CoE 111 MP: I2C Controller

November 29, 2017

Introduction

This machine problem extends the I2C controller in previous lab exercises by allowing full bus operation by modeling open-drain connections through tristate buffers. I2C read operations are also introduced, as well as burst modes for both read and write operations.

I2C Bus environment

The real I2C protocol allows multiple devices to be connected through a single bus of SCL and SDA. This bus is pulled up to logic '1' through a pull-up resistor. All devices that connect to the bus use open-drain configurations - they can only pull the bus down to logic '0'. Whenever a device wants to set the bus to logic '1', it just enters a high impedance state, allowing the pull-up resistor to pull the bus to logic '1'.

The standard cell library used in class does not support bidirectional or open-drain ports. In order to model open-drain operation, we will use "wrapper" modules in the testbench which use tristate buffers to model a high impedance state. This module will be provided and should not be synthesized along with your design, as it will only be used for test purposes. Figure 1shows the operation of the wrapper module. Instead of having bidirectional ports in your design, you will need to separate these ports into two - one for input and another for output. To control bus ownership, the tris signal will be used for each wrapper. When x1_tris is '1', the tristate buffer operates in high-impedance mode. Otherwise, x1_out would directly drive x. All input signals from the wrapper are shorted directly to the bus x, such that any changes to x, be it due to another wrapper driving it or it's own x_out signal driving it, should be reflected in the input signal. In the example shown, two wrappers drive x. The timing diagram shows x1's wrapper driving x first, as highlighted in yellow. The wrapper of x2 then drives it moments later. There is an important assumption that at any given time, only one wrapper has his tristate buffer driving the bus. Whenever there are no wrappers driving the bus, it will be pulled up to '1' (this behavior will be done in the testbench. A sample testbench demonstrating the pullup will be provided to you).

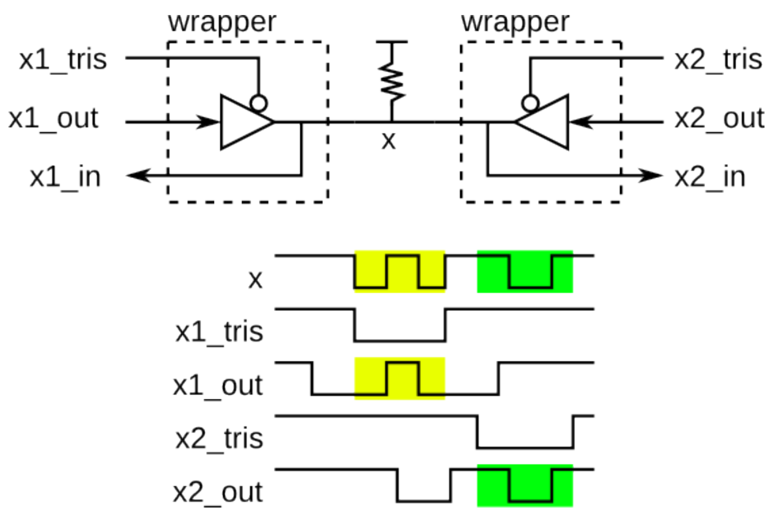

Figure 1: Wrapper module

Controller ports

The controller's ports are divided into 4 groups. The first group consists of the I2C bus signals driving the wrapper module. Table 1 shows the said ports. All ports are 1-bit and connect to two wrapper modules - one for SDA and another for SCL.

Port	Direction	Description
SCL_out	output	SCL signal to wrapper
SCL_in	input	SCL signal from wrapper (from bus)
SCL_tris	output	Controls bus. Set to '0' to drive bus
SDA_out	output	SDA signal to wrapper
SDA_in	input	SDA signal from wrapper (from bus)
SCL_tris	output	Controls bus. Set to '0' to drive bus

Table 1: I2C bus ports

The second group consists of signals used to operate the controller in master mode. The tx_en signal starts master mode operation. Along with the tx_en signal, tx_rd should also be set correspondingly to select either a read or write operation to be performed. The tx_cnt signal determines the number of bytes to be transferred during a read/write operation - 1 byte if tx_cnt = 0, 2 bytes if tx_cnt = 1, and so on. The 7-bit address or 8-bit data to be sent to the slave should be placed in the tx_data input. In cases where no slave responds to the read/write request, the tx_fail signal would assert. Lastly, the busy signal indicates that the I2C bus is currently in use (either by the controller or by another controller), which is useful for preventing the user from enabling tx_en during those times.

Port	Direction	Width	Description
tx_en	input	1	Start read/write operation as master
tx_rd	input	1	Set to '1' for read, '0' for write
tx_cnt	input	2	(Number of bytes to transfer) - 1
tx_data	input	8	Address/Data to send to slave
tx_fail	output	1	Indicates failure to find slave
busy	output	1	Indicates I2C bus is currently used

Table 2: Master mode signals

The third group consists of signals used for reading the internal register file of the controller. These ports are used to read any of the four internal 8-bit registers of the I2C controller. These registers are used for storing data sent to the I2C controller in master or slave mode.

Port	Direction	Width	Description
rd_addr	input	2	Internal register address
rd_data	output	8	Data from internal register

Table 3: Internal register read ports

The last group are miscellaneous ports. The dev_id port sets the 7-bit address of the controller for purposes of identification in the I2C protocol. The baud port sets the baud rate, which determines the duration of the SCL pulses.

Port	Direction	Width	Description
clk	input	1	Global clock (10ns period)
nrst	input	1	Global, active-low reset
dev_id	input	7	I2C address of controller
baud	input	1	Baud rate select

Table 4: Miscellaneous ports

I2C Write Operation

During an I2C write operation, data is sent from master to slave. Figure 2 illustrates a successful write operation. To start the write operation from the idle state, tx_cnt and tx_rd are set appropriately (tx_rd is '0' for writes, and is no longer shown in the figure). The 7-bit address of the target slave device must also be set in the tx_data signal. The tx_en signal is then asserted for at least 1 clock cycle to start the operation. Immediately after starting, the busy signal is asserted. The controller then gets control of the bus by initiating a start bit, followed by sending the 7-bit slave address and write bit serially. Because the address is only 7-bits, the most significant bit of tx_data is dropped. The length of the SCL pulses should be 10ns for baud='0'. The slave then takes control of the SDA bus (as highlighted in yellow) and acknowledges by pulling it down. After acknowledging, the controller pulls down the busy signal and waits for the next data to be sent. During this time, SCL should be pulled low and SDA high. The first byte of data to be sent must then be placed in tx_data, and another assertion of tx_en must be done to start sending that byte. The number of times the controller will do this wait for bytes is determined by the value of tx_cnt in the FIRST assertion of tx_en (when tx_data was set to the slave address). Thus, one cannot start a new read/write operation as long as there are still bytes left to be sent in a pending write operation. Once all bytes have been sent, the controller then sends a stop bit after the last slave acknowledge and returns to the idle state, pulling the busy signal back low in the process.

Take note that while the busy signal is asserted, any assertion to tx_en and changes to tx_cnt, tx_rd, and tx_data are ignored. These three signals should be internally latched by the controller accordingly. The figure also shows the SDA and SCL signals after the wrapper, so you should set the tristate buffer enable signals accordingly.

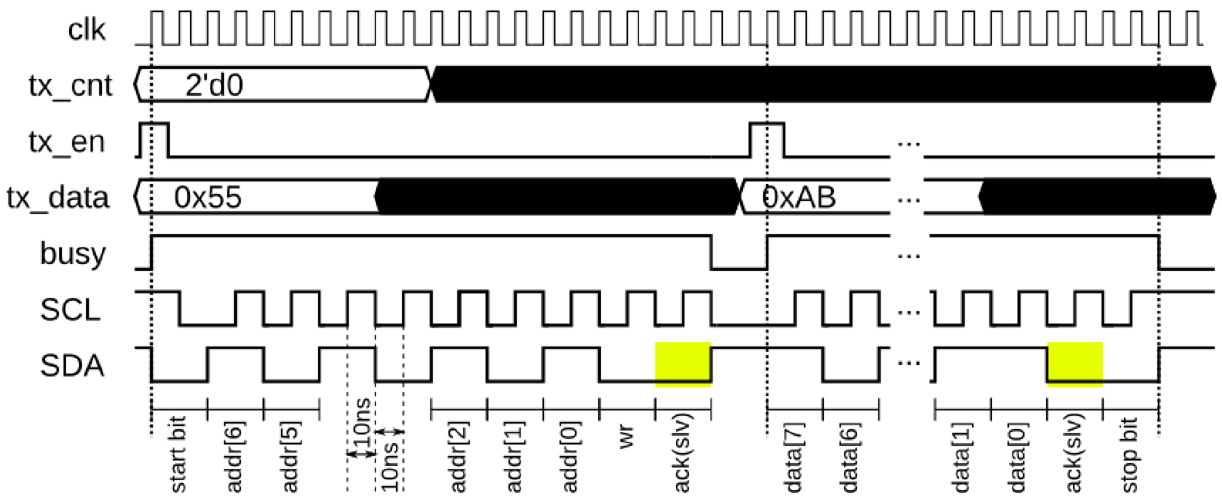

Figure 2: Successful write operation. baud = 0, 1 byte transferred

In cases where there is no slave that matches the target slave address, failure in communication will occur. Figure 3 illustrates this. In the cycle where the slave is supposed to pull down SDA, SDA remains pulled up (by the external pullup). The master (controller) sees this when it pulses SCL, and thus immediately sends a stop bit to end the operation. While the stop bit is being sent, the controller also sets tx_fail to '1' to indicate failure.

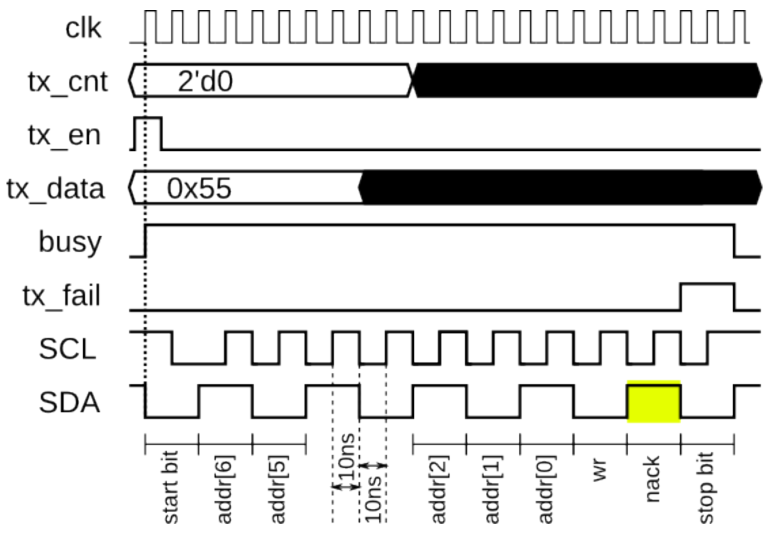

Figure 3: Write operation failure - slave not present.

Slave Mode Operation

While the controller is idle, it should be constantly checking for changes in the I2C bus. Whenever a start bit is initiated by another device, the controller must enter slave mode operation. In this mode, it monitors the I2C bus for the target slave address. If the controller (slave) address (set by the dev_id signal) matches the slave address sent in the I2C bus, then the controller must take control of the SDA bus and acknowledge accordingly (as shown in figure 2). If the operation is a write, the controller must save all data sent to it in it's internal registers. The controller then returns to the idle mode when the stop bit is sent.

The controller has four internal 8-bit registers for storing data sent to it by another I2C device. The registers are addressed from 0 to 3. Whenever data is sent, the controller must save the first byte to register 0, the second byte to register 1, and so on. In cases where less than four bytes are sent, registers which are not written to should retain their old values. Whenever a global reset is done, all registers must be cleared to 0.

If the I2C operation is a read, the controller (slave) must send data in it's internal registers to the I2C master. The number of bytes to be sent out is determined by the value of the slave's tx_cnt signal during the time of it's first acknowledge (immediately after the slave address is sent). The order of bytes sent is from register 0 to register 4. Details on ending the read operation are discussed in the following section.

I2C Read Operation

The read operation starts similarly to the write operation, except that tx_rd should be set to '1' during the assertion of tx_en. Figure 4 illustrates the read operation. Take note that the tx_cnt value set in the master is irrelevant, since the tx_cnt of the slave determines the number of bytes sent. After the slave's first acknowledge, the slave takes control of the SDA bus (as highlighted in yellow) and begins to send the data through it. For each byte of data sent, it sends an acknowledge, except for the last byte. By not sending an acknowledge, the slave tells the master that it no longer has data to send and that the master can now end communication by sending a stop bit. This is necessary because the master has no knowledge of the value of the slave's tx_cnt.

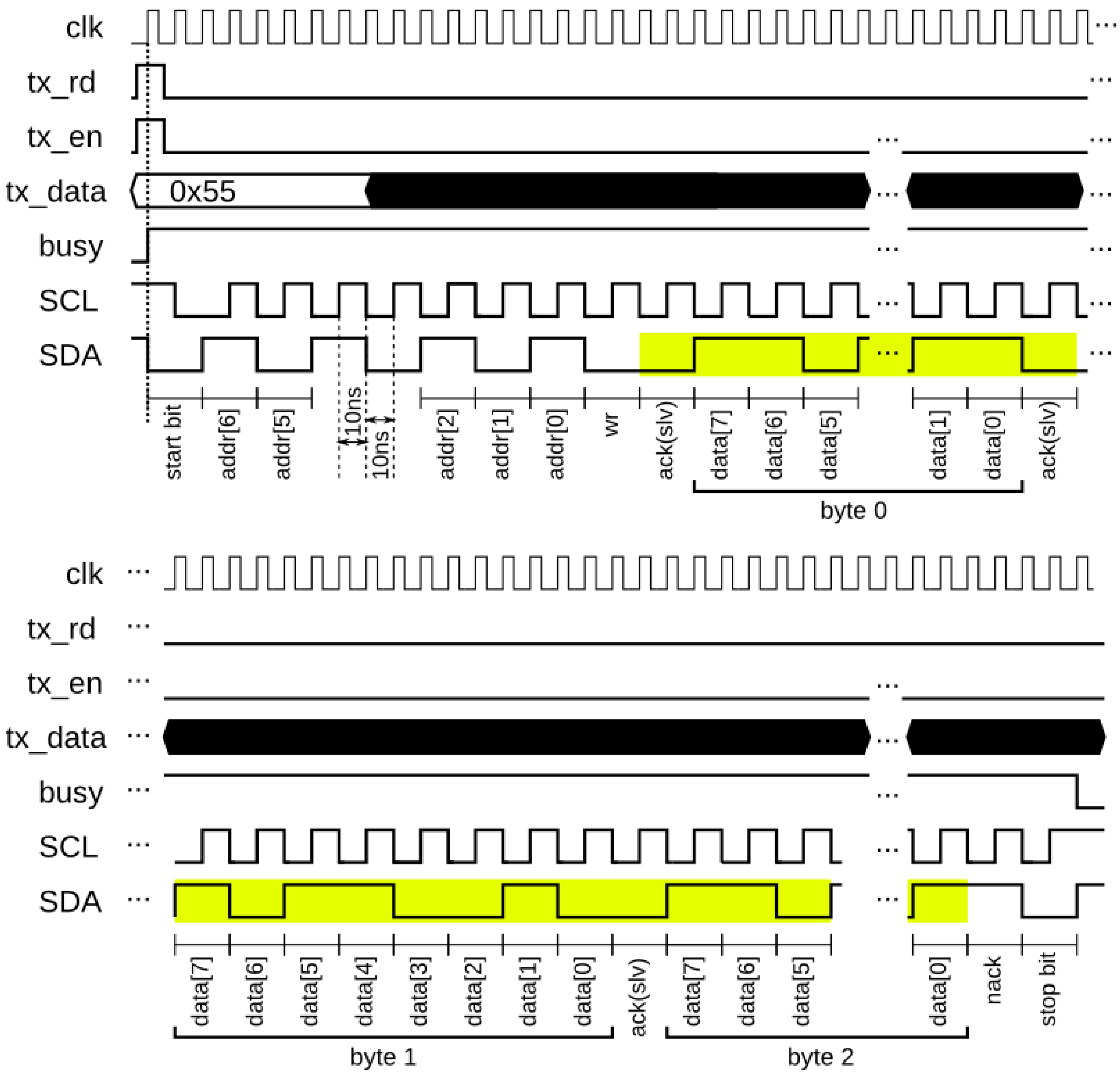

Figure 4: Successful read operation. baud=0, 3 bytes transferred.

Each byte sent to the master during a read operation must be saved to the master's internal registers. The order of saving should be similar to that of the slave mode during writes - byte 0 is saved to register 0, byte 1 saved to register 1, and so on. Registers not accessed should retain their old values.

Failure to communcate with a slave during a read operation should execute in the same way as in figure 3. After failure to acknowledge, the master sends a stop bit and asserts the tx_fail signal for the duration of the stop bit.

Read Port Operation

In order to check the contents of the internal registers, a read port is present for the controller. Setting the rd_addr signal to the address of a certain register should set the rd_data signal to that register's contents. This path must be purely combinational in nature (you don't have to wait for the next clock edge to display the contents).

Baud rate selection

The controller should support two baud rate settings. Figures 2 to 4 show operations using baud=0. For baud=1, all I2C bus signal durations are stretched to double. For example, the pulse width of SCL should now be 20ns, and the duration of the start/stop bits should now be 40ns.

Submission

The I2C controller must be implemented in Verilog and synthesized with the saed90nm generic standard cell library as target. Scoring will be broken down as follows:

- In-class checking
  - Write operation, single byte transfer, baud=0 (20 points)
  - Read operation, single byte transfer, baud=0 (20 points)
  - Failed read/write (20 points)
- E-mail Submission
  - Write operation, multiple byte transfer, baud=0 (10 points)
  - Read operation, multiple byte transfer, baud=0 (10 points)
  - Write operation, multiple byte transfer, baud=1 (10 points)
  - Write operation, multiple byte transfer, baud=1 (10 points)

In class checking will be done on Dec 14 for the Wednesday and Thursday classes, and Dec 15 for the Friday class. Final schedule for in-class checking will be posted in the class web page. E-mail submissions will also have the same deadline as in-class checking. Time deadline is within the day (11:59PM (+0800H) of the same day). Details for e-mail submission are as follows:

- You will be required to so send your whole mp directory for submission. First, make sure that all necessary Verilog files are in the mp5/rtl and mp/mapped directories. Only *.v and *.sdf files should remain in the said directories.
- Delete all other directories and files in the mp directory - only the mp/rtl and mp/mapped directories should remain.
- Create a tarball or zip file containing the mp directory. Name the file as "mp_[surname]_[firstname].[file type]. For example: mp_densing_chris.tar.gz. Only *.tar, *.tar.gz, and *.zip files will be allowed. Other file formats WILL NOT BE ACCEPTED.
- Send the file containing your submission to chris.densing@eee.upd.edu.ph, with the subject header "CoE 111 MP Submission". Failure to follow instructions detailed earlier will net appropriate demerits from your final grade in this exercise.