I2C Communication protocol

- It consist of a Master and a slave among which there is a communication that exist.

① Master (controller)

⇒ Initiates comms :- Always starts a data transfer by sending START cond^n.

⇒ Generates CLK :- It controls the clk signal SCL for entire bus

⇒ Addresses Slave :- Sends 7bit (or 10 bit) addr to slect a specific device on bus

* There can be multiple slave to a single Master

⇒ Controls Rd / wr op^n :- After address, master sends wr/rd bit to indicate its intension to write or read from slave

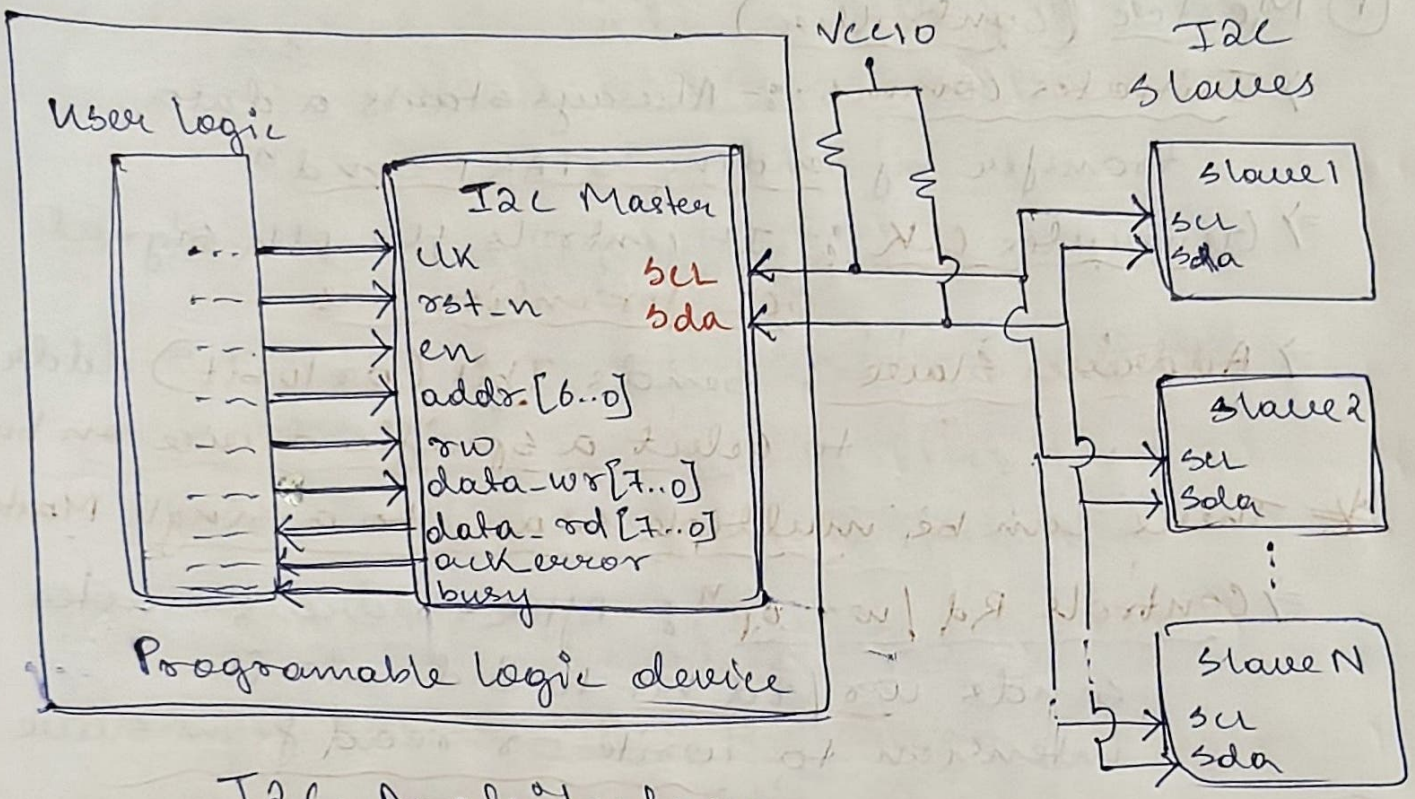
② Slave (peripheral)

⇒ Responds to Master :- Slave cannot initiate transfer; it can only listen to addr and respond when addressed by master

⇒ Receives clk / SCL

* ⇒ Acknowledges the communication :-
After Master sends byte, slave pulls the data line (SDA) Low to send a Acknowledge (ACK) bit confirming successful reception. And if it doesn't pull the line low it sends (NACK) Not Acknowledge

* ⇒ Handles data Request :- when master request data, the slave sends it back, when master sends data, the slave receives and store it



I2C Architecture

- Communication flow — v.v imp

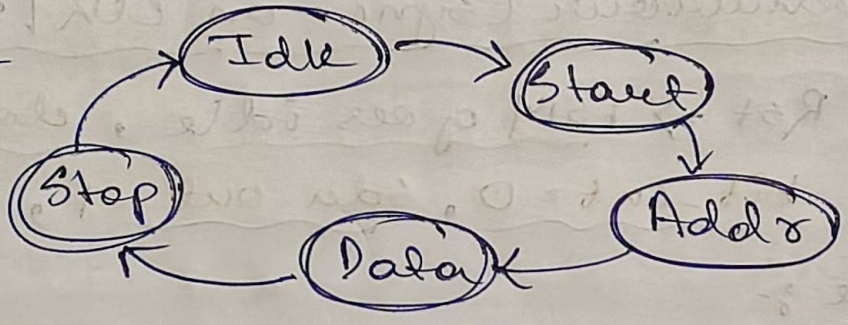
(1) Master sends a START cond^{n}
(2) The master sends slave addr along with rd / wr bit
(3) The addressed slave sends an ACK bit back to Master
(4) The Master - Slave exchange data (dir^{n} is determined by rd/wr bit), with ACK every byte
(5) Master then sends STOP to end transfer allowing Master to start a new one

I2C Master Explained (for code)

⇒ ports and local_params

1. clk, rst = system clock and Reset
2. start = single cycle pulse to begin transfer
3. slave_addr [6:0], data_in [7:0]
4. scl (o/p) = Master driven I2C clock
5. sda (inout) = bidirectional data line; master drives it when sda_oe = 1 otherwise tristated
6. done = asserted when stop has been issued and transfer completes

⇒ FSM :-



⇒ Internal signals

1. state - current FSM state
2. bit_cnt [3:0] - bit index used to shift addr or data (initialised to 7)
3. sda_out - logic value Master drives onto SDA when SDA_oe is enable
4. sda_oe - when 1, master drives SDA - when 0, SDA is high z
5. clk_div [7:0] - simple counter to generate slower SCL from system clk

⇒ SDA tri-state :-

The master drives SDA only while sda_oe = 1 otherwise pin is high-impedance allowing other devices to drive the line

assign sda = sda_oe ? sda_out : 1'bz;

⇒ SCL generation (clk divider) :-

① clk divider (clk_div) increments every clk rising edge. and when it reaches 100 it toggles scl and rst the clk_div.

② on rst scl is set high and clk_div cleared

Result :- A slow sq. wave on scl used as I2C clk (approx. = 2×(100+1)) system clks in divider)

⇒ FSM behaviour (sync. on clk / rst)

↳ on Rst ⇒ FSM goes idle, done=0, bit_cnt=0, sda_out=1, sda_oe=0

① Idle :-

→ done kept low

→ If start asserted, move to START

- drive sda_out = 0 & enable sda_oe = 1 to create a START cond^n
- transition to START state on next clk edge

② START :-

→ Initialises bit_cnt <= 7 and transitn to the Addr state.

(3) ADDR :-

→ Master drives sda_out with bit_cnt-th bit of slave_addr (MSB 1st)

→ sda_oe=1 ensures the master is driving SDA

→ The code checks when SCL is high, it decrements bit_cnt until it hits 0, then transition to DATA.

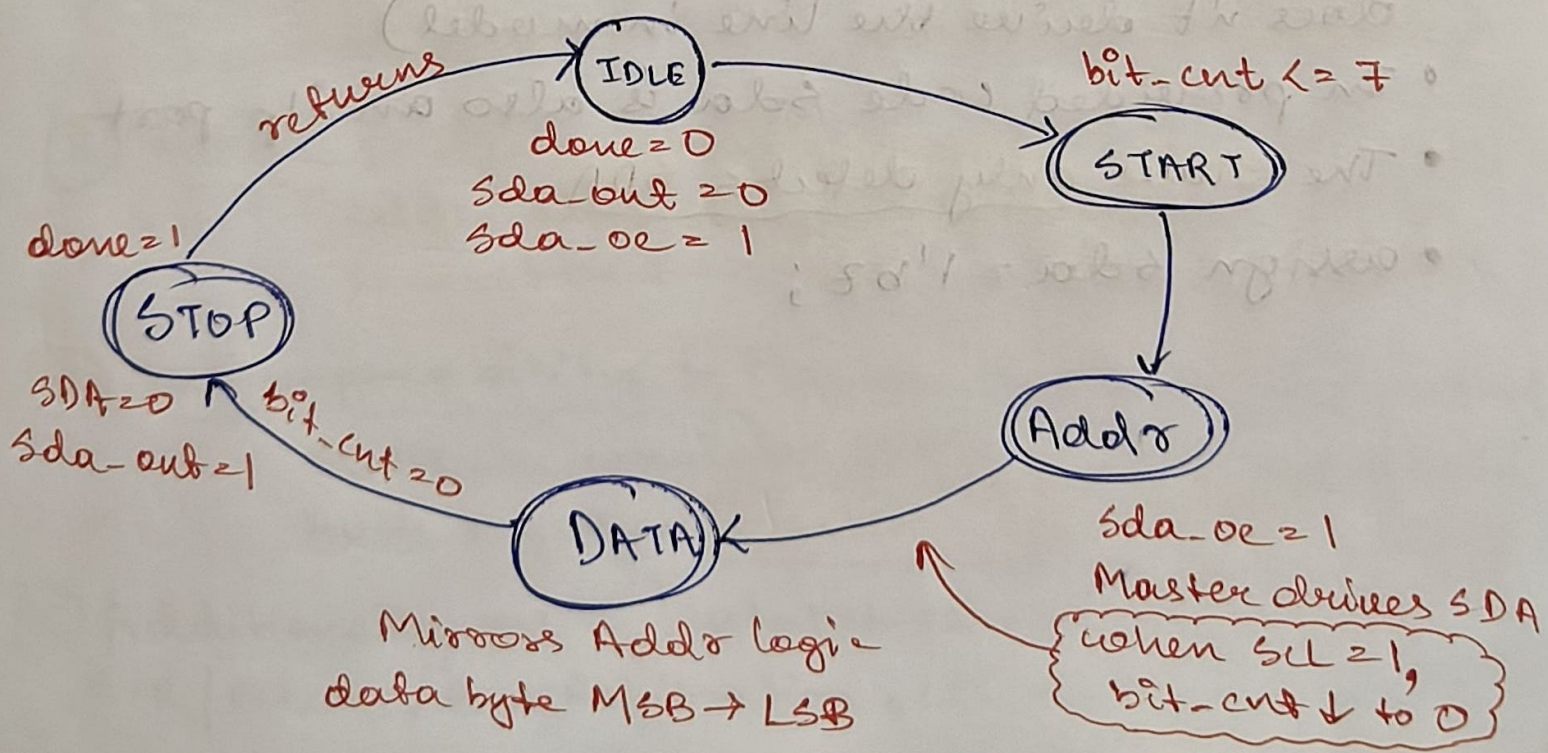
(4) DATA :-

→ Mirrors Addr logic but uses data_in[bit_cnt] drives the data byte MSB → LSB onto SDA on successive SCL high events. when bit_cnt reaches 0 the FSM goes to STOP

(5) STOP :-

→ Drive SDA low then sets sda_out=1 to produce STOP, sets done=1 and return to the IDLE.

⇒ So FSM state of Master goes like :-



- I2C Slave Explained (for code)

⇒ ports :-

① clk, rst = local clk & rst (syn to system clk)
② scl, sda = sampled by slave
③ data_out [7:0] = captured byte when valid
④ valid - pulse indicating data_out holds a newly received byte.

⇒ Internal signals & synchronization :-

① bit_count [2:0] = counts bits shifted [0...7]
② shift_reg [7:0] = accumulates received bits
③ scl_d, sda_d = delayed version of scl & sda for edge detectⁿ
④ (scl_rise = scl & ~scl_d) = detecting rising edge of scl (sample pt.)

⇒ SDA detectⁿ

- Slave assign sda to high-Z (shows slave does it drive the line in model)
- In provided code sda is also an i/p port
- The Slave only detects SDA
- assign sda = 1'bz;

⇒ Sampling & shift Reg. Logic

- on each rising edge of local clk, the code synchronises scl and sda into scl_d & sda_d
- on rst :- clrs counter / reg / status
- otherwise
  - → valid is cleared by default each clk
  - → when scl_rise is detected, slave shifts sda into shift_reg LSB side :

```
shift_reg <= {shift_reg[6:0], sda};
```

  - → increments bit_count. when bit_count == 7
    - ⇒ data_out <= {shift_reg[6:0], sda}
    - ⇒ valid <= 1 to indicate valid data_out
    - ⇒ Resets bit_count <= 0 to set a new byte on next rising edge.

- Observations

(1) Master :- demonstrates START gen., shift Addr and data (MSB → LSB) synch to SCL toggles, and produce STOP; done signal completion.

(2) Slave :- demonstrates how to detect SCL rise edge and sample SDA into shift reg to reconstruct Tx byte and flag it valid.

(3) Interoperability :- Master_tb drive SDA/SCL and Slave_tb emulates SDA/SCL. To fully exercise both together Master's nets ⇒ Slave ports

(4) Additionally more features can be added like ACK/NACK, Bus Arbitration, clk stretching, etc