**I2C (Inter Integrated Circuit)**

**CONTENTS**

1. **LIST OF FIGURES**
2. **LIST OF TABLES**
3. **INTRODUCTION**
4. **BLOCK DIAGRAM**
5. **I2C DATA FRAME STRUCTURE**
6. **STATE DIAGRAM FOR I2C**
7. **TIMING DIAGRAM**
8. **BLOCK DESCRIPTION**
9. **PIN INTERFACES**
10. **RESULTS**
11. **REAL-TIME SCENARIOS**
12. **REFERENCES**
13. **APPENDIX**

# LIST OF FIGURES

## FIGURE:1 (BLOCK DIAGRAM) 5

## FIGURE:2 (TIMING DIAGRAM) 13

## FIGURE:3 (OUTPUT WAVEFORM) 15

# LIST OF TABLES

## Table:1 – Pin\_level\_table 14

# INTRODUCTION

I2C (Inter-Integrated Circuit) is a serial communication protocol widely used to connect low-speed peripherals like LCD, EEPROMs, ADCs, RTCs, and various sensors. It enables data transfer between a master device (typically a microcontroller) and one or more slave devices over just two wires: a data line (SDA) and a clock line (SCL).

Due to its two-wire design, I2C is often referred to as a two-wire communication protocol. The SDA line carries the data, while the SCL line carries the clock signal used to synchronize communication between the master and slave devices.

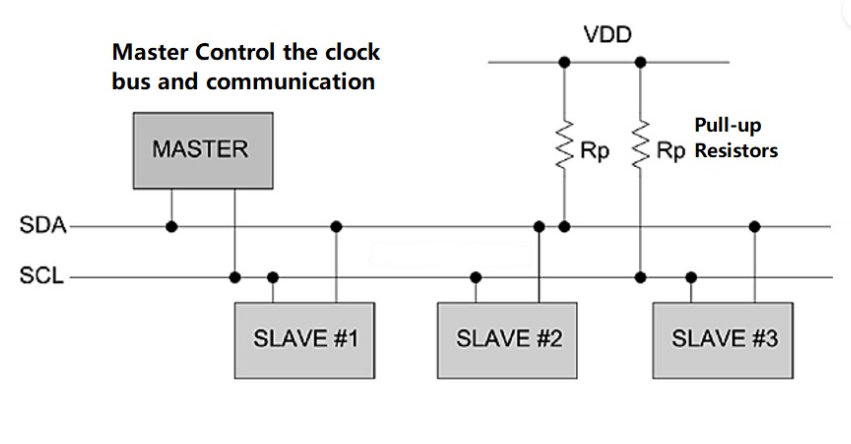
Both SDA and SCL lines are bidirectional, meaning they can be used for sending and receiving signals. I2C is a half-duplex protocol, meaning data transmission happens in one direction at a time. The master initiates communication and can either transmit data to or request data from a slave device.

Feature of I2C Bus:

* In I2C only two buses are required for the communication, the serial data bus (**SDA**) and serial clock bus (**SCL**).
* Each component in the I2C bus is software addressable by a unique address, this unique address is used by the master to communicate with a particular slave.
* I2C supports**7-bit** and **10-bit** addressing structures to target a specific slave (device or IC) on the I2C bus.
* Always a master and slave relationships exist at all times in I2C Bus.
* In I2C, communication always started by the master by sending the start bit.
* During the communication, we get the acknowledgment bit after each byte.
* The I2C bus provides the ability of arbitration and collision detection.
* I2C is the 8-bit oriented serial bidirectional communication, there are the following speed mode supported by the I2C Bus.

|  |  |
| --- | --- |
| MODE | SPEED |
| Standard-mode | 100 kbit/s |
| Fast-mode | 400 kbit/s |
| Fast-mode Plus | 1 Mbit/s |
| High-speed mode | 3.4 Mbit/s |
| Ultra-Fast-mode (UFM) | 5 Mbit/s |

# BLOCK DIAGRAM



The clock bus is used to control the data transfer, and it synchronizes the master and slave together. The master controls the clock signal, making I2C a synchronous serial communication protocol. However, in some scenarios, a slave device may hold the clock line low to delay communication—this is known as clock stretching. I will discuss it in this blog post.

Additionally, I2C is also the foundation for several related control bus architectures, such as:

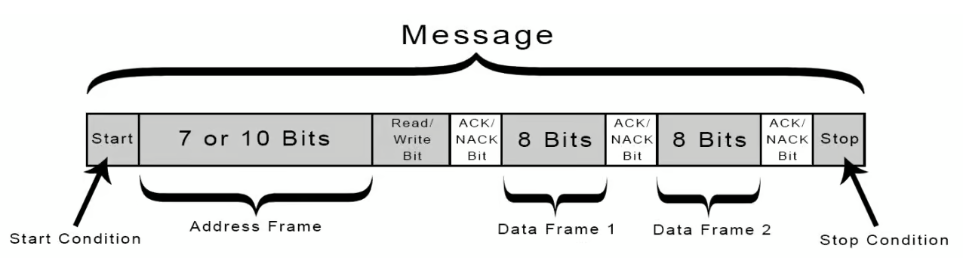
* SMBus (System Management Bus)
* PMBus (Power Management Bus)
* IPMI (Intelligent Platform Management Interface)

You’ll often encounter I2C in embedded projects involving components like:

* I2C LCD displays
* Accelerometers
* Biometric sensors
* OLED screens
* And many other sensor modules

# I2C Data Frame Structure:

* With I2C, data is transferred in messages. Messages are broken up into frames of data.
* Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted.
* The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame



**1. Address Frame:**

* Contains the 7-bit or 10-bit slave address
* Followed by a Read/Write (R/W) bit to indicate direction (0 for write, 1 for read)

**2. Data Frames:**

* Each data frame is 8 bits (1 byte) of payload.
* After each byte, the receiver (slave or master) sends an ACK or NACK to indicate successful receipt

**3. Control Bits:**

* Start Condition (initiates the communication).
* Stop Condition (ends the communication)
* Repeated Start (optional, used in combined message formats)

The following section describes each part of the I2C data frame to provide a clear understanding of the I2C message format

**Default Bus State:**

By default, both the SDA (Serial Data Line) and SCL (Serial Clock Line) are held high due to external pull-up resistors, thanks to the external pull-up resistors. This high state indicates that the bus is idle and ready for communication.

**Start Condition:**

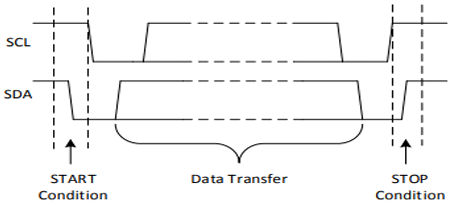
In I2C, every communication begins with a Start Condition. This signal is essential for synchronizing communication between the master and slave devices on the bus. It is a specific signal that tells all devices on the bus that a communication session is about to begin.

A Start Condition occurs when the SDA line transitions from HIGH to LOW while the SCL line remains HIGH. It used by the master to signal the beginning of communication on the bus.

**Stop Condition:**

The STOP Condition is used by the master device to signal the end of communication on the I2C bus. When the master wants to end communication, it first ensures the SDA line transitions from low to high while the SCL line is still high.

A STOP Condition occurs when the SDA line transitions from LOW to HIGH while the SCL line remains HIGH. This specific pattern is recognized by all devices on the bus as the end of the current transaction.



**Address Frame:**

I2C supports both 7-bit and 10-bit addressing schemes. The most common is the 7-bit address, where:

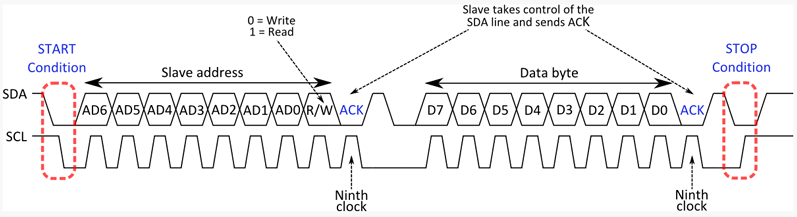
* The first 7 bits represent the slave address.
* The 8th bit (LSB) is the Read/Write (R/W’) bit:
  + 0 = Write
  + 1 = Read

In Slave address bits are used to address a specific slave device on the bus. 7bit address let the master to address maximum of 128slaves on the bus. Although address 0000 000 is reserved for general call and all address of the format 1111 xxx are reserved in many devices. That means 119 device scan share an I2C bus. In I2C bus the MSB of the address is transmitted first.

****

**Data Transfer Sequence:**

Data validity: The data on the SDA line must be stable during the HIGH period of the clock. The HIGH or LOW state of the data line can only change when the clock signal on the SCL line is LOW, One clock pulse is generated for each data bit transferred.



**Read/Write Bit:**

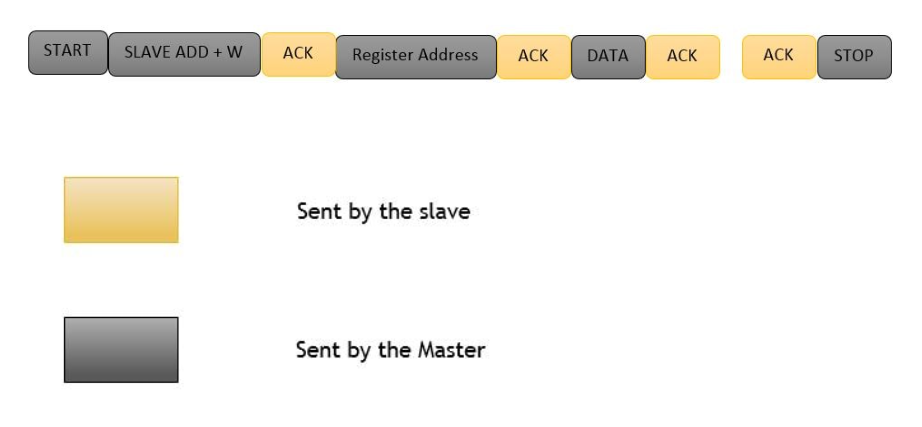
As mentioned earlier, the address frame in I2C communication includes a single Read/Write (R/W) bit at the end. This bit indicates the direction of data transfer between the master and the slave device.

* If the R/W bit is ‘0’, it means the master wants to write data to the slave (i.e., transmit data).
* If the R/W bit is ‘1’, it means the master wants to read data from the slave (i.e., receive data).

1. **write operation**

Before performing a write operation on the I2C bus, the master must first assert a Start condition, followed by transmitting the 7-bit slave address along with the write control bit (R/W bit = 0 for write).

If the transmitted address matches any slave device (e.g., an EEPROM) connected to the I2C bus, the slave responds with an ACK (acknowledge) bit. Upon receiving this ACK, the master sends the register address where it intends to write the data. The slave again acknowledges, indicating it is ready to receive the data.



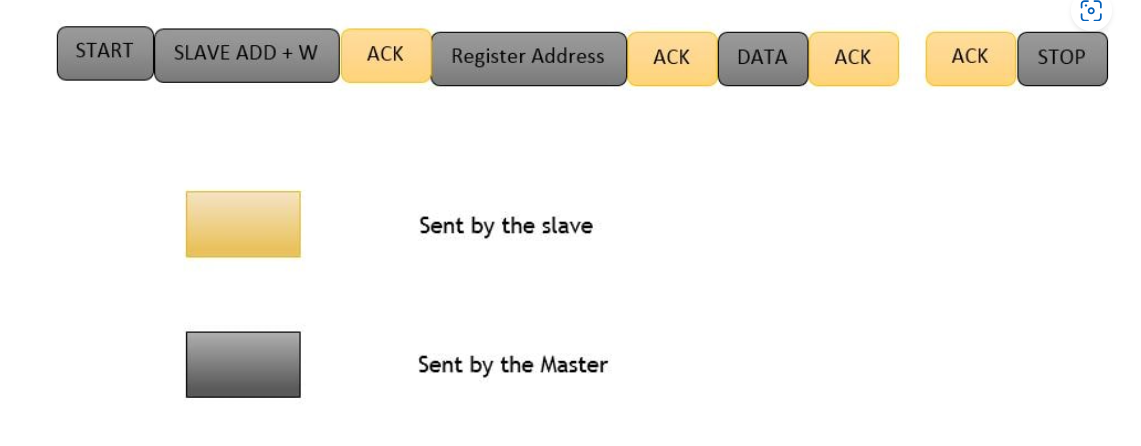
After the second acknowledgment, the master starts sending data bytes to the slave. Each byte is followed by an acknowledgment from the slave. If the master fails to receive an acknowledgment from the slave at any point, it can either:

* Assert a Stop condition to terminate the communication, or
* Assert a Repeated Start condition to attempt re-establishing communication.

Once all data bytes are successfully transmitted and acknowledged, the master ends the transaction by asserting a Stop condition.

1. **I2C Read operation:**

The I2C read operation is similar to the write operation, where the master first asserts a Start condition. After this, the master sends the slave address along with the Read control bit (i.e., R/W bit = 1 for read). If a device on the I2C bus recognizes this address, it responds with an ACK by pulling the SDA (data) line low.



Once the ACK is received, the master releases the SDA line but continues to generate the clock pulses. At this point, the master becomes the receiver, and the slave becomes the transmitter.

During the read operation, the master sends an ACK after receiving each byte to inform the slave that it is ready to receive more data. After receiving the last expected byte, the master sends a NACK to indicate the end of the read operation, followed by a Stop condition to release the bus and terminate the communication.

**ACK/NACK Bit:**

In the I2C protocol, every byte of data (whether it’s an address or data byte) is followed by a single bit known as the ACK (Acknowledge) or NACK (Not Acknowledge) bit. This bit is used by the receiver to inform the sender whether the byte was successfully received.

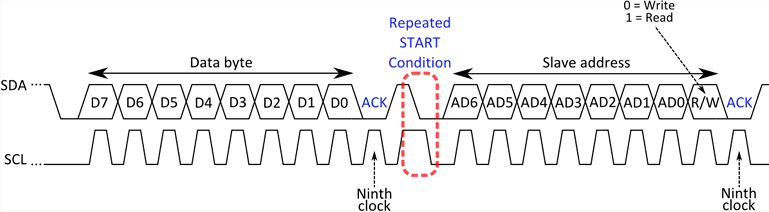
If the receiver successfully receives a byte, it pulls the SDA line low during the 9th clock cycle, which the sender interprets as an ACK. If the receiver does not acknowledge, it leaves the SDA line high, signalling a NACK.

Below are some scenarios, where NACK bit is generated.

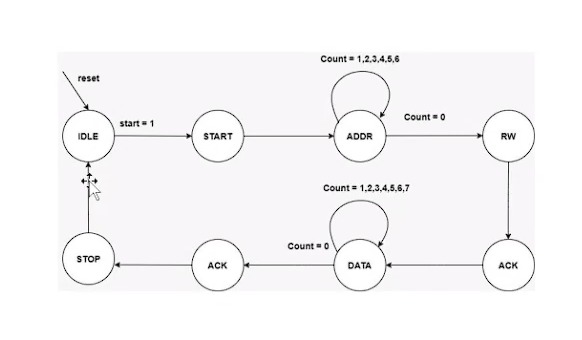
* When the receiver is unable to receive or transmit the data, in that situation it generates a NACK bit to stop the communication.
* During the communication, if the receiver gets any data or commands which are not understood by the receiver then it generates a NACK bit.
* During the transfer, if the receiver performs any real-time operation and not able to communicate with master then assert a NACK bit.
* When Master is a receiver and reads the data from the slave, then after the reading of whole data it asserts a NACK bit on data lines to stop the communication.
* If there is no device present in the I2c bus of the same address which is transmitted by the master, then the master will not get the acknowledge by any slave and treat this situation as NACK

**Repeated Start Condition:**

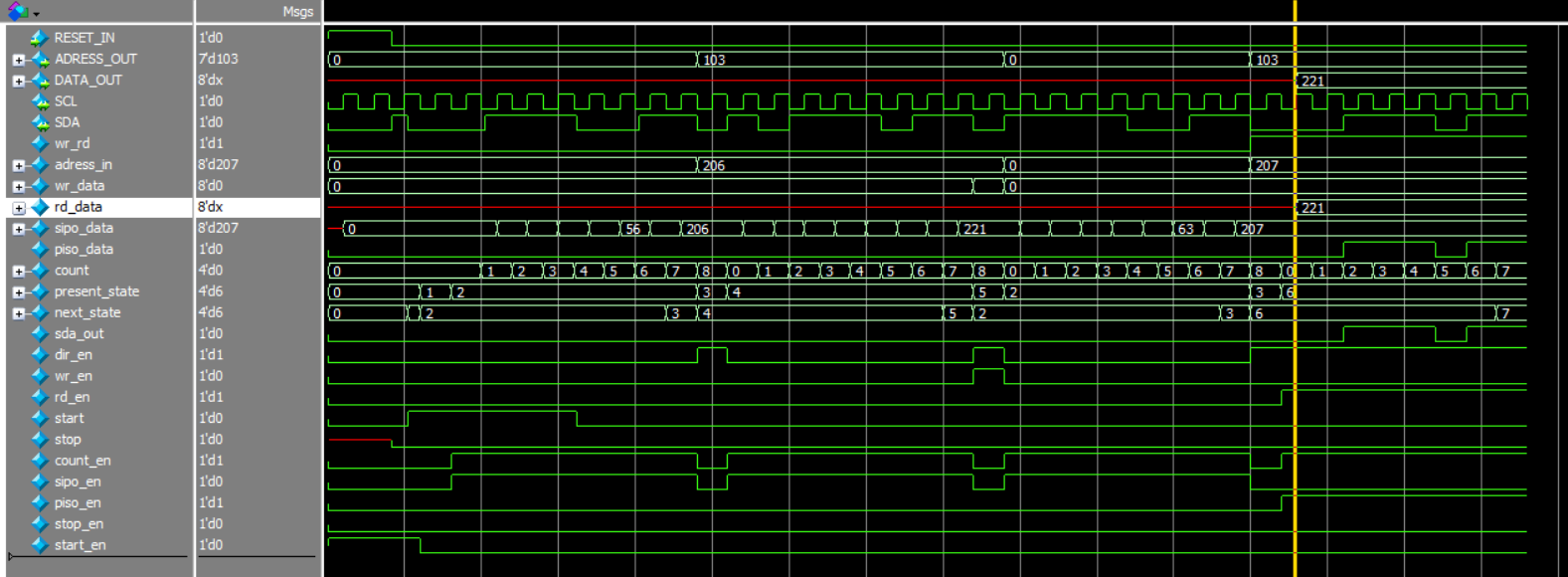
In a multi-master configuration, there is the possibility for the master to own the bus in order to make several transactions without being interrupted by another master. In this case is used what is called a Repeated START condition.



# State Diagram for I2C:



# TIMING DIAGRAMS



# BLOCK DESCRIPTION

## Top-Level Signals

* + RESET\_IN: Goes low (reset active) at the beginning, then high (de-asserted) — simulation starts.
  + ADDRESS\_OUT and DATA\_OUT: Driven values captured from the I²C transactions.

## I²C Lines (SCL & SDA)

* + SCL: Clock line, controlled by master — you can see regular toggling.
  + SDA: Data line — shows actual address and data bits.

## Internal Signals

**Start Condition**

* + Look at start signal — it pulses high → indicates SDA falling while SCL is high.
  + start\_en also activates — triggering FSM to start transaction.

**Address Phase**

* + SDA transmits address: You see bits forming 8'h85 = 10000101:
  + 7-bit address: 0x42 (1000010)
  + R/W bit: 1 → Read operation

**ACK/NACK**

* + After 8 bits, check for SDA low → ACK from slave.
  + ACK/NAK not clearly marked here, but inferred from logic advancing and rd\_en activating.

**Read Phase**

* + piso\_data (Parallel In Serial Out) → Starts transmitting slave data.
  + DATA\_OUT = 0xDD (221), matches waveform.
  + piso\_data shifts bits serially onto SDA.
  + rd\_data also shows 221.

**Stop Condition**

* + stop signal pulses — indicates end of transaction.
  + SDA rises while SCL is high → standard STOP condition.
* Internal FSM States
  + present\_state and next\_state change through values:

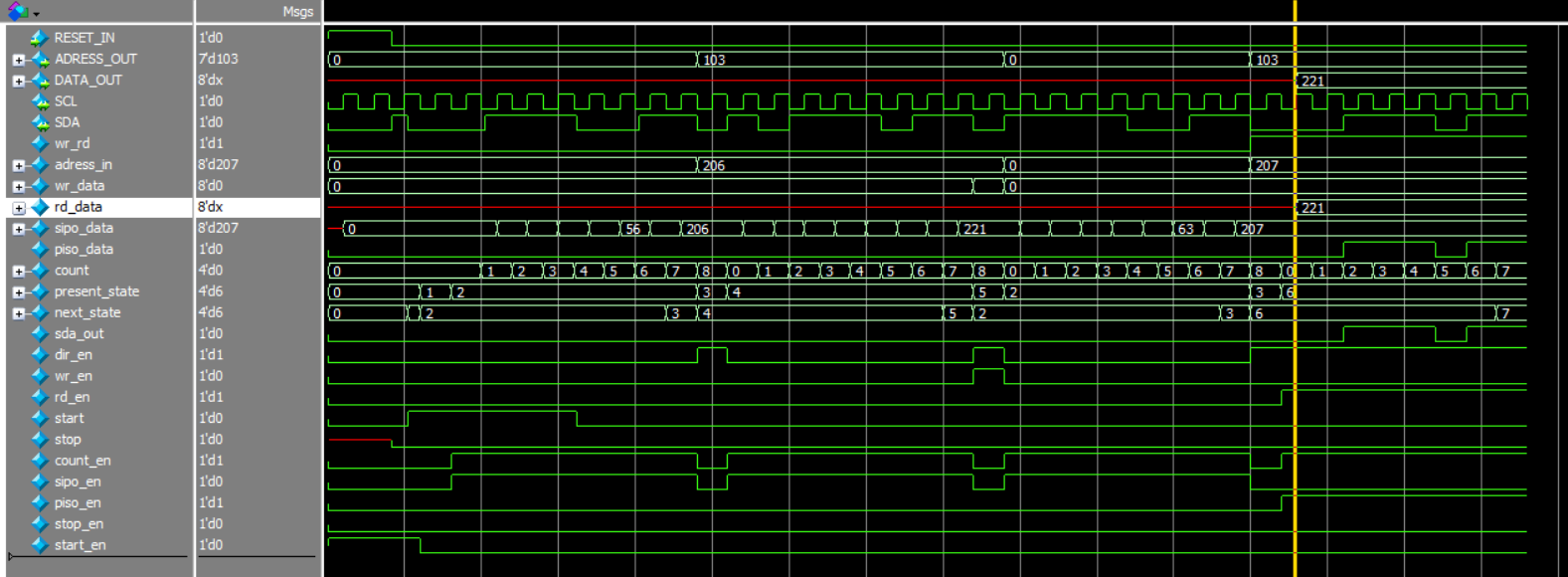
Likely: IDLE -> START -> ADDRESS -> ACK -> DATA -> STOP

* + Corresponding to expected I²C read sequence.
* Counters & Data Buffers
  + count: Increments 0–7 → tracking bits received/transmitted.
  + sipo\_data: Serial-in parallel-out → collects incoming address.
  + piso\_data: Parallel-in serial-out → drives outgoing data.
  + wr\_data, wr\_rd, and address\_in support internal data handling logic.

# PIN INTERFACES

|  |  |
| --- | --- |
| **Phase** | **Observations** |
| Start | Detected correctly with start high |
| Address | 0x85 on SDA → 0x42 with Read |
| ACK | FSM moves forward → ACK likely received |
| Data Transfer | piso\_data sends 0xDD = 221 |
| Stop | Clean stop detected by stop pulse |
| FSM Behavior | Proper transitions in present\_state, next\_state |
| Control Signals | rd\_en, start\_en, stop\_en aligned properly |

# RESULTS



# REAL-TIME SCENARIOS

The AI Accelerator is suited for:

* **EEPROM for Non-Volatile Storage**
  + - Device: 24C02, 24LC256, etc.
    - Scenario: Storing user settings, calibration data, boot configuration.
    - Why I²C?: EEPROMs typically come with I²C for small memory maps.
* **Printers and Scanners**
  + - Scenario: Internal communication between sensors (like paper presence), motor controllers, and the processor.
    - I²C Role: Ideal for connecting multiple internal peripherals.
* **Temperature Sensors in Embedded Systems**
  + - Device: LM75, TMP102, or similar digital temp sensors.
    - Scenario: A microcontroller reads temperature values from multiple sensors over I²C to display or log temperature data.
    - Real-world example: Environmental monitoring systems, HVAC systems, thermostats.
* **Camera Module Control in Mobile Devices**
  + - Device: Smartphones, webcams.
    - Scenario: The main processor configures camera settings (exposure, focus, zoom) via I²C, while image data is transmitted over faster interfaces (like MIPI CSI).
    - Why I²C?: It’s ideal for sending configuration commands.

# REFERENCES

* I2C Protocol Design for Reusability - IEEE
* Design and Implementation of I2C Bus Controller Based on Verilog HDL

# APPENDIX-CODE

**DUT CODE**

module I2C\_slave(RESET\_IN,SCL,SDA,DATA\_OUT,ADRESS\_OUT);

input RESET\_IN;

output [6:0] ADRESS\_OUT;

output [7:0] DATA\_OUT;

inout SCL;

inout SDA;

// signal delcaration

wire wr\_rd; // 0: write , 1 read

reg [7:0] adress\_in;// for the adress and write\_read signal

reg [7:0] wr\_data; // for the data in while writing

reg [7:0] rd\_data; // data\_out while reading

reg [7:0] sipo\_data;// register to convert serila to parallel

reg piso\_data;

reg [3:0] count; // to count upto 8.

reg [3:0] present\_state;

reg [3:0] next\_state;

reg sda\_out;// serial data out from slave

reg dir\_en;// 0:data from master to slave.1 data from slave to master

reg [7:0] mem [127:0];

reg wr\_en;

reg rd\_en;

reg start;

reg stop;

reg count\_en;

reg sipo\_en;

reg piso\_en;

// reg sipo\_valid;

reg stop\_en;

// reg piso\_valid;

reg start\_en;

// parameter declaration to state

parameter state\_idle =4'd0,

state\_start =4'd1,

state\_reg\_addr =4'd2,

state\_reg\_addr\_ack =4'd3,

state\_write =4'd4,

state\_write\_ack =4'd5,

state\_read =4'd6,

state\_read\_ack =4'd7,

state\_stop =4'd8,

state\_stop\_m =4'd9;

assign SDA=dir\_en ? sda\_out:1'bz;

assign DATA\_OUT=rd\_data;

assign {ADRESS\_OUT,wr\_rd}=adress\_in;

// logic to change state

always @ (negedge SCL or posedge RESET\_IN ) begin

if(RESET\_IN) present\_state <=state\_idle;

else begin

present\_state<=next\_state;

end

end

// counter logic

always @ (negedge SCL or posedge RESET\_IN ) begin

if(RESET\_IN) count<=0;

else begin

if (count\_en) count<=count+1;

else count<=0;

end

end

//logic for slave

always @ (posedge SCL or posedge RESET\_IN ) begin

if(!RESET\_IN) begin

if(wr\_en) mem[ADRESS\_OUT]<=wr\_data;

if (rd\_en) begin

rd\_data<=mem[ADRESS\_OUT];

end

end

end

//start condition

always @ (negedge SDA) begin

start<=0;

if (SCL && start\_en ) start<=1;

end

// stop condition

always @( posedge SDA) begin

stop<=0;

if(SCL && stop\_en) stop<=1;

end

//always block for next state

always @ (\*) begin

case(present\_state)

state\_idle: begin

if (start)

next\_state=state\_start;

else next\_state=state\_idle;

end

state\_start :begin

next\_state=state\_reg\_addr;

end

state\_reg\_addr :begin

// waiting for 8 clk period

if(count>=0 && count<7) begin

next\_state=state\_reg\_addr;

end

else begin

next\_state=state\_reg\_addr\_ack;

end

end

state\_reg\_addr\_ack : //slave will generate ack

begin

if (sda\_out==0) begin //ack //if sda=0 then read mode,sda=1 dataout mode

if(wr\_rd==0)begin //write

next\_state=state\_write;

end

else begin //read

next\_state=state\_read;

end

end

else begin // nack

next\_state=state\_stop\_m;

end

end

state\_write :// write state

begin

// waiting for 8 clk period

if(count>=0 && count<7) begin

next\_state=state\_write;

end

else begin

next\_state=state\_write\_ack;

end

end

state\_write\_ack : //slave will generate ack

begin

if (sda\_out==0) next\_state=state\_reg\_addr;

else next\_state=state\_stop\_m;

end

state\_read :// read state

begin

// waiting for 8 clk period

if(count>=0 && count<7) begin

next\_state=state\_read;

end

else begin

next\_state=state\_read\_ack;

end

end

state\_read\_ack : //master will generate ack

begin

if (SDA==0) next\_state=state\_reg\_addr;

else next\_state=state\_stop\_m;

end

state\_stop\_m: begin

if (stop) next\_state=state\_stop;

else if (start) next\_state=state\_start;

end

state\_stop : begin if (start) next\_state=state\_start;

else next\_state=state\_stop;

end

default: next\_state=state\_idle;

endcase

end

//always block for output

always @ (\*) begin

case(present\_state)

state\_idle :begin

dir\_en=0;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

stop\_en=0;

start\_en=1;

adress\_in=0;

wr\_data=0;

end

state\_start :begin

dir\_en=0;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

start\_en=0;

adress\_in=0;

wr\_data=0;

end

state\_reg\_addr :begin

dir\_en=0;

sda\_out=0;

count\_en=1;

piso\_en=0;

sipo\_en=1;

wr\_en=0;

rd\_en=0;

start\_en=0;

adress\_in=0;

wr\_data=0;

end

state\_reg\_addr\_ack : //slave will generate ack

begin

adress\_in=sipo\_data;

dir\_en=1;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

wr\_data=0;

end

state\_write :// write state

begin

dir\_en=0;

sda\_out=0;

count\_en=1;

piso\_en=0;

sipo\_en=1;

wr\_en=0;

rd\_en=0;

//adress\_in=0;

wr\_data=0;

rd\_en=0;

end

state\_write\_ack : //slave will generate ack

begin

dir\_en=1;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_data=sipo\_data;

wr\_en=1;

rd\_en=0;

//adress\_in=0;

end

state\_read :// read state

begin

dir\_en=1;

rd\_en=1;

sda\_out=piso\_data;

count\_en=1;

piso\_en=1;

sipo\_en=0;

wr\_en=0;

//adress\_in=0;

wr\_data=0;

end

state\_read\_ack: //master will generate ack

begin

dir\_en=0;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

//adress\_in=0;

wr\_data=0;

end

state\_stop\_m :begin

dir\_en=0;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

stop\_en=1;

adress\_in=0;

wr\_data=0;

$display("stope\_en=%b,stop=%b,state=%0b",stop\_en,stop,present\_state);

end

state\_stop :begin

dir\_en=0;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

adress\_in=0;

wr\_data=0;

end

default: begin

dir\_en=0;

sda\_out=0;

count\_en=0;

piso\_en=0;

sipo\_en=0;

wr\_en=0;

rd\_en=0;

adress\_in=0;

wr\_data=0;

end

endcase

end

// code for SIPO

always @ (posedge SCL) begin

if (RESET\_IN) sipo\_data<=0;

else begin

if

(sipo\_en) sipo\_data<={SDA,sipo\_data[7:1]};

end

end

// code for PISO

always @ (negedge SCL) begin

if (RESET\_IN) piso\_data<=0;

else begin

if (piso\_en) begin

piso\_data<=rd\_data[count+1];

end

end

end

endmodule

**TESTBENCH**

`include "i2c\_slave.v"

module I2C\_TOP\_test;

reg RESET;

wire [6:0] ADRESS\_OUT;

wire [7:0] DATA\_OUT;

wire SCL;

wire SDA;

reg clk;

reg sda;

I2C\_slave dut(RESET,SCL,SDA,DATA\_OUT,ADRESS\_OUT);

assign SCL=clk;

assign SDA=dut.dir\_en?1'bz:sda;

// for initialzing input values

task initialization;

begin

clk=0;

sda=0;

RESET=0;

$display($time,"@ IN INTIALIZATION");

end

endtask

task rst;

begin

RESET=1;

repeat (2) @(negedge clk);

#2;

RESET=0;

end

endtask

// clock generation

always #10 clk=!clk;

// task to generate start condition

task start\_gen;

begin

$display($time,"@ start gen");

sda=1;

@ (posedge clk);

#2;

sda=0;

@ (negedge clk);

end

endtask

// task to generate stop condtion

task stop\_gen;

begin

$display($time,"\t@stop gen");

@ (negedge clk);

sda=0;

@(posedge clk); #2;

sda=1;

end

endtask

// task to generate stop condtion

task stop\_gen2;

begin

$display($time,"\t@stop gen 2");

$display($time,"sda=%b",sda);

sda=0;

@(negedge clk);

#2;

$display($time,"sda=%b",sda);

sda=1;

@(posedge clk);

$display($time,"sda=%b",sda);

end

endtask

// task to generate adress values

task wr\_address;

reg [7:0] temp;

begin

temp=8'b1100\_1110;// 110\_0111='h67/'d103 0:write //1100\_1110='b206/'h ce

$display($time,"temp\_disp=%b",temp,sda);

repeat (8) begin

@( negedge clk) ;#2;

sda=temp;

temp=temp>>1;

$monitor($time,"temp=%b,sda=%b",temp,sda);

end

@( negedge clk) ;// for ack signal

end

endtask

task rd\_address;

reg [7:0] temp;

begin

temp=8'b1100\_1111; // 110\_0111='h67/'d103 0:write //1100\_1110='b206/'h ce

repeat (8) begin

@( negedge clk) ;

sda=temp;

temp=temp>>1;

end

@ (negedge clk);// for ack signal

end

endtask

// task to generate write data values

task write\_data;

reg [7:0] temp;

begin

temp=8'b 1101\_1101; //dd

repeat (8) begin

@(negedge clk);

sda=temp;

temp=temp>>1;

end

@ (negedge clk);// for ack signla

end

endtask

// task to get read data values

task read\_data;

begin

repeat (8)begin

@ (negedge clk);

end

@(negedge clk)sda=0; // ack

end

endtask

// calling all task

initial begin

initialization;

rst;

start\_gen();

wr\_address();

write\_data;

rd\_address();

read\_data;

//#800;

//stop\_gen();

//@(dut.present\_state==)stop\_gen2();

//#1000;

$stop;

//#200;

//$finish;

end

//initial begin

// // Dump waves

// $dumpfile("dump.vcd");

// $dumpvars;

//end

endmodule