textsuperscript{2}C Bus

Reading from a slave is very similar to writing, but with some extra steps. To read from a slave, the master must first instruct the slave which register it wishes to read from. This is done by the master starting off the transmission in a similar fashion as the write, by sending the address with the R/W bit equal to 0 (signifying a write), followed by the register address it wishes to read from. Once the slave acknowledges this register address, the master will send a START condition again, followed by the slave address with the R/W bit set to 1 (signifying a read). This time, the slave will acknowledge the read request, and the master releases the SDA bus, but will continue supplying the clock to the slave. During this part of the transaction, the master will become the master-receiver, and the slave will become the slave-transmitter.

The master will continue sending out the clock pulses, but will release the SDA line, so that the slave can transmit data. At the end of every byte of data, the master will send an ACK to the slave, letting the slave know that it is ready for more data. Once the master has received the number of bytes it is expecting, it will send a NACK, signaling to the slave to halt communications and release the bus. The master will follow this up with a STOP condition.

Figure 9 shows an example of reading a single byte from a slave register.
4 Verilog

We will be implementing I2C protocol in Verilog so that we can send and receive data from the FPGA and peripheral sensors. I2C requires that the SDA pin on the sensor serve as both an input and an output. The FPGA will act as a master and the peripheral sensors will be slaves.

4.1 Working with bidirectional pins

The FPGA has to serially send data the peripheral devices, such as device addresses, register addresses, and register values. The FPGA also has to be able to receive data from the sensors. All data is serially received and sent as described in the previous section. It is important to note that at any given moment, the SDA pin is either an input or an output and that the FPGA must be able to switch the pin between an input and output state. Therefore, we must ensure that our Verilog is written such that the data pin is considered both an input and an output and is driven appropriately based on context, and we must ensure that our XDC file is written such that the proper hardware is generated for both of these purposes.

In Verilog, a pin that serves as both an input and an output must be marked as an `inout` signal in any module definitions that refer to the pin. When we want to write to the pin, we will assert a low signal or a high signal as normal, but when we want to read from the pin, we will assert a high-impedance state. When a high signal or a low signal is asserted, the pin will behave as an output pin, but when the high-impedance state is asserted, we can read from the pin like we can read from any other input pin. An example is provided below:

```
module top(
    input data_input, mode_select,
    inout inout_pin);

    assign inout_pin = (mode_select == 1'b1) ? data_input : 1'bz;

    reg internal_register;
    always @(*) begin
        internal_register <= inout_pin;
    end
endmodule
```
4.2 Timing diagram for the read/write finite state machine

There are few things that you should watch for when implementing the finite state machine (FSM) for I2C protocol in Verilog. Here is a list:

1. Make sure that the SCL frequency is around 100kHz or below. The main clock on the FPGA runs at ~200MHz, so you will have to create a slower clock for your FSM.
2. An example of the timing diagram for your FSM is shown in the Figure 10. In this figure, the slow clock signal is what is used to run the FSM and is running at ~200kHz.

<table>
<thead>
<tr>
<th>slow_clk</th>
<th>SCL</th>
<th>SDA</th>
<th>SDA mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Idle</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Start</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Data Transfer Address of Slave Device (7 bits)</td>
</tr>
</tbody>
</table>

**Figure 10: Timing diagram for the FSM for the I2C serial protocol**

3. The FSM should start out in the IDLE stage and wait a signal to start transmitting data. This signal can come from the PC by using WireIn or a button. Use one of these two, so that you can easily debug your state machine – you can trigger your emulator on the start signal and collect the necessary data points to observe the signals.

4. The reason for having the slow_clk to dictate the FSM is the following:
   - **SDA has to be set when SCL is low and cannot change when SCL is high.
   - **Never change SDA on the rising or falling edge of SCL. This is a common mistake.
   - Once the first 8 bits are transmitted (7 bits of address and one bit for the read/write bit), the SDA pin must switch from an output pin to an input pin (i.e. high z state). Once you change the directionality of the SDA pin, you will have to read the value of the ACK bit to make sure that the sensor is working OK.
   - You will sample the ACK bit while the SCL pin is high. **Never sample the ACK signal on the rising or falling edge of SCL.
   - Based on all these reasons, you cannot run your state machine based on the SCL signal. In fact, the frequency of slow_clock should be at least twice of the SCL.

5. You should write the FSM such that on every positive edge of slow_clk, the FSM machine advances to a new state with new output values for both SDA and SCL. The
only input to your FSM is slow_clk and you will be always advancing the state machine.

6. For example, as it is shown in the figure above, you will have 40 states in your FSM: start the FSM, send 7 bits of address of the slave device, send 1 bit of R/W and receive one bit back from the slave. The easiest way to write the FSM is to have 40 different states and individually define what is the output value for each state for SCL, SDA and also set the appropriate directionality for the SDA pin (i.e. either input or output).