**I²C**

I²C (Inter-Integrated Circuit) uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock (SCL), pulled up with resistors.

4.7KΩ

4.7KΩ

VDD

SDA

SCL

MASTER

SLAVE

SLAVE

The bus is a multi-master bus which means any number of master nodes can be present. Additionally, master and slave roles may be changed between messages. Each device connected to the bus is referred to as the node.

* Master node — node that generates the clock, start the communication, provide the slave address and end the communication after the message is delivered.
* Slave node — node that receives the clock and address.

The master is initially in master transmit mode by sending a start bit followed by the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write(0) to or read(1) from the slave.

If the slave exists on the bus then it will respond with an ACK bit (active low for acknowledged) for that address. The master then continues in either transmit or receive mode (according to the read/write bit it sent), and the slave continues in its complementary mode (receive or transmit, respectively).

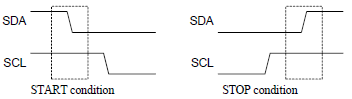
|  |  |
| --- | --- |
| Address 7-bits | Comments |
| 0000 000 | broadcast |
| 1111 xxx | reserved |
| 0000 001 to 1111 000 | Are used to address up to119 slaves |

The address and the data bytes are sent most significant bit first. The start bit is indicated by a high-to-low transition of SDA with SCL high; the stop bit is indicated by a low-to-high transition of SDA with SCL high.

If *any* node is driving the line low, it will be low. Nodes that are trying to transmit a logical one can see this, and thereby know that another node is active at the same time.

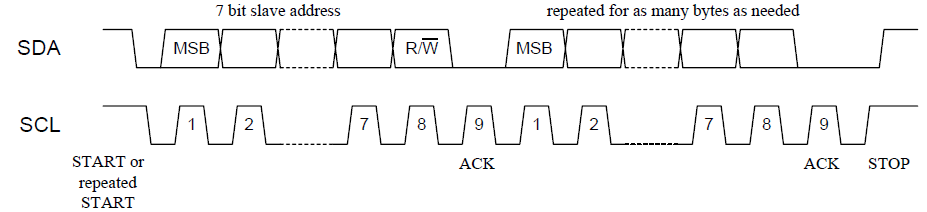
When idle, both lines are high. To start a transaction, SDA is pulled low while SCL remains high.

Except for the start and stop signals, the SDA line only changes while the clock is low; transmitting a data bit consists of pulsing the clock line high while holding the data line steady at the desired level.



While SCL is low, the transmitter (initially the master) sets SDA to the desired value and (after a small delay to let the value propagate) lets SCL float high. The master then waits for SCL to actually go high; this will be delayed by the finite rise-time of the SCL signal (the RC time constant of the pull-up resistor and the parasitic capacitance of the bus), and may be additionally delayed by a slave's clock stretching.

Once SCL is high, the master waits a minimum time (4 μs for standard speed I²C) to ensure the receiver has seen the bit, then pulls it low again. This completes transmission of one bit.



After every 8 data bits in one direction, an "acknowledge" bit is transmitted in the other. The transmitter and receiver switch roles for one bit and the receiver transmits a single 0 bit (ACK) back. After the acknowledge bit, the master may do one of three things:

1. Prepare to transfer another byte of data: the transmitter set SDA, and the master pulses SCL high.
2. Send a "Stop": Set SDA low, let SCL go high, then let SDA go high. This releases the I²C bus.
3. Send a "Repeated start": Set SDA high, let SCL go high, and pull SDA low again. This starts a new I²C bus transaction without releasing the bus.

**Clock stretching using SCL**

An addressed slave device may hold the clock line (SCL) low after receiving (or sending) a byte, indicating that it is not yet ready to process more data. The master that is communicating with the slave may not finish the transmission of the current bit, but must wait until the clock line actually goes high. The master must wait until it observes the clock line going high, and an additional minimum time (4 μs for standard 100 kbit/s I²C) before pulling the clock low again. Normally the Slave will stretch the clock after each byte, until the software decides whether to send a positive acknowledgment or a NACK.

**Arbitration using SDA**

Every master monitors the bus for start and stop bits, and does not start a message while another master is keeping the bus busy. However, two masters may start transmission at about the same time; in this case, arbitration occurs. Each transmitter checks the level of the data line (SDA) and compares it with the levels it expects; if they don't match, that transmitter has lost arbitration, and drops out of this protocol interaction.

If the two masters are sending a message to two different slaves, the one sending the lower slave address always "wins" arbitration in the address stage. Since the two masters may send messages to the same slave address—and addresses sometimes refer to multiple slaves—arbitration must continue into the data stages.

**Master writes one byte of data to the slave:**



* The master initiates the data transfer by issuing the START.
* Transmit the 7-bit slave address plus the write bit (0).
* After receiving an acknowledgment from the slave, the master sends register address.
* The slave responds with an acknowledgment.
* The master sends the data byte.
* After the slave acknowledges the receipt of the data, the master generates the STOP signal.

**The master reads one byte of data from the slave:**



* The master initiates the data transfer by issuing the START.
* Transmit the 7-bit slave address plus the write bit (0).
* After receiving an acknowledgment from the slave, the master sends register address.
* The slave responds with an acknowledgment.
* The master issue a repeated START signal (Sr)
* Send the slave address again but with read bit (1)
* The slave acknowledges.
* The slave sends the data byte from the addressed register.
* The master acknowledges with a zero if it want to receive more data from the slave. If the master does not want to receive any more data it keeps SDA high.
* The master terminates by issuing the STOP signal.

**Step 1:** The object of this step is to Interface to the LCD. The LCD can display two lines of text. Each line could have up to 16 characters.



A Verilog module (LCDI) is provided to interface to the LCD. The LCDI uses a 32 character two port RAM to hold what is displayed. The LCDI module periodically scans the content of the RAM and displays it on the LCD. Three ports DIN, W, and WADD can be used by other modules to load the display RAM.

Controller

LCDI

W

5

WADD

8

DIN

4

dataout

3

control

clk

clk

clk

Include the Verilog provided LCD interface (LCDI) in your design. Describe a simple controller to display your name on the LCD.



**LCD Character Set**

**Step 2: I2C Master:**

Slave 1

scl

sda

Slave 2

scl

sda

Slave 3

scl

sda

I2C Master

scl

sda

Vcc

Controller

8

7

6

scl and sda are inout ports

reset

LCDI

W

5

WADD

8

DIN

4

dataout

3

control

clk

clk

go

go

done

done

ready

ready

rw

rw

N\_Byte

N\_Byte

dev\_add

dev\_add

8

dwr

dwr

8

R\_Pointer

R\_Pointer

drd

drd

ack\_e

ack\_e

The FPGA board clk is 50MHz, if we choose the bus clk (bclk) to be 50KHz then bclk measure in clk cycles is

scl

Seg1

Seg2

Seg3

Seg4

wbit

rbit

ne

pe

Q3

scl period=1000 clk cycles ( a 10-bit binary counter will do the job)

The data bits are injected when scl is low and data bits are read when scl is high. Thus scl is divided in four segments, each 250 clk cycles long (seg=250). The bits are injected at the end of Seg1 and extracted at the end of Seg3. If the slave lowers scl stretching it then stretch signal is asserted. When the 8 least significant bits of the counter are zero the most significant 2 bits of the counter determines whether we are at ne, wbit, pe, or rbit. The stretch signal should be asserted if while in the third quarter (Q3) scl is low otherwise this signal is rested.

When both sda and scl are high the bus is considered idle.

stretch🡨0, Q3🡨0

reset

pe

1

Q3🡨1

rbit

0

Q3🡨0

1

Q3

0

scl

0

1

stretch🡨1

0

1

stretch🡨0

scl

s

I1

I0

0

1’bz

sda

s

sda\_int

I1

I0

0

1’bz

scl\_int

The counter should be reset while in state waiting and it should stop counting when the signal stretch is asserted otherwise it should be incremented at every positive edge of the clk.

counter

state==waitting

count🡨0

1

count🡨0

reset

0

stretch==1’b0

0

count🡨count+1

1

The signal ready is asserted at wbit for the following situations:

1. state=sack2 and R\_W=0
2. state=sack and NB>0
3. state=mack

waitting

reset

go&& idle

RTX🡨R\_Pointer, R🡨{dev\_add,1’b‘0}, NB🡨N\_Byte, R\_W🡨rw, ack\_e🡨 0, done🡨 0

scl\_int🡨1, sda\_int🡨1

1

0

start

ne && sda==0

scl\_int🡨0, bc🡨8

sda\_int🡨0

1

0

1

wbit

sda\_int🡨R[bc-1],  
 bc🡨bc-1

1

0

d\_add

pe

scl\_int🡨0

scl\_int🡨1, sda\_int🡨1,  
drd🡨0, done🡨1, ack\_e🡨0

rbit

0

bc>0

1

0

scl\_int🡨1

1

0

ne

bc==0

0

0

1

1

scl\_int🡨0, sda\_int🡨1

sack1

wbit

sda\_int🡨RTX[bc-1],  
 bc🡨bc-1

1

0

wr\_rp

pe

scl\_int🡨0

bc>0

1

0

scl\_int🡨1

1

0

ne

bc==0

0

0

1

1

scl\_int🡨0, sda\_int🡨1

pe

scl\_int🡨1

0

rbit

ack\_e🡨1

1

0

1

0

ne

1

0

1

scl\_int🡨0, sda\_int🡨1,  
bc🡨8

sda !=1’b0

ack\_e🡨0

sack2

pe

scl\_int🡨1

0

rbit

ack\_e🡨1

1

0

1

0

ne

1

0

1

scl\_int🡨0, sda\_int🡨1,  
bc🡨8, RTX🡨dwr

sda !=1’b0

ack\_e🡨0

R\_W

scl\_int🡨0, sda\_int🡨1,  
bc🡨8,

1

0

Sr

ne && sda==1’b0

scl\_int🡨0, bc🡨8, R🡨{dev\_add,1’b1}

sda\_int🡨0

1

0

1

rbit

0

wbit

scl\_int🡨1

1

wbit

sda\_int🡨R[bc-1],  
 bc🡨bc-1

1

0

d\_add1

pe

scl\_int🡨0

bc>0

1

0

scl\_int🡨1

1

0

ne

bc==0

0

0

1

1

scl\_int🡨0, sda\_int🡨1

Sack3

pe

scl\_int🡨1

0

rbit

ack\_e🡨1

1

0

1

0

ne

1

0

1

scl\_int🡨0, sda\_int🡨1,  
bc🡨8

sda!=1’b0

ack\_e🡨0

0

wbit

sda\_int🡨RTX[bc-1],  
 bc🡨bc-1

1

0

pe

scl\_int🡨0

bc>0

1

0

scl\_int🡨1

1

0

ne

bc==0

0

0

1

1

scl\_int🡨0, sda\_int🡨1,NB🡨 NB-1

wr

sack

pe

scl\_int🡨1

0

rbit

ack\_e🡨1

1

0

1

0

ne

1

0

1

scl\_int🡨0, sda\_int🡨0,

sda !=1’b0

ack\_e🡨0

NB>0

scl\_int🡨0, sda\_int🡨1,  
bc🡨8,RTX🡨dwr

1

0

stop

pe

scl\_int🡨1

0

rbit

1

0

1

0

ne

1

sda\_int🡨1’b1

scl\_int🡨1, sda\_int🡨1,  
drd🡨0, ack\_e🡨0, done🡨1

pe

scl\_int🡨1

1

0

rbit

scl\_int🡨0

bc>0

1

0

RRX[bc-1]🡨sda,  
 bc🡨bc -1

1

0

ne

bc==0

0

0

1

1

scl\_int🡨0, drd🡨RRX,NB🡨 NB-1, RRX🡨 0

rd

mack

pe

scl\_int🡨1

0

sda\_int🡨0

1

0

1

0

ne

1

0

1

scl\_int🡨0, bc🡨8, sda\_int🡨0,

sda\_int🡨1

NB>0

scl\_int🡨0, bc🡨8, sda\_int🡨1,  
,RTX🡨dwr

1

0

wbit

NB>0

**Pre-Lab:**

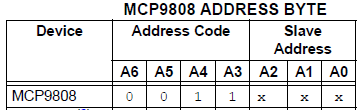
1. Code the I2C Master and simulate it.
2. Acquire a temperature sensor with I2C interface (you might have to get one mounted on a breakout board such as MCP9808)

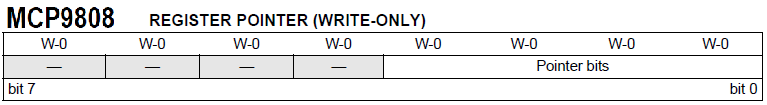
Lab:

1. Implement the I2C Master
2. Include the LCDI module

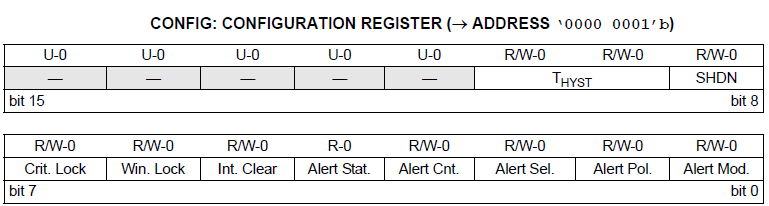
**Step 3: Communication with a temperature sensor (MCP9808)**

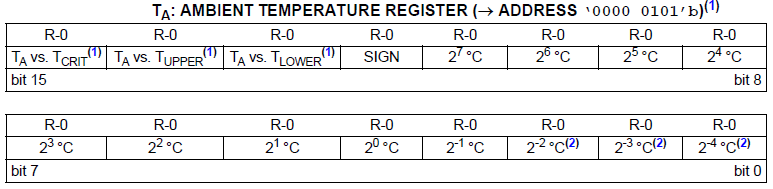
To initiate the communication between the master I2C and an external device the controller after making sure the signal done is asserted, should place the slave address, rw signal, and register pointer (address) on the ports and assert the go signal (for only one clk period). Then it should wait for the ready signal to be asserted. At this time the controller should either read data or write data for next transaction. After all the data are read or written, the controller should then wait for the done signal to be asserted which indicate the completion of the current transaction.





00000001 = Configuration register (CONFIG)  
00000101 = Temperature register (TA)





if(sign)T<=256-data[11:4]; else T<=data[11:4];   
 Config\_R\_add≡1 ; device\_address≡0011\_xxx ; config\_first\_byte ≡0; config\_second\_byte ≡ 0 ; Temp\_R\_add ≡5;

reset

so

done

0

rw🡨0. N\_Byte🡨2, R\_Pointer🡨Confg\_R\_add, dev\_add🡨 device\_addresss

1

go

S1

S2

ready

0

dwr🡨conifg\_first\_byte

1

S3

Controller

S4

ready

0

dwr🡨conifg\_second\_byte

1

S5

done

0

S6

done

0

rw🡨1. N\_Byte🡨2, R\_Pointer🡨Temp\_R\_add, dev\_add🡨 device\_address

1

S7

go

S8

ready

0

Data[15:8) 🡨drd

1

1

S9

ready

0

data[7: 0] 🡨drd

1

S10

Data[12]

0

T🡨256 – data[11:4]

1

T🡨data[11:4]

S11

done

0

D1t🡨0, D2t🡨0

S12

1

S13

T>99

1

D2t🡨D2t+1, T🡨T – 100

0

S14

T>9

1

D1t🡨D1t+1, T🡨T – 10

0

DIN🡨{4’b0011,D2t}, WADD🡨

S15

DIN🡨{4’b0011,D2t},W , WADD🡨  
, W

S16

DIN🡨{4’b0011,T[3:0]}, W ,  
WADD🡨

S17

W

S18

Pre-Lab:

1. Complete and code a controller which uses the I2C Master to:
   1. Configure the temperature sensor.
   2. Read the temperature from the sensor.
   3. Display the temperature after converting it to BCD on the LCD.

Lab:

1. Implement the communication between the FPGA and the temperature sensor
2. Test your system
3. Demonstrate your work.