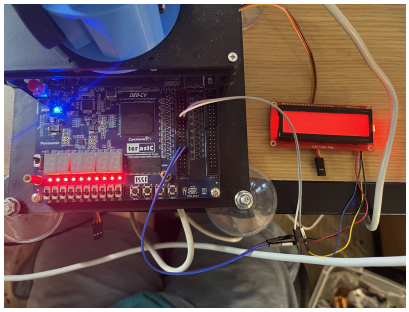
Testing I2C Protocol: Controlling LCD Screen from FPGA



My attempt

I attempted to print “HELLO” on the LCD screen using the FPGA via the I2C protocol. I did not end up getting it to work, but I did learn a lot of things about the I2C protocol. I will go through what I learned by comparing it with the USB protocol.

I used the LCD screen I had left over from 18-100 ([SparkFun 16x2 SerLCD - RGB Text (Qwiic)](#)). I included an annotated implementation of my I2C implementation that sends HELLO in the directory (i2c.sv, https://github.com/coder6583/fpga-i2c).

I2C specification that I read: https://www.nxp.com/docs/en/user-guide/UM10204.pdf\

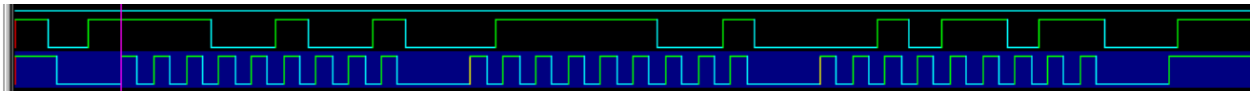


Figure 1. Waveform of my implementation of I2C, SDA(above) and SCL(below)

The I2C protocol physically is two wires, SCL (the clock line) and SDA (the data line), pulled up to the VCC. It is a master and slave system, so the master can send/request data from slaves. I2C allows you to connect multiple masters and slaves to the bus, which is one beneficial aspect of I2C. The clock frequency is usually 100kHz, but it can go up to 400kHz.

Its IDLE state is when both SCL and SDA are released, which means their voltage levels are both HIGH. This is a little bit similar to the USB protocol, in which the IDLE state was J. In SystemVerilog, I write this by declaring SDA and SCL as inout tri1 variables.

The master begins its transaction by pulling SDA low, while the SCL is released. The transaction ends when the master releases SDA after SCL has also been released. One question I had while reading the I2C specification was how these start and stop conditions are distinguished from normal SDA value changes when transmitting data. I figured out it was because all normal data bit changes occur when SCL is being pulled low. Whenever SCL is HIGH, SDA should never change, except for the start and stop conditions. It can be seen in Figure 1, how SDA is stable on every edge of SCL, except for the start and end. I found this solution very elegant and interesting, because for USB, we had to send a whole START byte to initiate the transaction.

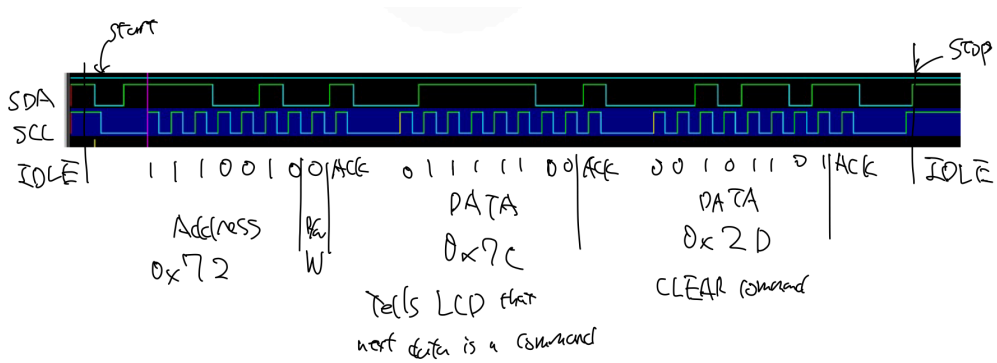
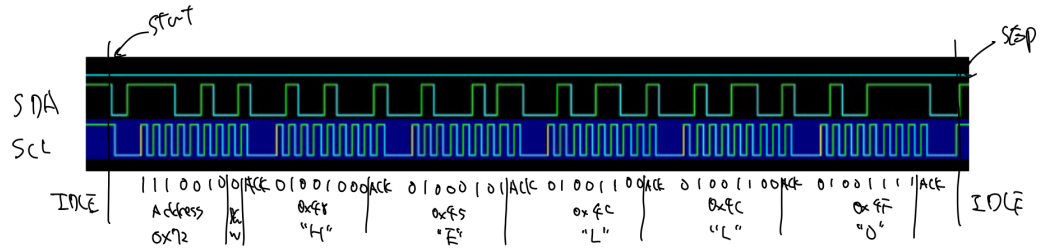
Another thing that interested me regarding the start condition was that I2C also uses the START byte sometimes. However, it was only for special cases when microcontrollers connected to the I2C bus did not have hardware I2C interfaces. This is because if a microcontroller does not have a hardware I2C interface, the microcontroller needs to check every clock cycle using software polling, which is not efficient. Therefore, by using the START byte, it could check only once per every eight clock cycles.

After the master initiates the transaction, it sends the address byte. This is necessary because multiple slaves can be connected to the bus, so there needs to be a way to specify which slave it is trying to communicate with. The address byte is made up of the slave address which is seven bits, and the R/W bit which specifies whether the master is either going to send or receive data. This is similar to the ADDR/ENDP in the IN/OUT packets of the USB protocol.

After the address byte, the master can send one or more bytes as data. Both the address byte and the data byte is MSB. After a byte is sent, the master will release the SDA for one clock cycle and the slave will either pull the SDA low if it successfully received the byte or HIGH if it did not. This is similar to the ACK/NAK packet of the USB protocol. How I2C does the ACK bit is a lot more simple and faster than the USB, but it is also slightly unreliable because it was not specified how a NAK is handled unlike the USB protocol.

After the ACK bit is transmitted by the slave, the slave can pull the SCL low to signal the master to wait to send the next byte. This is necessary because the slave might have some interrupt handler that executes after every byte received, and it will not be ready to receive the next byte. I thought this was interesting because this makes SCL an inconstant clock signal, compared to the USB clock signal which is always constant.

There are a lot more things to cover such as clock synchronization but the above is basically what I needed to understand to write my implementation.

Finally, to print HELLO, I need to send commands that the LCD screen understands. From the 18-100 lab, I knew I needed to send a clear command and then the ASCII code of every letter. Below is an annotated explanation of the waveform from my implementation.

I think one reason my LCD is not printing anything is because of the wait mechanism after the ACK bit. Because of time constraints, I could not do extensive testing on the ACK bit and the wait mechanism, so I will definitely work on that.