JUNE 26, 2021

DIGITAL SYSTEM DESIGN

TEAM NO 8

Yash Mahale – 01fe19bec265
Ifra – 01fe19bec266
Bharat Gunhalkar – 01fe19bec274

Contents

Problem Statement:

"WRITE A VERILOG CODE TO IMPLEMENT THE I2C PROTOCOL"

Explanation:

I2C bus controller is designed in Verilog and simulated in Xilinx and ISim. The working of I2C is software addressing based which can attach large number of devices by using only two lines SDA and SCL and pull up resistors. Any low speed peripheral devices can be interfaced using I2C bus protocol as a master.

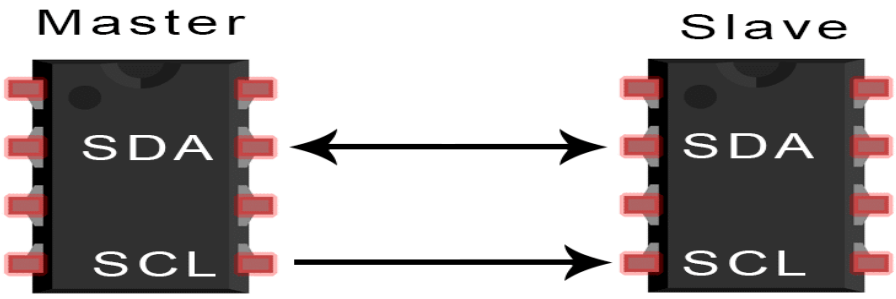
Introduction:

I. I2C Protocol

I2C communication is the short form for inter-integrated circuits. It is a communication protocol developed by Philips Semiconductors for the transfer of data between a central processor and multiple ICs on the same circuit board using just two common wires. Owing to its simplicity, it is widely adopted for communication between microcontrollers and sensor arrays, displays, IoT devices, EEPROMs etc. This is a type of synchronous serial communication protocol. It means that data bits are transferred one by one at regular intervals of time set by a reference clock line.

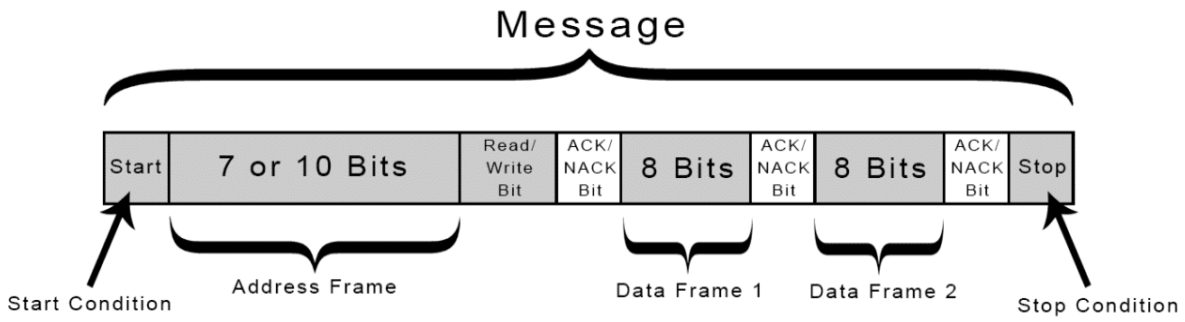
II. Working

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD. Like UART communication, I2C only uses two wires to transmit data between devices:



SDA (Serial Data) – The line for the master and slave to send and receive the data. **SCL (Serial Clock)** – The line that carries clock signal.

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:



Start Condition:

The SDA line switches from a high voltage level to low voltage level before the SCL line switches from high to low.

Stop Condition:

The SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high.

Address Frame:

I2C doesn't have slave select lines like SPI, so it needs another way to let the slave know that data is being sent to it, and not another slave. It does this by addressing. The address frame is always the first frame after the start bit in a new message.

The master sends the address of the slave it wants to communicate with to every slave connected to it. Each slave then compares the address sent from the master to its own address. If the address matches, it sends a low voltage ACK bit back to the master. If the address doesn't match, the slave does nothing and the SDA line remains high.

Read/Write Bit:

The address frame includes a single bit at the end that informs the slave whether the master wants to write data to it or receive data from it. If the master wants to send data to the slave, the read/write bit is a low voltage level. If the master is requesting data from the slave, the bit is a high voltage level.

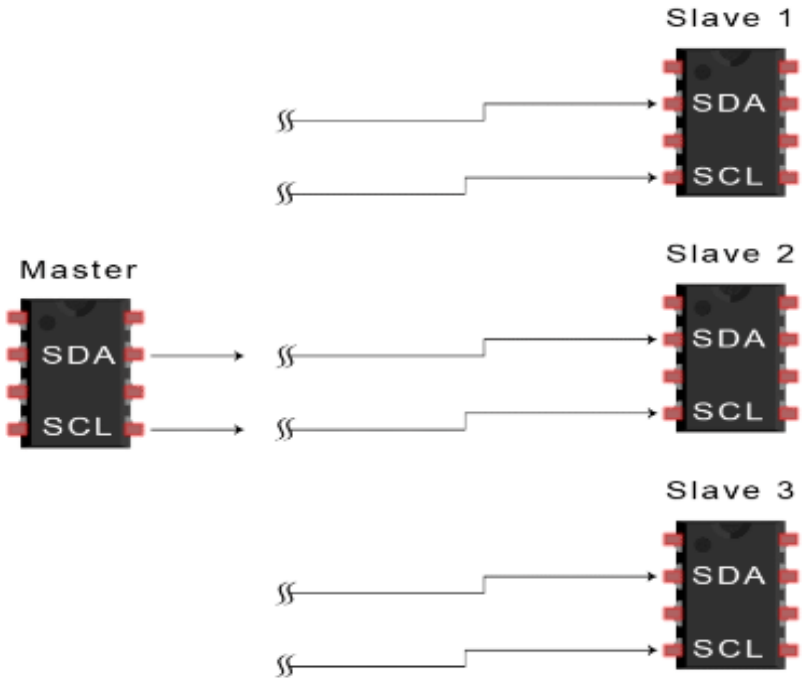
The Data Frame:

After the master detects the ACK bit from the slave, the first data frame is ready to be sent. The data frame is always 8 bits long, and sent with the most significant bit first. Each data frame is immediately followed by an ACK/NACK bit to verify that the frame has been received successfully. The ACK bit must be received by either the master or the slave (depending on who is sending the data) before the next data frame can be sent. After all of the data frames have been sent, the master can send a stop condition to the slave to halt the transmission. The stop condition is a voltage transition from low to high on the SDA line after a low to high transition on the SCL line, with the SCL line remaining high.

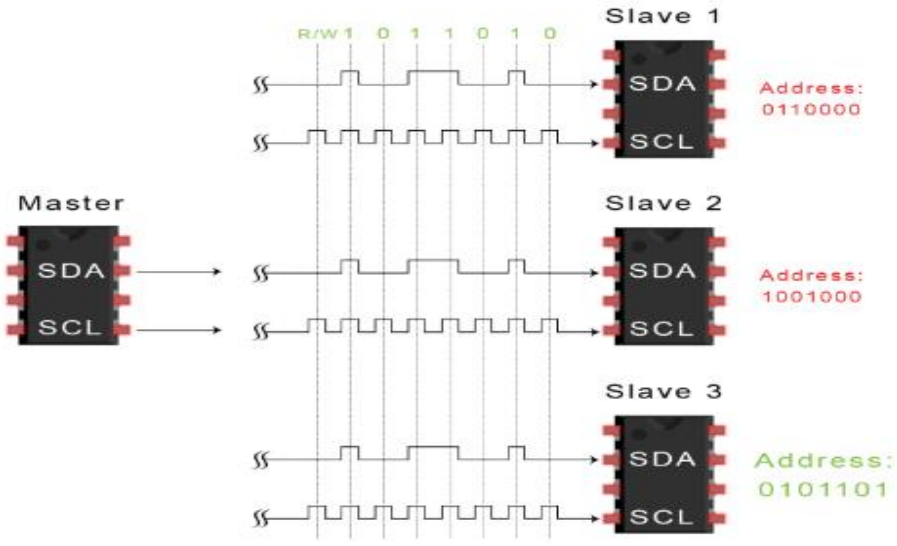
III. Steps of I2C Data Transmission

Single Master with Multiple Slaves

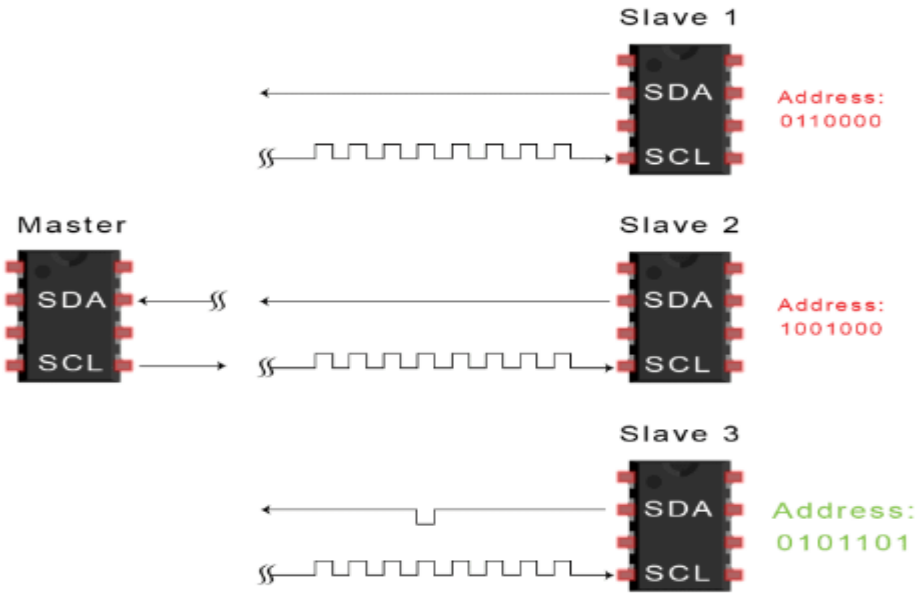
1) The master sends the start condition to every connected slave by switching the SDA line from a high voltage level to a low voltage level *before* switching the SCL line from high to low



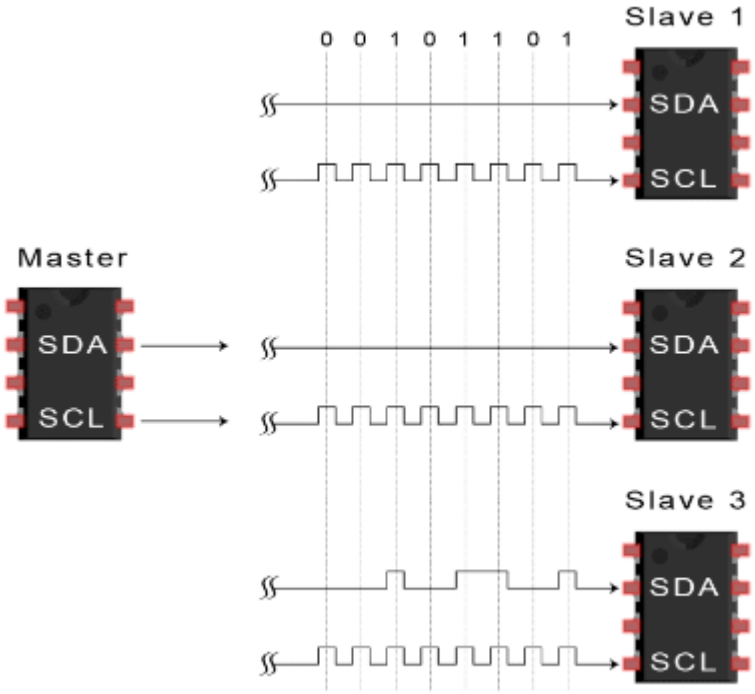
2) The master sends each slave the 7- or 10-bit address of the slave it wants to communicate with, along with the read/write bit:



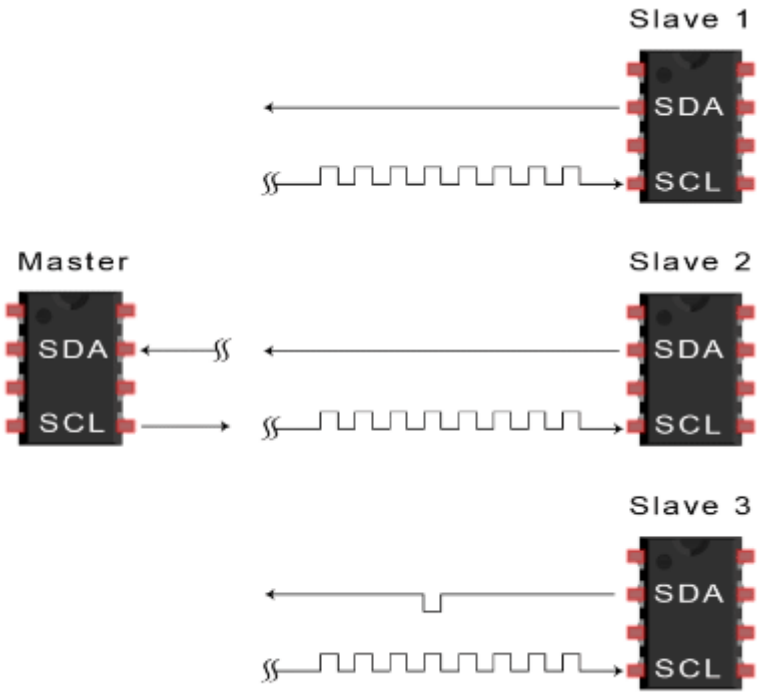
3) Each slave compares the address sent from the master to its own address. If the address matches, the slave returns an ACK bit by pulling the SDA line low for one bit. If the address from the master does not match the slave's own address, the slave leaves the SDA line high.



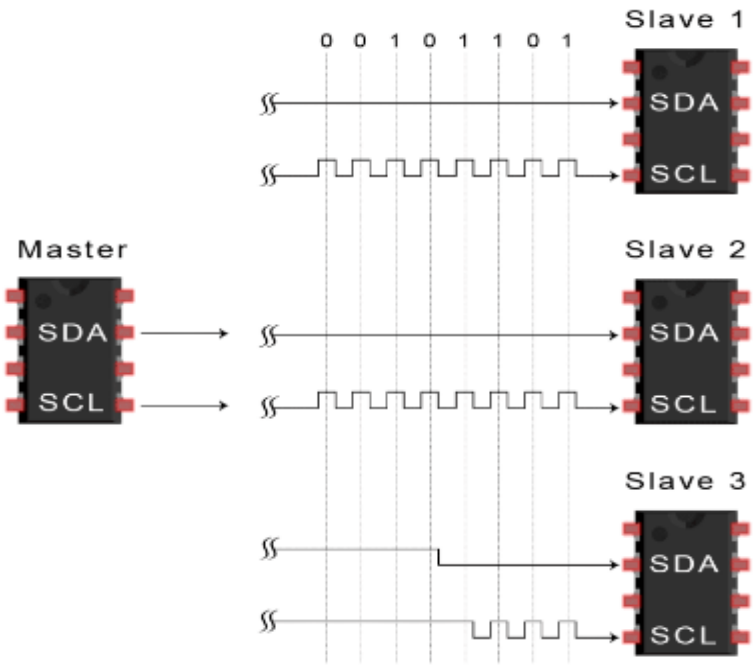
4) The master sends or receives the data frame:



5) After each data frame has been transferred, the receiving device returns another ACK bit to the sender to acknowledge successful receipt of the frame:

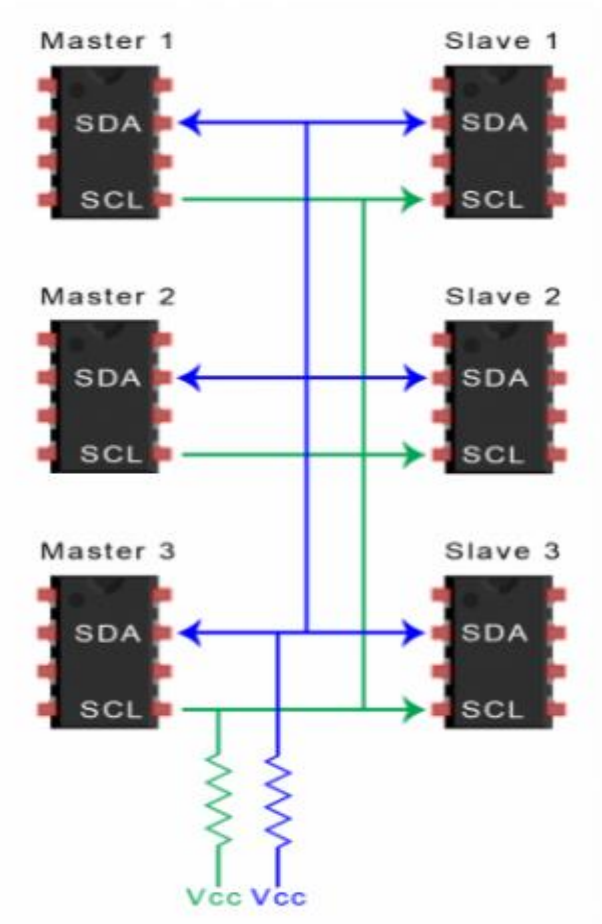


6) To stop the data transmission, the master sends a stop condition to the slave by switching SCL high before switching SDA high:



Multiple Masters with Multiple Slaves.

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it's safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



Design Procedure:

- The clock signal (SCL) is set at the frequency of 396Khz. (the frequency can be modified according to the standard: 100Khz/400Khz and 3.4Mhz).

- The data signal (SDA) is a bidirectional line, which can be read and written, by the master and the slave.

- The communication between a master and a slave begins with a START condition followed by the slave address to be reached, one read/write bit, a bit of recognition that can be ACK (if the communication was successful) or NACK (if the communication was unsuccessful or the end of the message is set by the master), the 8 bits of data to send or to receive, the ACK or NACK bit, and finishes with a STOP condition or a condition Repeated START.

- If there is no communication between a master and a slave, the data and clock signals remain in the high impedance state.

- The protocol neither implements the designed multi-master function, nor the extension function clock, nor the sending and receiving of more than one da-tum.

- Each master and slave manage two operating modes: receive and send data.

- The slave address is 7-bits length.

- If the master receives a non-recognized signal by the slave, a STOP condition is generated.

- When the master or the slave does not acknowledge the received data, both the clock and the data signals change to the high impedance state and thus releasing the bus.

Applications:

- Computing
  - Voltage Translation
  - Power Supply
  - Fan Control
  - Storage Server

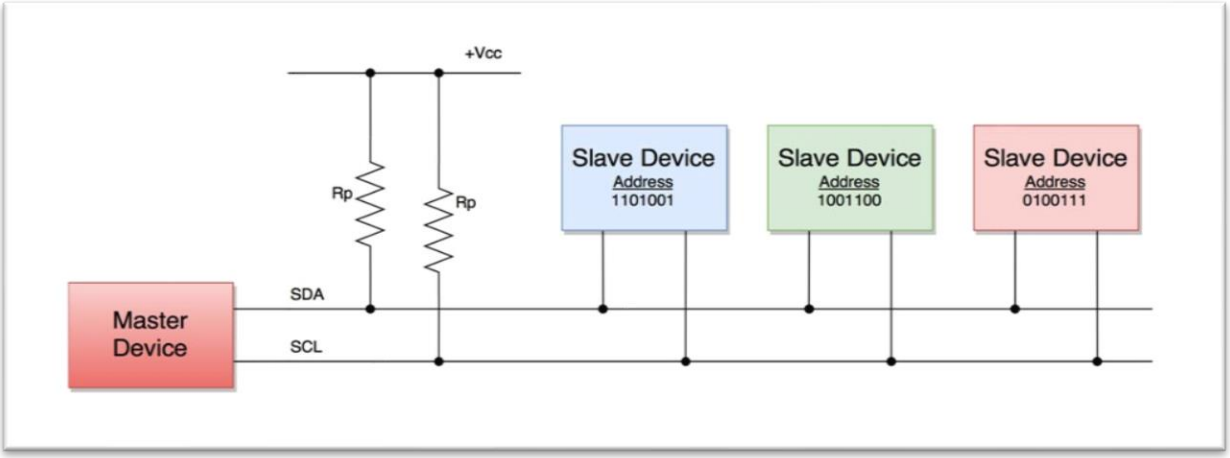
- Communication
  - Router
  - Networking line card
  - Advanced TCA

- Industrial
  - Bus Buffers
  - Bridges
  - GPIO

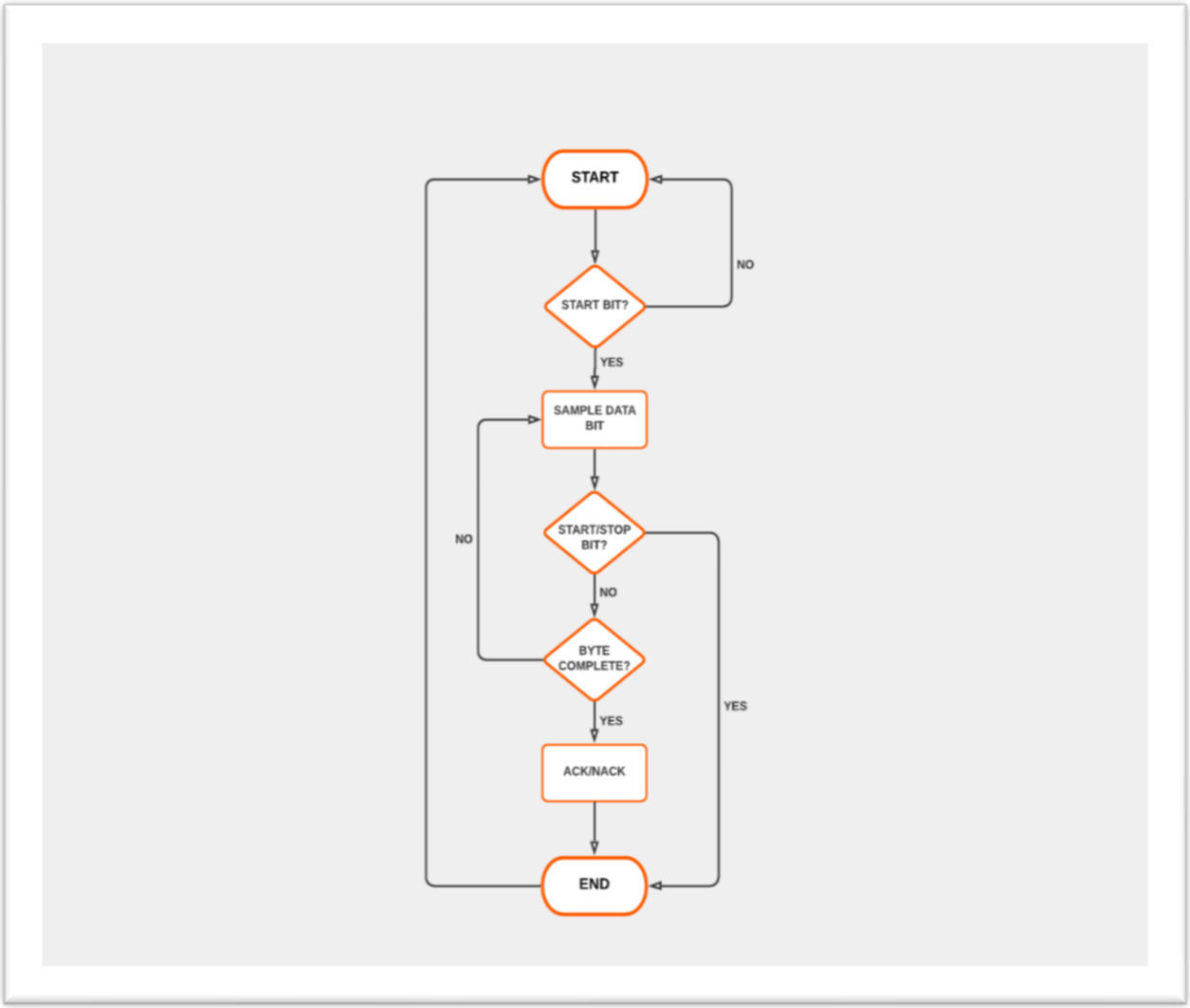
- Mobile
  - GPIO
  - LED control

- Gaming/LED Sign
  - LED Controllers

Block Diagram:



Flow Chart:



Code:

I2C Master Code

```
module I2C_Master(
        input wire clk,
        input wire rst,
        input wire [6:0] addr,
        input wire [7:0] data_in,
        input wire enable,
        input wire rw,

        //output reg [7:0] data_out,
        output wire ready,

        inout i2c_sda,
        inout wire i2c_scl
        );

        localparam IDLE = 0;
        localparam START = 1;
        localparam ADDRESS = 2;
        localparam READ_ACK = 3;
        localparam WRITE_DATA = 4;
        localparam WRITE_ACK = 5;
        localparam READ_DATA = 6;
        localparam READ_ACK2 = 7;
        localparam STOP = 8;
```

```
localparam DIVIDE_BY = 4;

        reg [7:0] state;
        reg [7:0] saved_addr;
        reg [7:0] saved_data;
        reg [7:0] counter;
        reg [7:0] counter2 = 0;
        reg write_enable;
        reg sda_out;
        reg i2c_scl_enable = 0;
        reg i2c_clk = 1;

        assign ready = ((rst == 0) && (state == IDLE)) ? 1 : 0;
        assign i2c_scl = (i2c_scl_enable == 0 ) ? 1 : i2c_clk;
        assign i2c_sda = (write_enable == 1) ? sda_out : 'bz;

        always @(posedge clk) begin
                if (counter2 == (DIVIDE_BY/2) - 1) begin
                        i2c_clk <= ~i2c_clk;
                        counter2 <= 0;
                end
                else counter2 <= counter2 + 1;
        end

        always @(negedge i2c_clk, posedge rst) begin
                if(rst == 1) begin
                        i2c_scl_enable <= 0;
                end else begin
                        if ((state == IDLE) || (state == START) || (state == STOP)) begin
                                i2c_scl_enable <= 0;
                        end else begin
```

```
                    i2c_scl_enable <= 1;
                end
            end
end


    always @(posedge i2c_clk, posedge rst) begin
        if(rst == 1) begin
            state <= IDLE;
        end
        else begin
            case(state)

                IDLE: begin
                    if (enable) begin
                        state <= START;
                        saved_addr <= {addr, rw};
                        saved_data <= data_in;
                    end
                    else state <= IDLE;
                end

                START: begin
                    counter <= 7;
                    state <= ADDRESS;
                end

                ADDRESS: begin
                    if (counter == 0) begin
                        state <= READ_ACK;
                    end else counter <= counter - 1;
```

```
                end
READ_ACK: begin
                if (i2c_sda == 0) begin
                        counter <= 7;
                        if(saved_addr[0] == 0) state <= WRITE_DATA;
                        else state <= READ_DATA;
                end else state <= STOP;
                end

                WRITE_DATA: begin
                if(counter == 0) begin
                        state <= READ_ACK2;
                end else counter <= counter - 1;
                end

                READ_ACK2: begin
                if ((i2c_sda == 0) && (enable == 1)) state <= IDLE;
                else state <= STOP;
                end

                READ_DATA: begin
                //data_out[counter] <= i2c_sda;
                if (counter == 0) state <= WRITE_ACK;
                else counter <= counter - 1;
                end
WRITE_ACK: begin
                state <= STOP;
                end


                STOP: begin
```

```
                state <= IDLE;
                        end
                endcase
        end
end

        always @(negedge i2c_clk, posedge rst) begin
                if(rst == 1) begin
                        write_enable <= 1;
                        sda_out <= 1;
                end else begin
                        case(state)

                                START: begin
                                        write_enable <= 1;
                                        sda_out <= 0;
                                end

                                ADDRESS: begin
                                        sda_out <= saved_addr[counter];
                                end

                                READ_ACK: begin
                                        write_enable <= 0;
                                end
WRITE_DATA: begin
                                        write_enable <= 1;
                                        sda_out <= saved_data[counter];
                                end
```

```
WRITE_ACK: begin
                                        write_enable <= 1;
                                        sda_out <= 0;
                                    end

                                    READ_DATA: begin
                                        write_enable <= 0;
                                    end

                                    STOP: begin
                                        write_enable <= 1;
                                        sda_out <= 1;
                                    end
                                endcase
                            end
                        end

endmodule
```

I2C Slave Code

```
module I2C_Slave(
        inout sda,
        inout scl
        );

        localparam ADDRESS = 7'b1000100;

        localparam READ_ADDR = 0;
        localparam SEND_ACK = 1;
        localparam READ_DATA = 2;
        localparam WRITE_DATA = 3;
        localparam SEND_ACK2 = 4;

        reg [7:0] addr;
        reg [7:0] counter;
        reg [7:0] state = 0;
        reg [7:0] data_in = 0;
        reg [7:0] data_out = 8'b11001100;
        reg sda_out = 0;
        reg sda_in = 0;
        reg start = 0;
        reg write_enable = 0;
assign sda = (write_enable == 1) ? sda_out : 'bz;

        always @(negedge sda) begin
                if ((start == 0) && (scl == 1)) begin
                        start <= 1;
                        counter <= 7;
```

```
end
        end

        always @(posedge sda) begin
                if ((start == 1) && (scl == 1)) begin
                        state <= READ_ADDR;
                        start <= 0;
                        write_enable <= 0;
                end
        end

        always @(posedge scl) begin
                if (start == 1) begin
                        case(state)
                                READ_ADDR: begin
                                        addr[counter] <= sda;
                                        if(counter == 0) state <= SEND_ACK;
                                        else counter <= counter - 1;

                                end

                                SEND_ACK: begin
                                        if(addr[7:1] == ADDRESS) begin
                                                counter <= 7;
                                                if(addr[0] == 0) begin
                                                        state <= READ_DATA;
                                                end
                                                else state <= WRITE_DATA;
                                end
                        end
```

```
READ_DATA: begin
                        data_in[counter] <= sda;
                        if(counter == 0) begin
                                state <= SEND_ACK2;
                        end else counter <= counter - 1;
                end

                SEND_ACK2: begin
                        state <= READ_ADDR;
                end

                WRITE_DATA: begin
                        if(counter == 0) state <= READ_ADDR;
                        else counter <= counter - 1;
                end

        endcase
    end
end

always @(negedge scl) begin
    case(state)

        READ_ADDR: begin
                write_enable <= 0;
        end

        SEND_ACK: begin
                sda_out <= 0;
                write_enable <= 1;
        end
```

```
READ_DATA: begin
                        write_enable <= 0;
                    end

                    WRITE_DATA: begin
                        sda_out <= data_out[counter];
                        write_enable <= 1;
                    end

                    SEND_ACK2: begin
                        sda_out <= 0;
                        write_enable <= 1;
                    end
                endcase
            end
endmodule
```

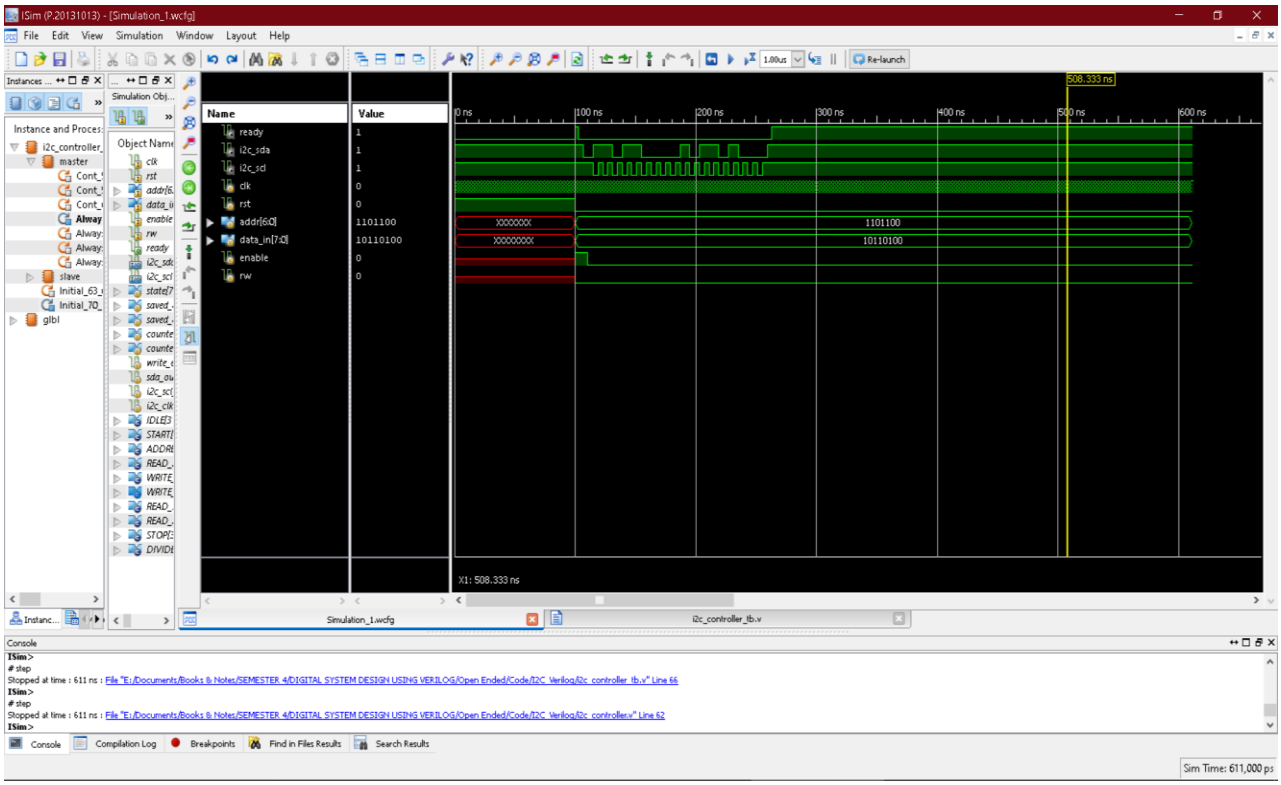
Test bench

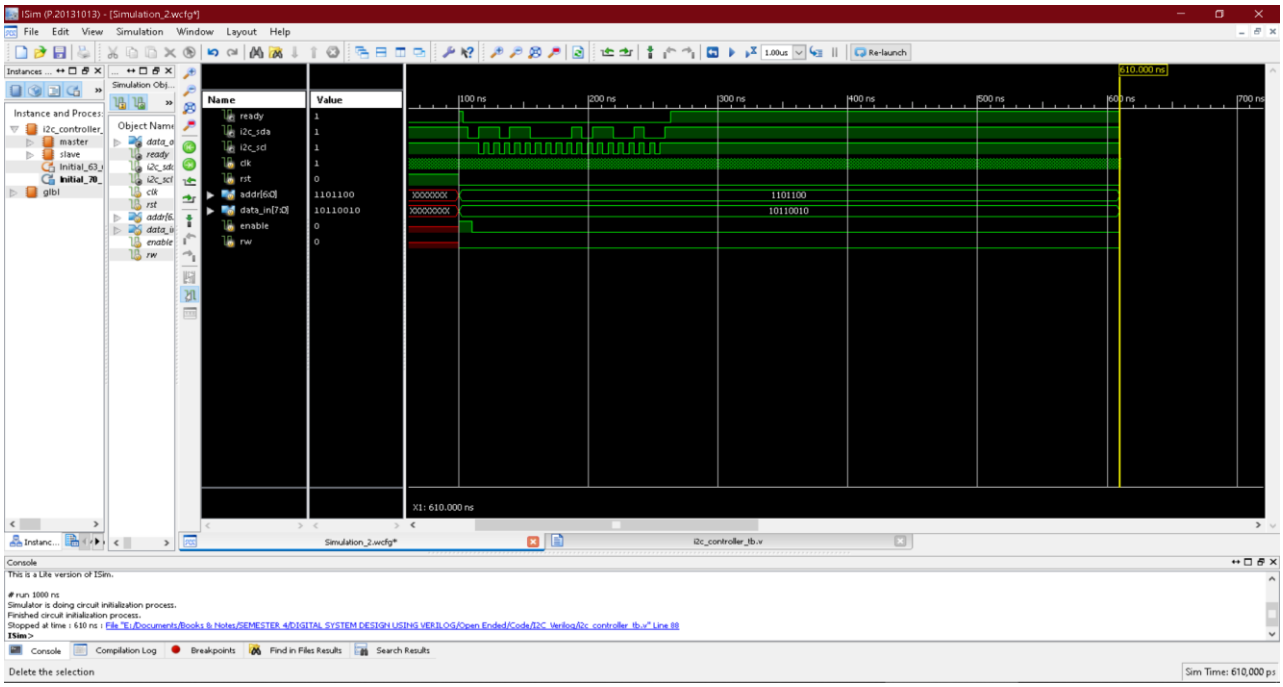
```
module I2C_TestBench;
	// Inputs
	reg clk;
	reg rst;
	reg [6:0] addr;
	reg [7:0] data_in;
	reg enable;
	reg rw;
	// Outputs
	wire [7:0] data_out;
	wire ready;
	// Bidirs
	wire i2c_sda;
	wire i2c_scl;
	// Instantiate the Unit Under Test (UUT)
	I2C_Master master (
		.clk(clk),
		.rst(rst),
		.addr(addr),
		.data_in(data_in),
		.enable(enable),
		.rw(rw),
		//.data_out(data_out),
		.ready(ready),
		.i2c_sda(i2c_sda),
		.i2c_scl(i2c_scl)
	);
```

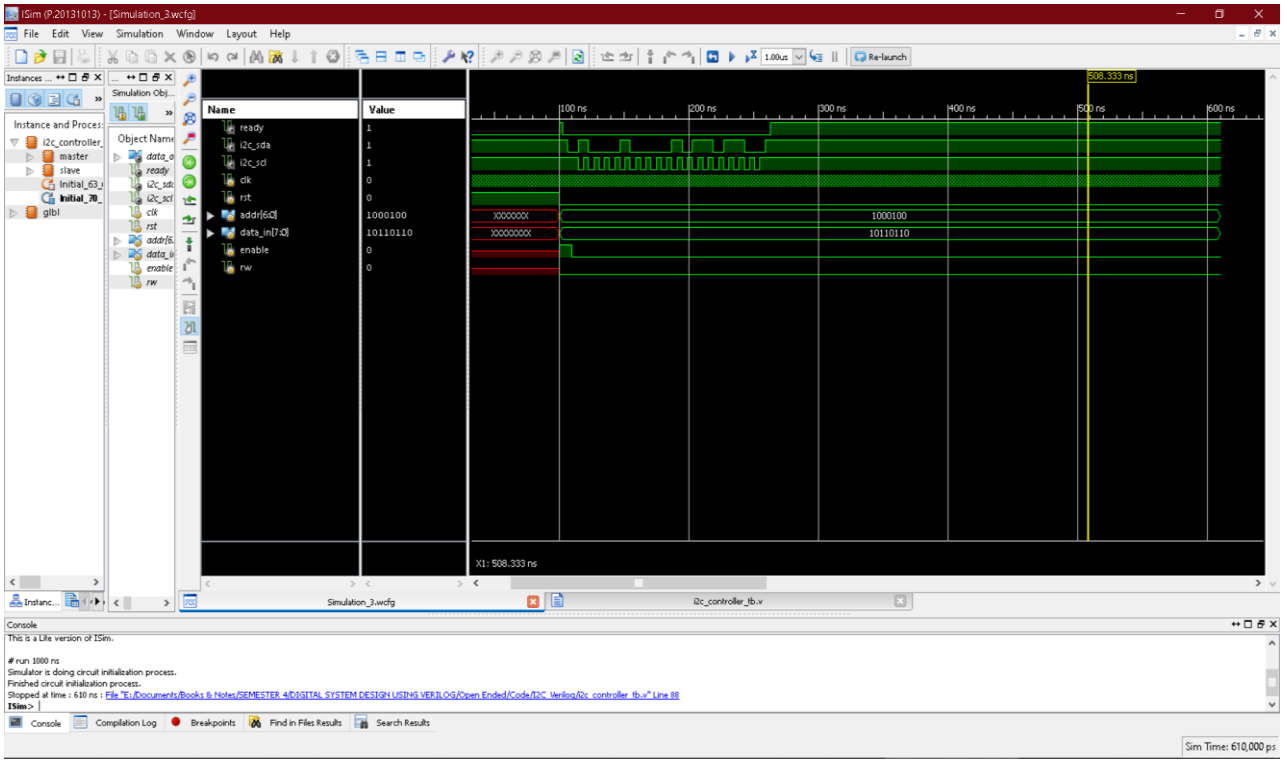
```
I2C_Slave slave (
   .sda(i2c_sda),
   .scl(i2c_scl)
   );

        initial begin
                clk = 0;
                forever begin
                        clk = #1 ~clk;
                end
        end

        initial begin
                // Initialize Inputs
                clk = 0;
                rst = 1;

                // Wait 100 ns for global reset to finish
                #100;

                // Add stimulus here
                rst = 0;
                addr = 7'b1000100;
                data_in = 8'b11110110;
                rw = 0;
                enable = 1;
                #10;
                enable = 0;
                                        #500
                $finish;
```

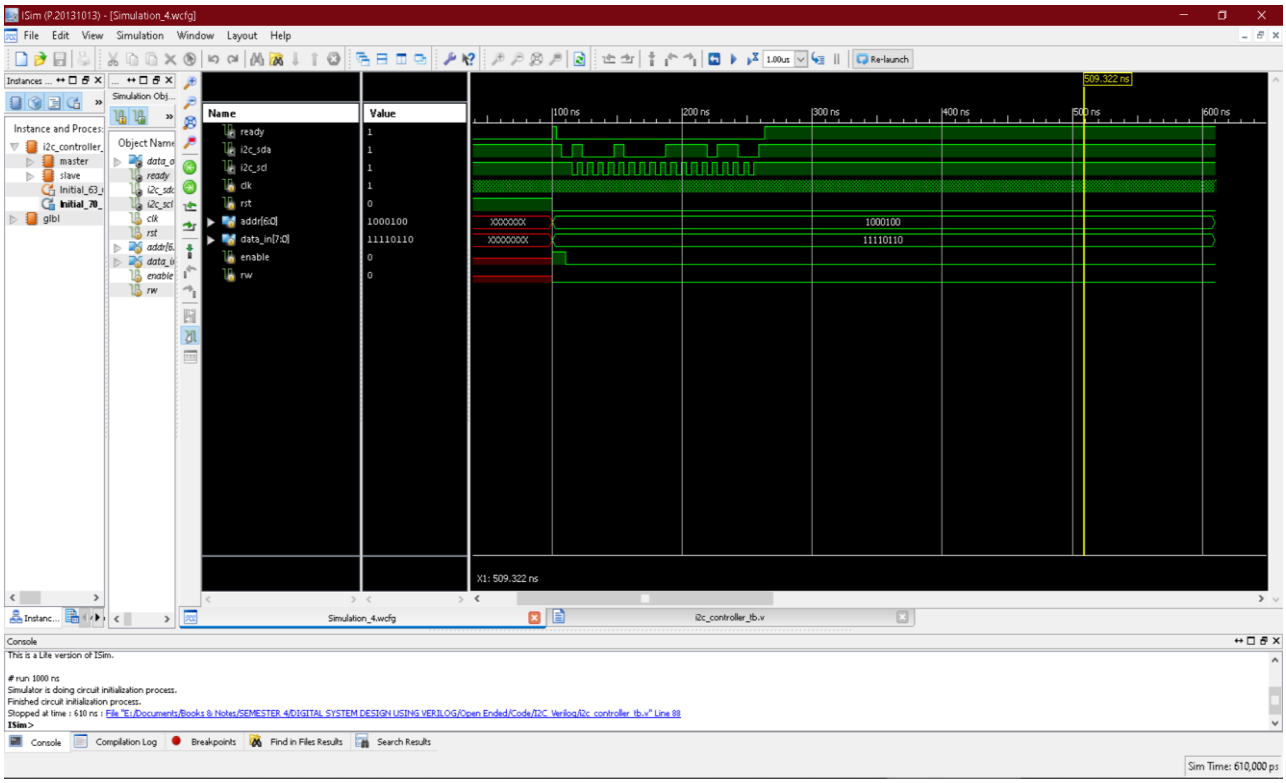
```
        end

endmodule
```

Results:






ISim (P.20131013) - [Simulation_3.wcfg]
File Edit View Simulation Window Layout Help
Re-launch
Name
Value
ready
i2c_sda
i2c_scl
clk
rst
addr[6:0]
1000100
data_in[7:0]
10110110
enable
rw
506.333 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
X1: 508.333 ns
Simulation_3.wcfg
i2c_controller_tb.v
Console
This is a Lite version of ISim.
run 1000 ns
Simulator is doing circuit initialization process.
Finished circuit initialization process.
Stopped at time : 610 ns : File "E:/Documents/Books & Notes/SEMESTER 4/DIGITAL SYSTEM DESIGN USING VERILOG/Open Ended/Code/I2C_Verilog/i2c_controller_tb.v" Line 88
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 610,000 ps


ISim (P.20131013) - [Simulation_4.wcfg]
File Edit View Simulation Window Layout Help
Re-launch
Name
Value
ready
i2c_sda
i2c_scl
clk
rst
addr[6:0]
1000100
data_in[7:0]
11110110
enable
rw
509.322 ns
100 ns
200 ns
300 ns
400 ns
500 ns
600 ns
X1: 509.322 ns
Simulation_4.wcfg
i2c_controller_tb.v
Console
This is a Lite version of ISim.
run 1000 ns
Simulator is doing circuit initialization process.
Finished circuit initialization process.
Stopped at time : 610 ns : File "E:/Documents/Books & Notes/SEMESTER 4/DIGITAL SYSTEM DESIGN USING VERILOG/Open Ended/Code/I2C_Verilog/i2c_controller_tb.v" Line 88
ISim>
Console
Compilation Log
Breakpoints
Find in Files Results
Search Results
Sim Time: 610,000 ps

Conclusion:

In the development and implementation of the SPI and I2C protocols into a FPGA, the following conclusions were obtained:

- Verilog is a high-level programming language that runs concurrently, to the difference with other programming languages that work sequentially as those used by microcontrollers, so it performs a faster and more efficient communication when implementing the SPI and I2C communication protocols.
- The developed code for the implementation of the I2C protocol is more robust and complex compared to the developed for the SPI protocol, due to the number of events to take under consideration in communicating devices such as the start and the stop conditions, bit recognition, and read/write data on bidirectional lines.
- It is needed to set at least two operating conditions for input and output data in, each device acting as master or slave for a correct reading and / or writing process between them, because it is not possible to ensure an ideal behaviour in the rising and falling edges of the signals, which are necessary for the execution of the code.
- A tri-stated buffer is implemented in the I2C slave for reading and writing data to the bidirectional SDA line, besides the high impedance statehood protects the FPGA peripherals against the phase shift operation signals.
- In the I2C slave, it is necessary to consider the lag introduced in the SDA and SCL signals at the time of writing the data, because the reading process occurs in the rising edge and the writing process occurs in the falling edge of the clock signal, so that these matches the structure of the data line SDA.