INDEX

AIM:

To demonstrate the I2C protocol with 32-bit data transfer and 7 bit address along with acknowledgements and read/write enable for 1-Master and 4-slaves using Verilog code.

APPARATUS:

Xilinx ISE design suite.

THEORY:

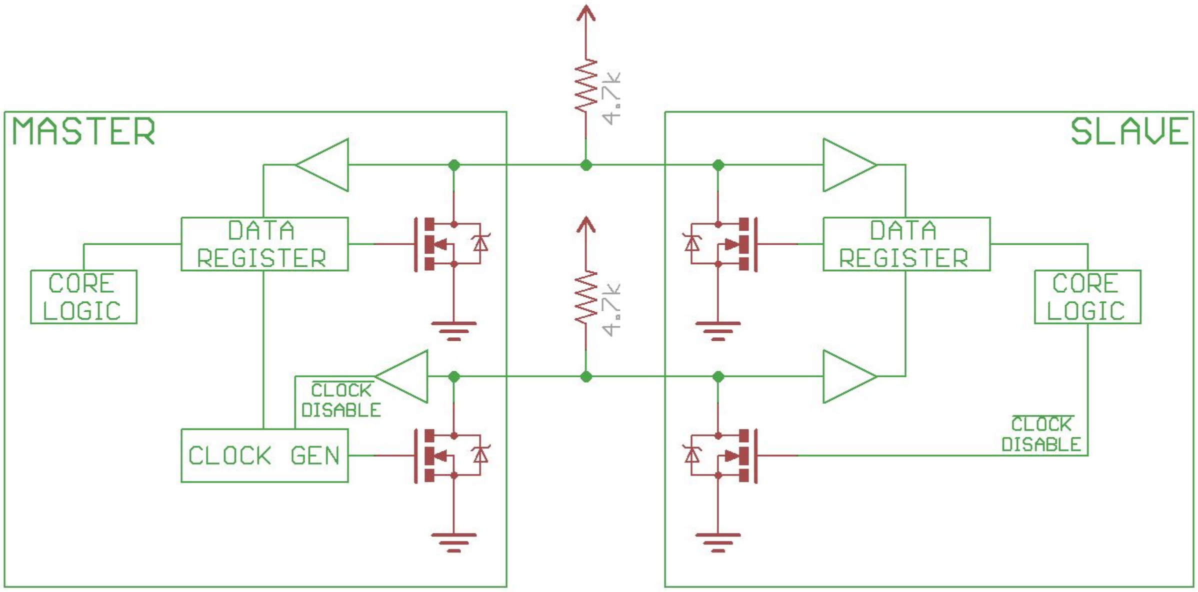
I2C - A Brief History

I2C was originally developed in 1982 by Philips for various Philips chips. The original spec allowed for only 100kHz communications, and provided only for 7-bit addresses, limiting the number of devices on the bus to 112 (there are several reserved addresses, which will never be used for valid I2C addresses). In 1992, the first public specification was published, adding a 400kHz fast-mode as well as an expanded 10-bit address space. Much of the time (for instance, in the ATMega328 device on many Arduino-compatible boards) , device support for I2C ends at this point. There are three additional modes specified: fast-mode plus, at 1MHz; high-speed mode, at 3.4MHz; and ultra-fast mode, at 5MHz.

I2C at the Hardware Level

Each I2C bus consists of two signals: SCL and SDA. SCL is the clock signal, and SDA is the data signal. The clock signal is always generated by the current bus master; some slave devices may force the clock low at times to delay the master sending more data (or to require more time to prepare data before the master attempts to clock it out). This is called "clock stretching" and is described on the protocol page.

Unlike UART or SPI connections, the I2C bus drivers are "open drain", meaning that they can pull the corresponding signal line low, but cannot drive it high. Thus, there can be no bus contention where one device is trying to drive the line high while another tries to pull it low, eliminating the potential for damage to the drivers or excessive power dissipation in the system. Each signal line has a pull-up resistor on it, to restore the signal to high when no device is asserting it low.



Notice the two pull-up resistors on the two communication lines.

Resistor selection varies with devices on the bus, but a good rule of thumb is to start with 4.7k and adjust down if necessary. I2C is a fairly robust protocol, and can be used with short runs of wire (2-3m). For long runs, or systems with lots of devices, smaller resistors are better.

Since the devices on the bus don't actually drive the signals high, I2C allows for some flexibility in connecting devices with different I/O voltages. In general, in a system where one device is at a higher voltage than another, it may be possible to connect the two devices via I2C without any level shifting circuitry in between them. The trick is to connect the pull-up resistors to the lower of the two voltages. This only works in some cases, where the lower of the two system voltages exceeds the high-level input voltage of the higher voltage system--for example, a 5V Arduino and a 3.3V accelerometer.

Advantageous comparison between SPI, UART & I2C:

UART	SPI	I2C
It is simple communication and most popular which is available due to UART support in almost all the devices with 9 pin connector. It is also referred to as RS232 interface.	•It is a simple protocol and hence does not require processing overheads. •Supports full duplex communication. •Due to separate use of CS lines, the same kind of multiple chips can be used in the circuit design. •SPI uses push-pull and hence higher data rates and longer ranges are possible. •SPI uses less power compare to I2C	Due to open collector design, limited slew rates can be achieved. •More than one master can be used in the electronic circuit design. •Needs fewer i.e. only 2 wires for communication. •I2C addressing is simple which does not require any CS lines used in SPI and it is easy to add extra devices on the bus. •It uses the open collector bus concept. Hence there is bus voltage flexibility on the interface bus. •Uses flow control.

Disadvantageous comparison:

UART	SPI	I2C
They are suitable for communication between only two devices. • It supports fixed data rate agreed upon between devices initially before communication otherwise data will be garbled.	As the number of slaves increases, the number of CS lines increases, this results in hardware complexity as the number of pins required will increase. • To add a device in SPI requires one to add an extra CS line and changes in software for particular device addressing is concerned. •Master and slave relationships can not be changed as usually done in I2C interface. •No flow control available in SPI.	Increases complexity of the circuit when number of slaves and masters increases. •The I2C interface is half duplex. •Requires software stack to control the protocol and hence it needs some processing overheads on microcontroller/microprocessor.

Features:

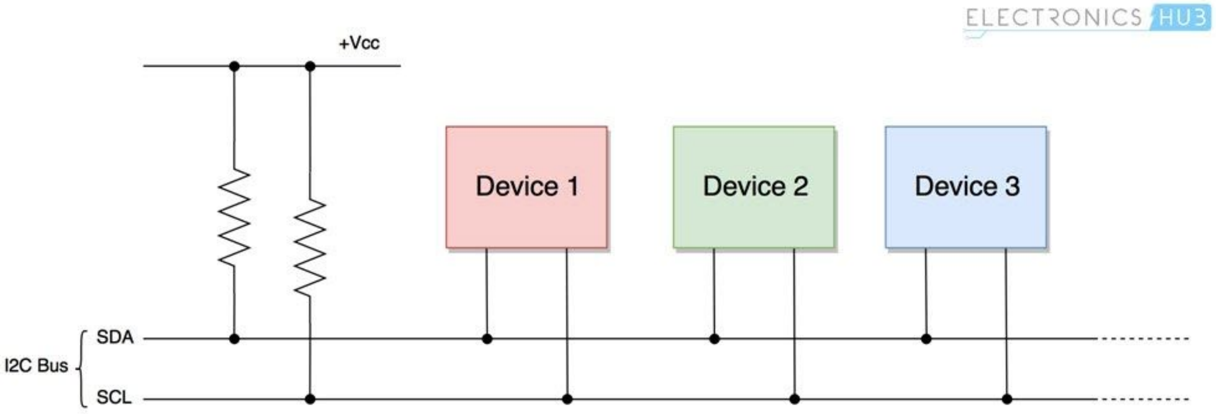
The following are some of the important features of I2C communication protocol:

- Only two common bus lines (wires) are required to control any device/IC on the I2C network
- No need of prior agreement on data transfer rate like in UART communication. So the data transfer speed can be adjusted whenever required
- Simple mechanism for validation of data transferred
- Uses 7-bit addressing system to target a specific device/IC on the I2C bus
- I2C networks are easy to scale. New devices can simply be connected to the two common I2C bus lines

Hardware

The physical I2C Bus

I2C Bus (Interface wires) consists of just two wires and are named as Serial Clock Line (SCL) and Serial Data Line (SDA). The data to be transferred is sent through the SDA wire and is synchronized with the clock signal from SCL. All the devices/ICs on the I2C network are connected to the same SCL and SDA lines as shown below:



Both the I2C bus lines (SDA, SCL) are operated as open drain drivers. It means that any device/IC on the I2C network can drive SDA and SCL low, but they cannot drive them high. So, a pull up resistor is used for each bus line, to keep them high (at positive voltage) by default.

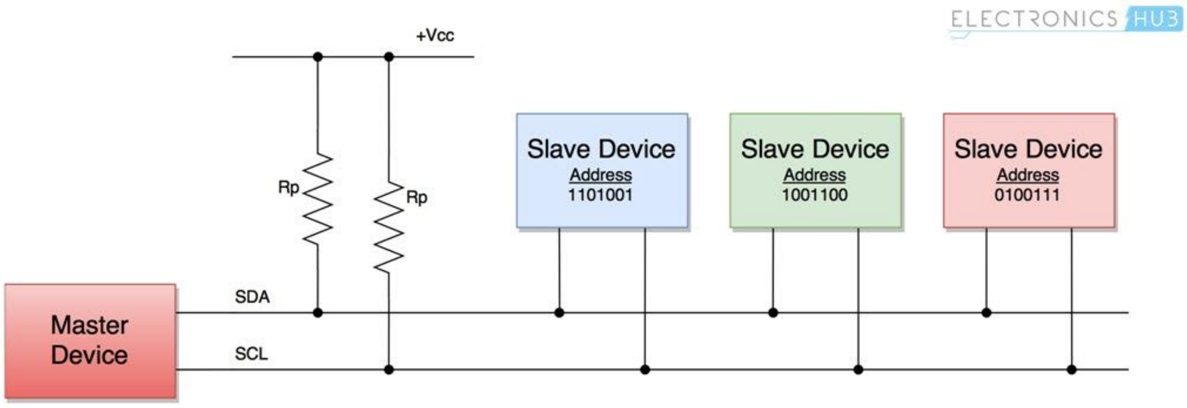
The reason for using an open-drain system is that there will be no chances of shorting, which might happen when one device tries to pull the line high and some other device tries to pull the line low.

Master and Slave Devices

The devices connected to the I2C bus are categorized as either masters or slaves. At any instant of time only a single master stays active on the I2C bus. It controls the SCL clock line and decides what operation is to be done on the SDA data line.

All the devices that respond to instructions from this master device are slaves. For differentiating between multiple slave devices connected to the same I2C bus, each slave device is physically assigned a permanent 7-bit address.

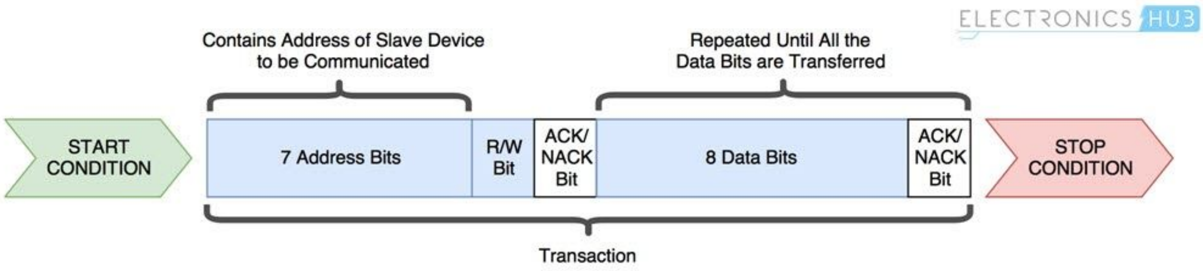
When a master device wants to transfer data to or from a slave device, it specifies this particular slave device address on the SDA line and then proceeds with the transfer. So effectively communication takes place between the master device and a particular slave device.All the other slave devices don't respond unless their address is specified by the master device on the SDA line.



Data Transfer Protocol

The following protocol (set of rules) is followed by master device and slave devices for the transfer of data between them.

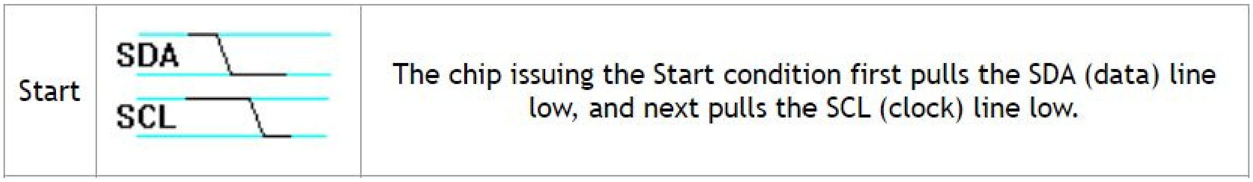
Data is transferred between the master device and slave devices through a single SDA data line, via patterned sequences of 0's and 1's (bits). Each sequence of 0's and 1's is termed as a transaction and the data in each transaction is structured as below:



Start Condition

Whenever a master device/IC decides to start a transaction, it switches the SDA line from high voltage level to a low voltage level before the SCL line switches from high to low.

Once a start condition is sent by the master device, all the slave devices get active even if they are in sleep mode, and wait for the address bits.



Start	SDA SCL	The chip issuing the Start condition first pulls the SDA (data) line low, and next pulls the SCL (clock) line low.

Address Block: It comprises 7 bits and is filled with the address of the slave device to/from which the master device needs send/receive data. All the slave devices on the I2C bus compare these address bits with their address.

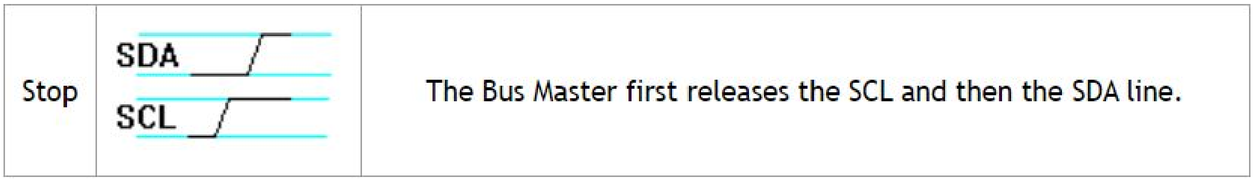
Read/Write Bit: This bit specifies the direction of data transfer. If the master device/IC needs to send data to a slave device, this bit is set to ‘1’. If the master IC needs to receive data from the slave device, it is set to ‘0’.

ACK/NACK Bit: It stands for Acknowledged/Not-Acknowledged bit. If the physical address of any slave device coincides with the address broadcasted by the master device, the value of this bit is set to ‘1’ by the slave device. Otherwise it remains at logic ‘0’ (default).

Data Block: It comprises 8 bits(can be adjusted) and they are set by the sender, with the data bits it needs to transfer to the receiver. This block is followed by an ACK/NACK bit and is set to ‘1’ by the receiver if it successfully receives data. Otherwise it stays at logic ‘0’.

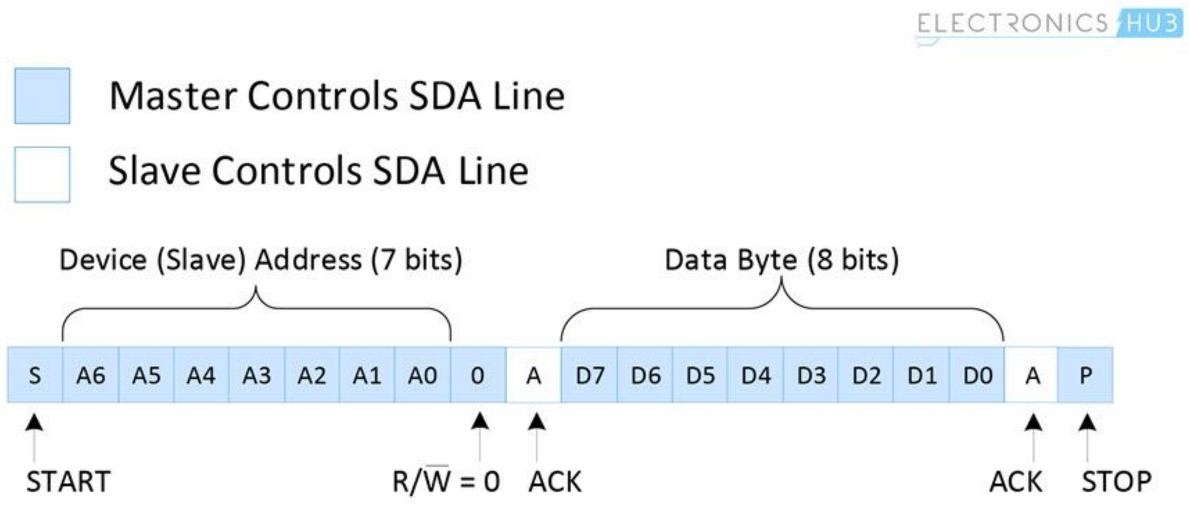
This combination of data blocks followed by ACK/NACK bit is repeated until the data is completely transferred.

Stop Condition: After required data blocks are transferred through the SDA line, the master device switches the SDA line from low voltage level to high voltage level before the SCL line switches from high to low.



Stop	SDA SCL	The Bus Master first releases the SCL and then the SDA line.

NOTE: Logic ‘0’ or setting a bit to ‘0’ is equivalent to applying low voltage on the SDA line and vice versa.



LOOK OF FRAME

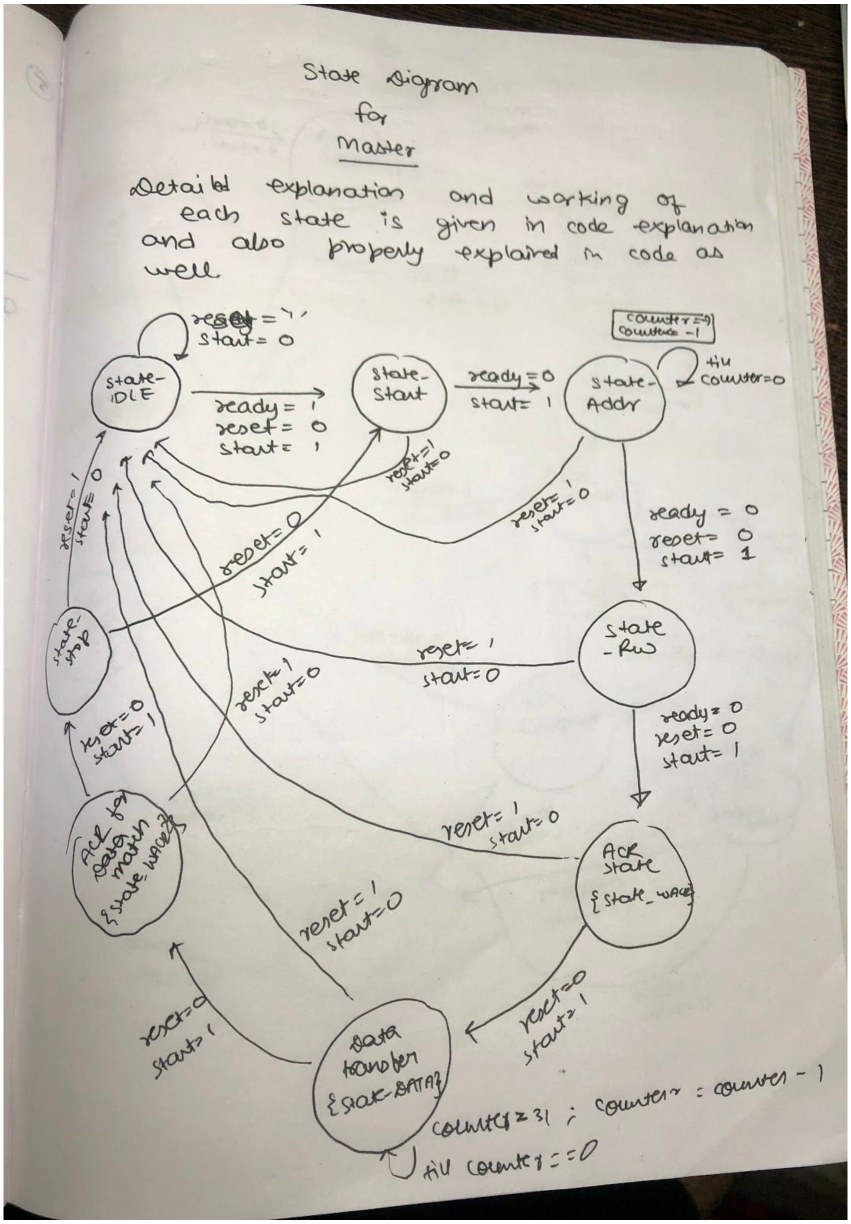
Here, instead of 8-bits data transfer , I have dedicated it for 32 bit data transfer with 7-bit address and acknowledgements, read/write enable along with start and stop bit.

Procedure :

1. The main block or module consists of two sub parts in the same module named code, which has one master and 4 slaves.
2. Each slave has unique addresses and already predefined in the code module.
3. Using Finite State Machine , the code block has been implemented.
   What is FSM:
   Finite State Machines (FSM) are sequential circuits used in many digital systems to control the behavior of systems and data flow paths. Examples of FSM include control units and sequencers.

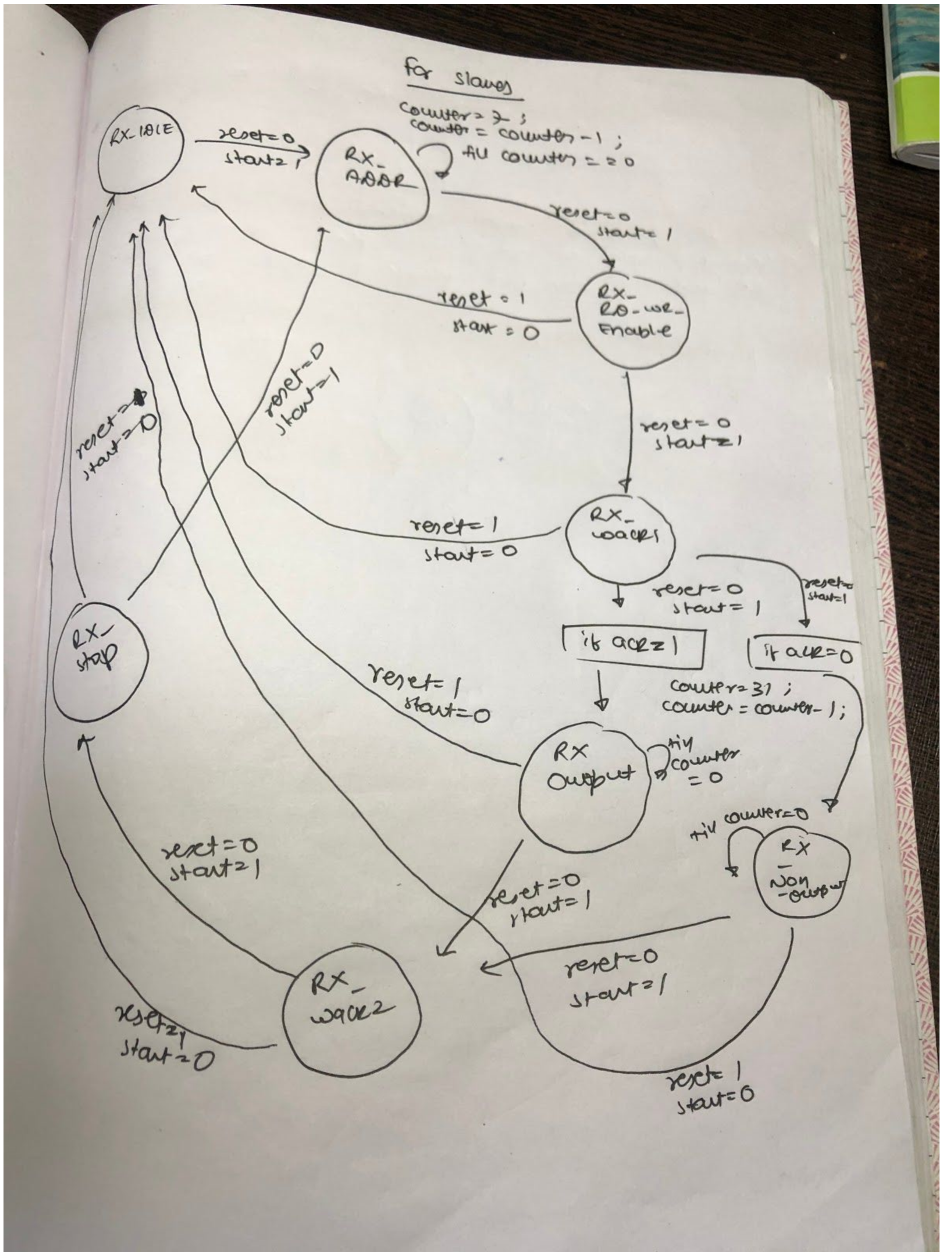
   1. **Mealy FSM :**A finite-state machine (FSM) or simply a state machine is used to design both computer programs and sequential logic circuits. It is conceived as an abstract machine that can be in one of a finite number of user-defined states. The machine is in only one state at a time; the state it is in at any given time is called the current state. It can change from one state to another when initiated by a triggering event or condition; this is called a transition. A particular FSM is defined by a list of its states, and the triggering condition for each transition.

2. **Moore FSM :** Its output is generated from the state register block. The next state is determined using the present (current) input and the present (current) state. Here the state register is also modeled using D flip-flops. Normally Moore machines are described using three blocks, one of which must be sequential and the other two can be modeled using always blocks or a combination of always and dataflow modeling constructs.



State diagram for Master

Detailed explanation and working of each state is given in code explanation and also properly explained in code as well



4. Proper test bench named *"test.v"* have been written with all conditions to verify that when:

a. When a given address doesn't match with any defined address of any slave.
b. When a given address matches with a particular address of slave 1.
c. When a given different address matches with a particular address of slave 2.
d. In between reset and start =0/1 to verify all cases.

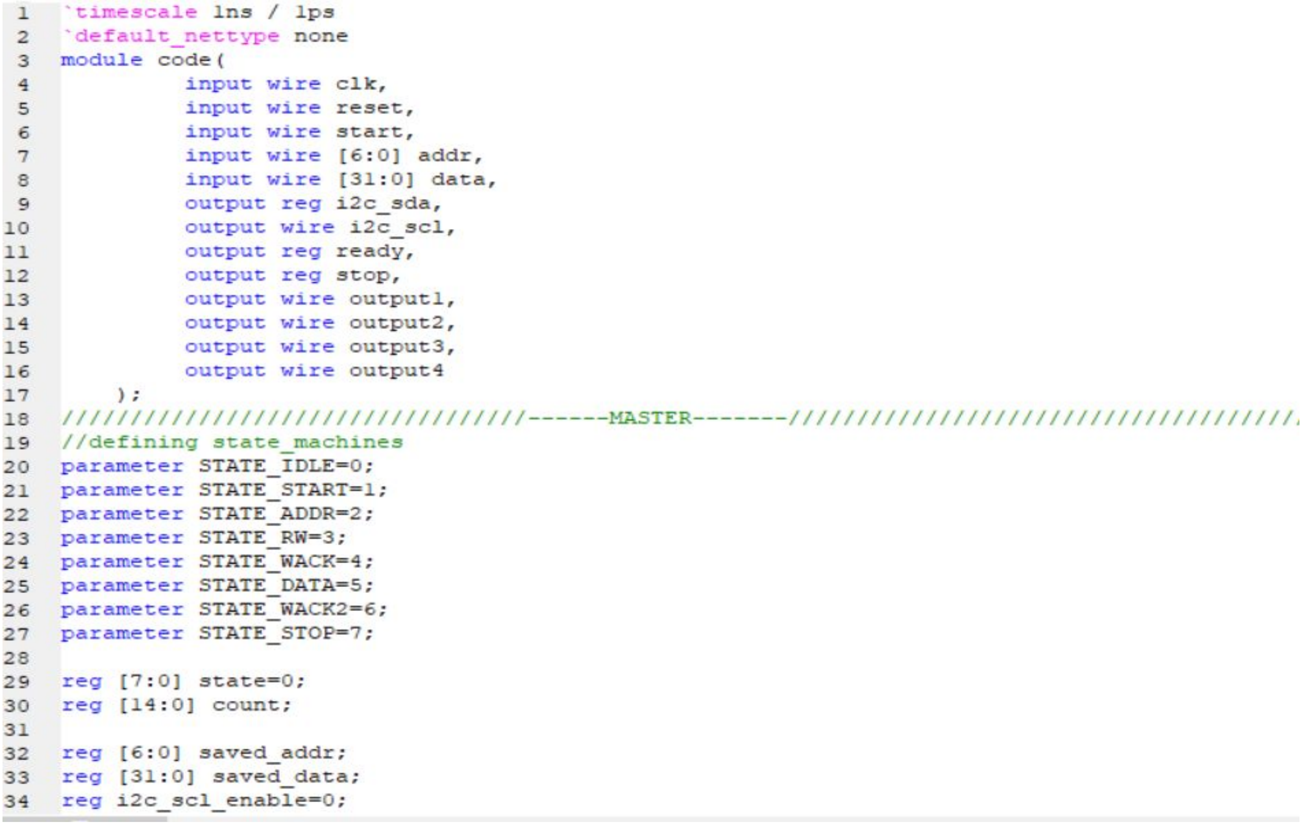
Specification and Assumption:

Since to model a protocol using HDL, certain assumptions have been made to realise without hardware.

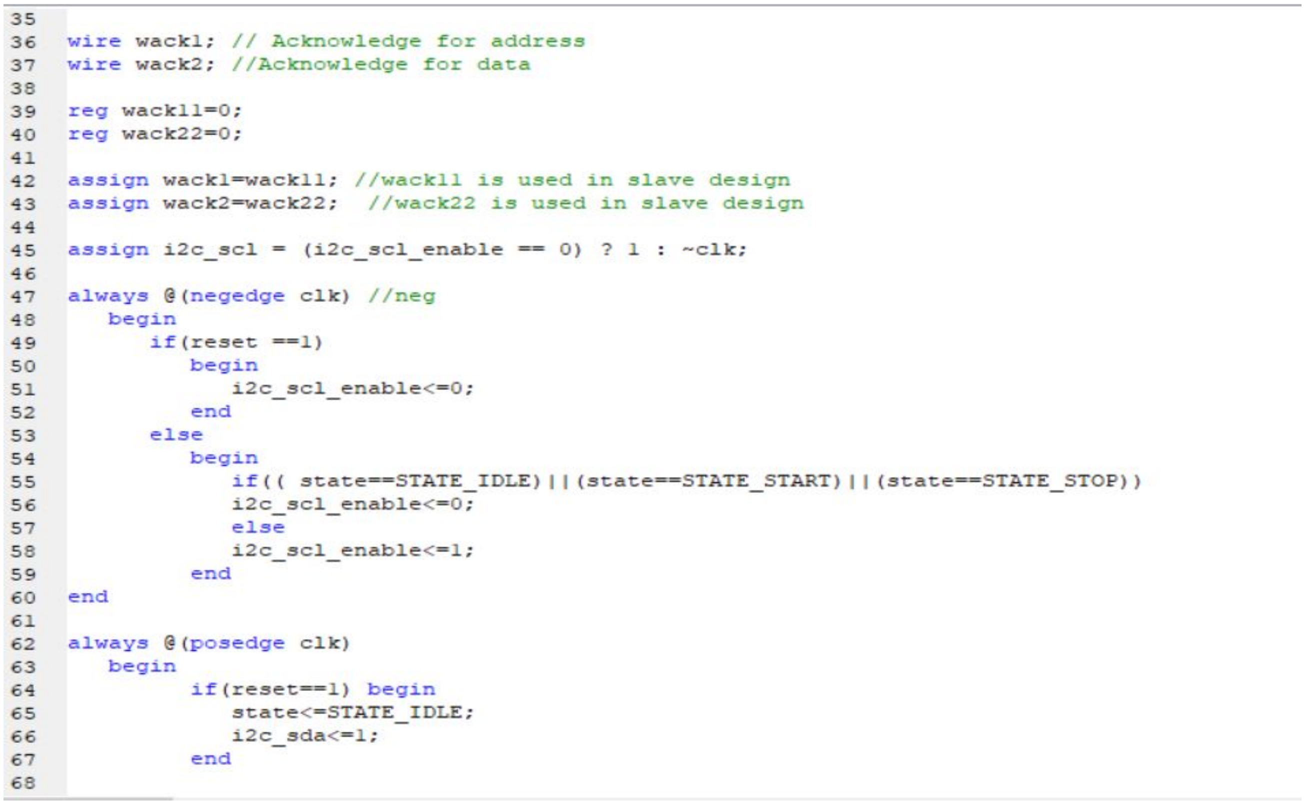
1. It has only 1 -Master and it guides 4- slaves.
2. Always read enable, i.e. always output terminals shows output (if it has to, i.e. when condition fulfills) by reading from the test bench or can be said as given by the master.
3. Here clock frequency can be adjusted to any level as the FSM works on triggering the clock. However in practical scenario I2C protocol as like most of the protocol has a certain range of operating frequency to work upon and beyond that it may fail to operate properly. However it may be adjusted to any level in order to make output look easier to visualize. (Practical Value: **The initial I2C specifications defined maximum clock frequency of 100 kHz. This was later increased to 400 kHz as Fast mode. There is also a High speed mode which can go up to 3.4 MHz and there is also a 5 MHz ultra-fast mode.**)

Verilog code:

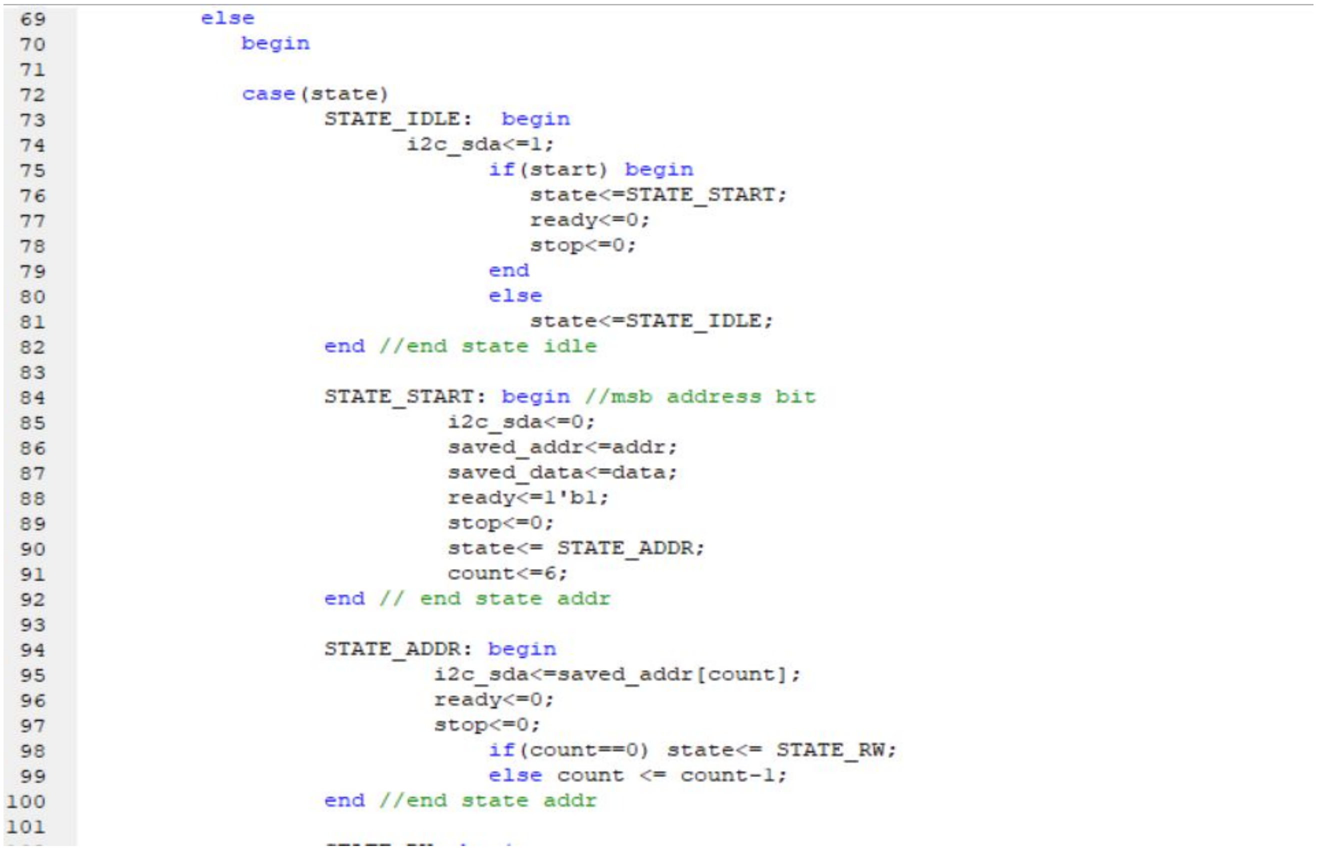
(I have taken a snip of code and added pictures instead of code, to make alignment compatible with the report documentation. Code and test bench has been submitted along with report submission.)



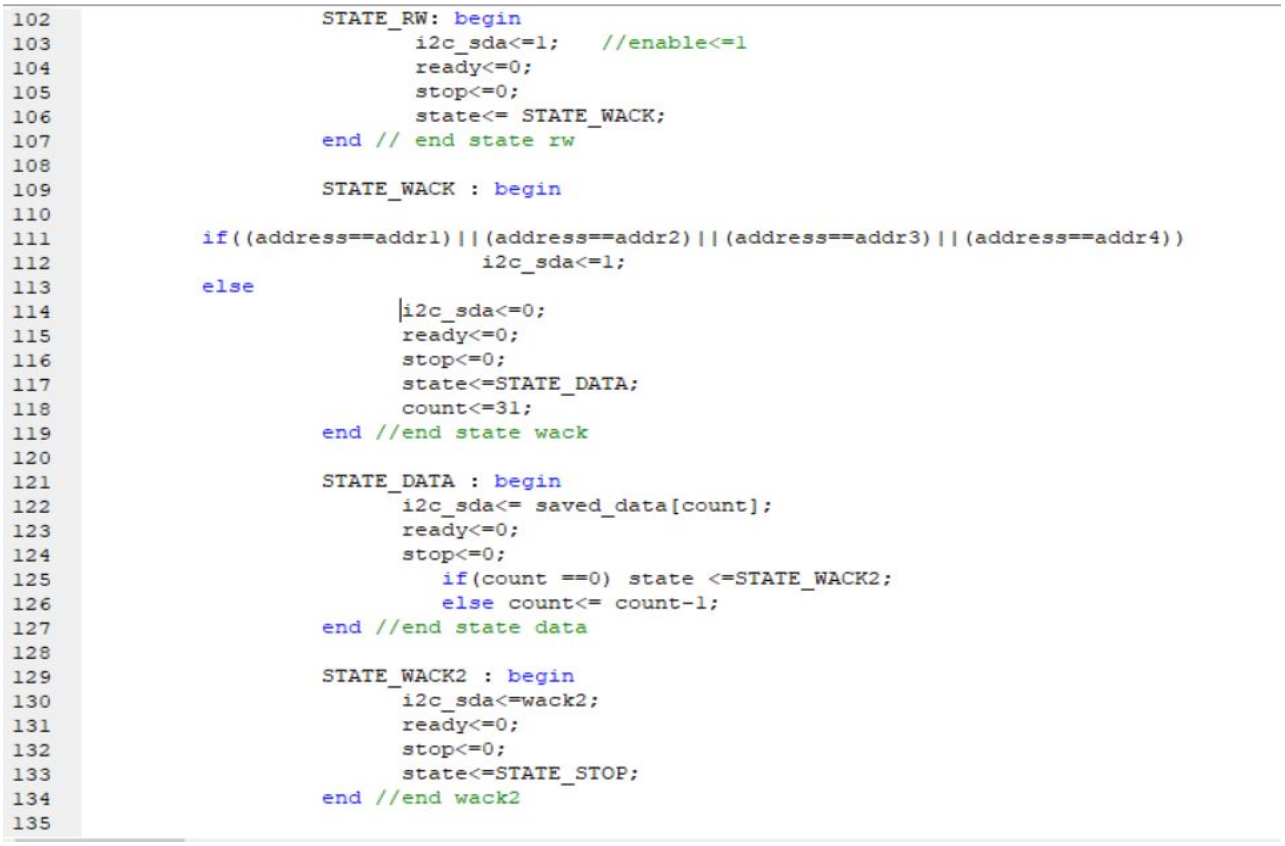
```
`timescale 1ns / 1ps
`default_nettype none
module code(
         input wire clk,
         input wire reset,
         input wire start,
         input wire [6:0] addr,
         input wire [31:0] data,
         output reg i2c_sda,
         output wire i2c_scl,
         output reg ready,
         output reg stop,
         output wire output1,
         output wire output2,
         output wire output3,
         output wire output4
    );
//////////////////////////////////data------MASTER-------/////////////////////////////////////
//defining state_machines
parameter STATE_IDLE=0;
parameter STATE_START=1;
parameter STATE_ADDR=2;
parameter STATE_RW=3;
parameter STATE_WACK=4;
parameter STATE_DATA=5;
parameter STATE_WACK2=6;
parameter STATE_STOP=7;

reg [7:0] state=0;
reg [14:0] count;

reg [6:0] saved_addr;
reg [31:0] saved_data;
reg i2c_scl_enable=0;
```



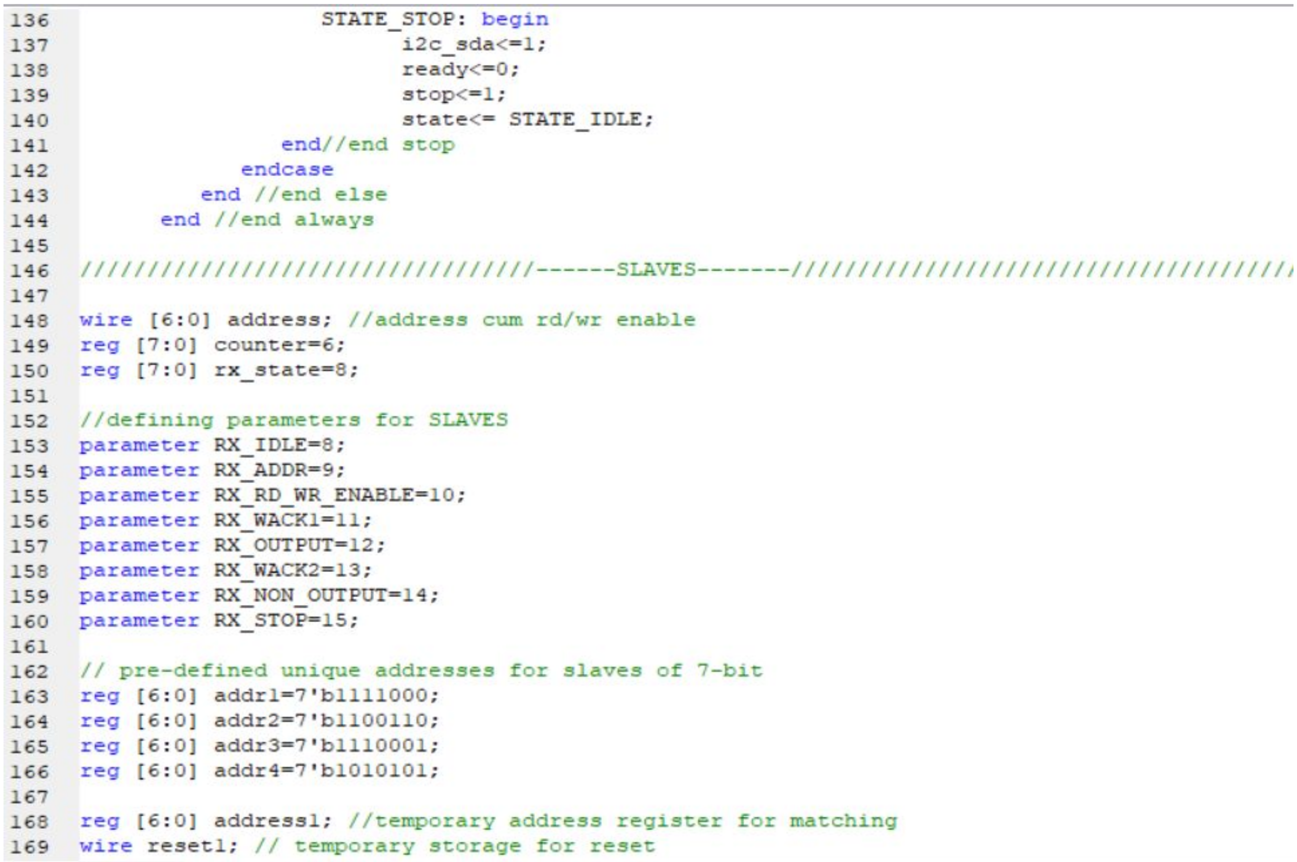
```

wire wack1; // Acknowledge for address
wire wack2; //Acknowledge for data

reg wack11=0;
reg wack22=0;

assign wack1=wack11; //wack11 is used in slave design
assign wack2=wack22;  //wack22 is used in slave design

assign i2c_scl = (i2c_scl_enable == 0) ? 1 : ~clk;

always @(negedge clk) //neg
   begin
      if(reset ==1)
         begin
            i2c_scl_enable<=0;
         end
      else
         begin
            if(( state==STATE_IDLE)||(state==STATE_START)||(state==STATE_STOP))
            i2c_scl_enable<=0;
            else
            i2c_scl_enable<=1;
         end
end

always @(posedge clk)
   begin
         if(reset==1) begin
            state<=STATE_IDLE;
            i2c_sda<=1;
         end

```



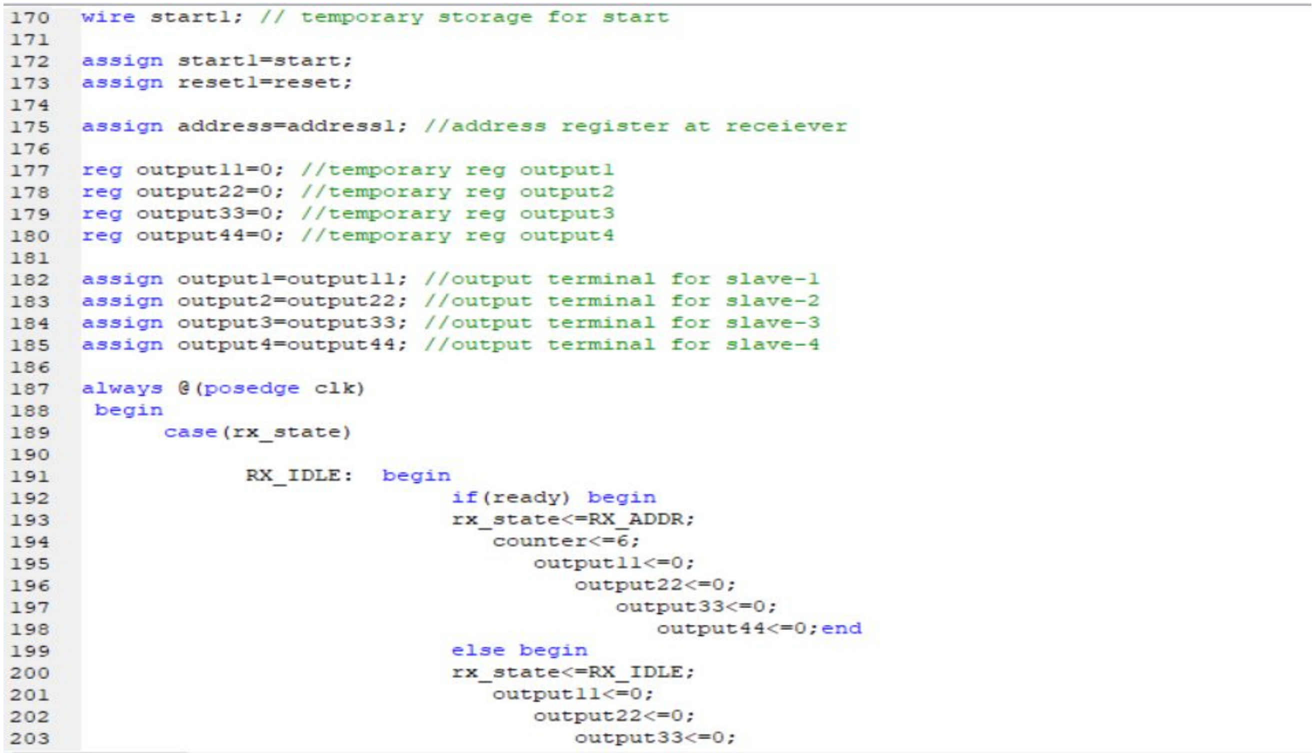
```
else
   begin

   case(state)
         STATE_IDLE:  begin
               i2c_sda<=1;
                     if(start) begin
                        state<=STATE_START;
                        ready<=0;
                        stop<=0;
                     end
                     else
                        state<=STATE_IDLE;
         end //end state idle

         STATE_START: begin //msb address bit
                  i2c_sda<=0;
                  saved_addr<=addr;
                  saved_data<=data;
                  ready<=1'b1;
                  stop<=0;
                  state<= STATE_ADDR;
                  count<=6;
         end // end state addr

         STATE_ADDR: begin
                 i2c_sda<=saved_addr[count];
                 ready<=0;
                 stop<=0;
                     if(count==0) state<= STATE_RW;
                     else count <= count-1;
         end //end state addr

```



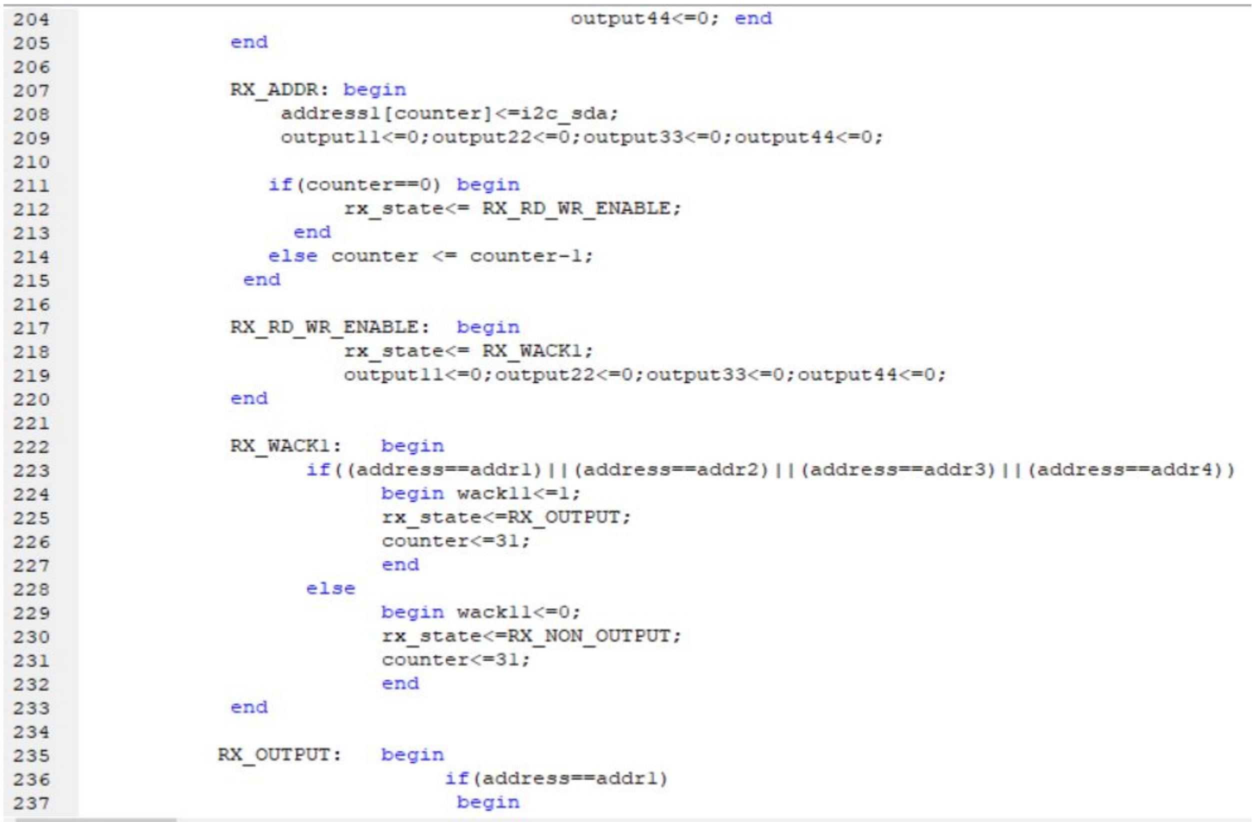
```
         STATE_RW: begin
                i2c_sda<=1;   //enable<=1
                ready<=0;
                stop<=0;
                state<= STATE_WACK;
         end // end state rw

         STATE_WACK : begin

if((address==addr1)||(address==addr2)||(address==addr3)||(address==addr4))
                    i2c_sda<=1;
else
               i2c_sda<=0;
               ready<=0;
               stop<=0;
               state<=STATE_DATA;
               count<=31;
         end //end state wack

         STATE_DATA : begin
               i2c_sda<= saved_data[count];
               ready<=0;
               stop<=0;
                  if(count ==0) state <=STATE_WACK2;
                  else count<= count-1;
         end //end state data

         STATE_WACK2 : begin
               i2c_sda<=wack2;
               ready<=0;
               stop<=0;
               state<=STATE_STOP;
         end //end wack2

```



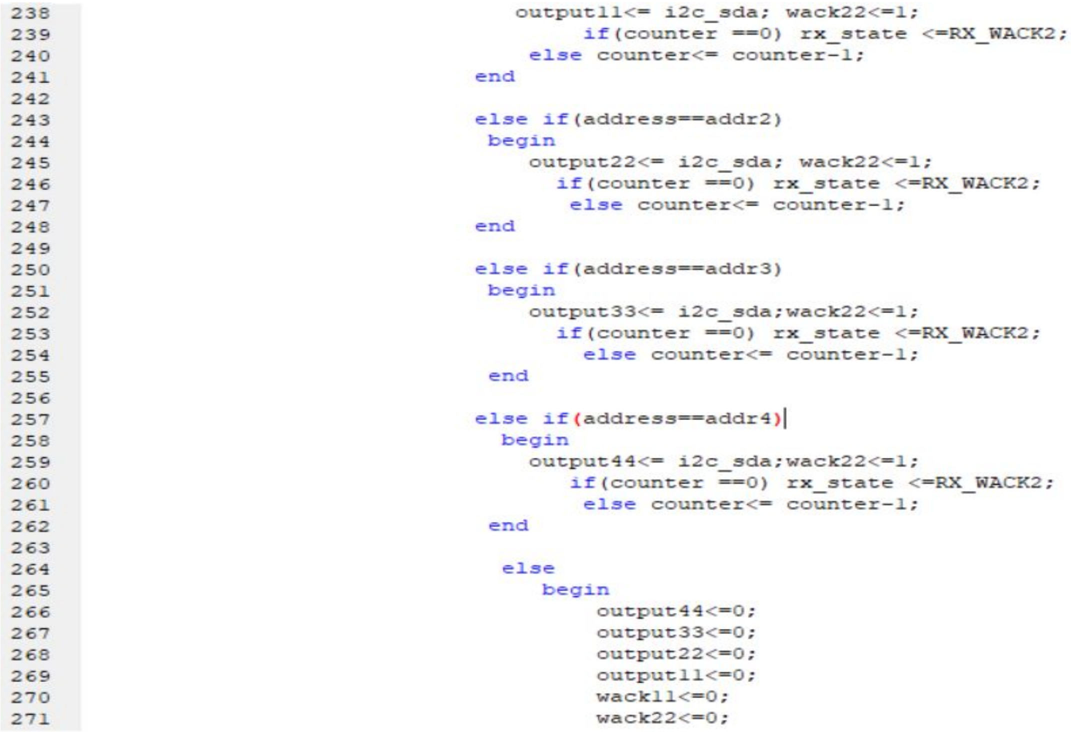
```
                  STATE_STOP: begin
                        i2c_sda<=1;
                        ready<=0;
                        stop<=1;
                        state<= STATE_IDLE;
               end//end stop
            endcase
         end //end else
      end //end always

//////////////////////////////////------SLAVES-------/////////////////////////////////////

wire [6:0] address; //address cum rd/wr enable
reg [7:0] counter=6;
reg [7:0] rx_state=8;

//defining parameters for SLAVES
parameter RX_IDLE=8;
parameter RX_ADDR=9;
parameter RX_RD_WR_ENABLE=10;
parameter RX_WACK1=11;
parameter RX_OUTPUT=12;
parameter RX_WACK2=13;
parameter RX_NON_OUTPUT=14;
parameter RX_STOP=15;

// pre-defined unique addresses for slaves of 7-bit
reg [6:0] addr1=7'b1111000;
reg [6:0] addr2=7'b1100110;
reg [6:0] addr3=7'b1110001;
reg [6:0] addr4=7'b1010101;

reg [6:0] address1; //temporary address register for matching
wire reset1; // temporary storage for reset
```



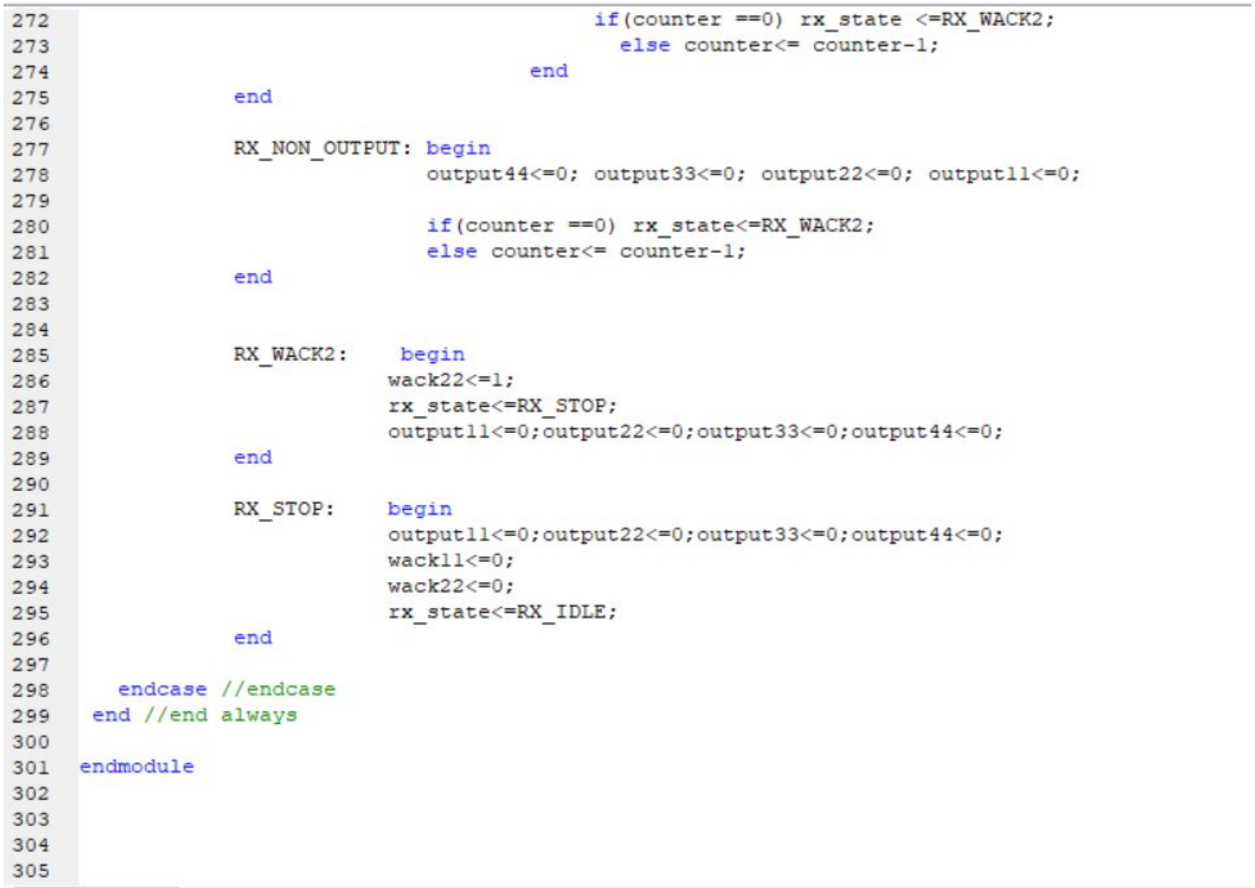
```
wire start1; // temporary storage for start

assign start1=start;
assign reset1=reset;

assign address=address1; //address register at receiever

reg output11=0; //temporary reg output1
reg output22=0; //temporary reg output2
reg output33=0; //temporary reg output3
reg output44=0; //temporary reg output4

assign output1=output11; //output terminal for slave-1
assign output2=output22; //output terminal for slave-2
assign output3=output33; //output terminal for slave-3
assign output4=output44; //output terminal for slave-4

always @(posedge clk)
 begin
      case(rx_state)

            RX_IDLE:  begin
                           if(ready) begin
                           rx_state<=RX_ADDR;
                              counter<=6;
                                 output11<=0;
                                    output22<=0;
                                       output33<=0;
                                          output44<=0;end
                           else begin
                           rx_state<=RX_IDLE;
                              output11<=0;
                                 output22<=0;
                                    output33<=0;
```



```
                                       output44<=0; end
            end

            RX_ADDR: begin
                address1[counter]<=i2c_sda;
                output11<=0;output22<=0;output33<=0;output44<=0;

               if(counter==0) begin
                     rx_state<= RX_RD_WR_ENABLE;
                 end
               else counter <= counter-1;
             end

            RX_RD_WR_ENABLE:  begin
                     rx_state<= RX_WACK1;
                     output11<=0;output22<=0;output33<=0;output44<=0;
            end

            RX_WACK1:   begin
                  if((address==addr1)||(address==addr2)||(address==addr3)||(address==addr4))
                        begin wack11<=1;
                        rx_state<=RX_OUTPUT;
                        counter<=31;
                        end
                  else
                        begin wack11<=0;
                        rx_state<=RX_NON_OUTPUT;
                        counter<=31;
                        end
            end

           RX_OUTPUT:   begin
                             if(address==addr1)
                              begin
```



```
                                output11<= i2c_sda; wack22<=1;
                                     if(counter ==0) rx_state <=RX_WACK2;
                                 else counter<= counter-1;
                             end

                             else if(address==addr2)
                              begin
                                 output22<= i2c_sda; wack22<=1;
                                   if(counter ==0) rx_state <=RX_WACK2;
                                    else counter<= counter-1;
                             end

                             else if(address==addr3)
                              begin
                                 output33<= i2c_sda;wack22<=1;
                                   if(counter ==0) rx_state <=RX_WACK2;
                                     else counter<= counter-1;
                              end

                             else if(address==addr4)
                               begin
                                 output44<= i2c_sda;wack22<=1;
                                    if(counter ==0) rx_state <=RX_WACK2;
                                     else counter<= counter-1;
                              end

                               else
                                  begin
                                      output44<=0;
                                      output33<=0;
                                      output22<=0;
                                      output11<=0;
                                      wack11<=0;
                                      wack22<=0;
```



```
                                        if(counter ==0) rx_state <=RX_WACK2;
                                          else counter<= counter-1;
                                   end
            end

            RX_NON_OUTPUT: begin
                           output44<=0; output33<=0; output22<=0; output11<=0;

                           if(counter ==0) rx_state<=RX_WACK2;
                           else counter<= counter-1;
            end


            RX_WACK2:    begin
                        wack22<=1;
                        rx_state<=RX_STOP;
                        output11<=0;output22<=0;output33<=0;output44<=0;
            end

            RX_STOP:    begin
                        output11<=0;output22<=0;output33<=0;output44<=0;
                        wack11<=0;
                        wack22<=0;
                        rx_state<=RX_IDLE;
            end

   endcase //endcase
 end //end always

endmodule
```

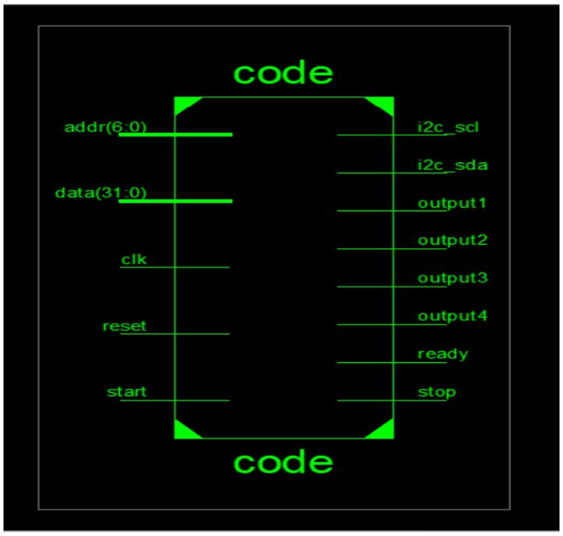
Code Explanation:

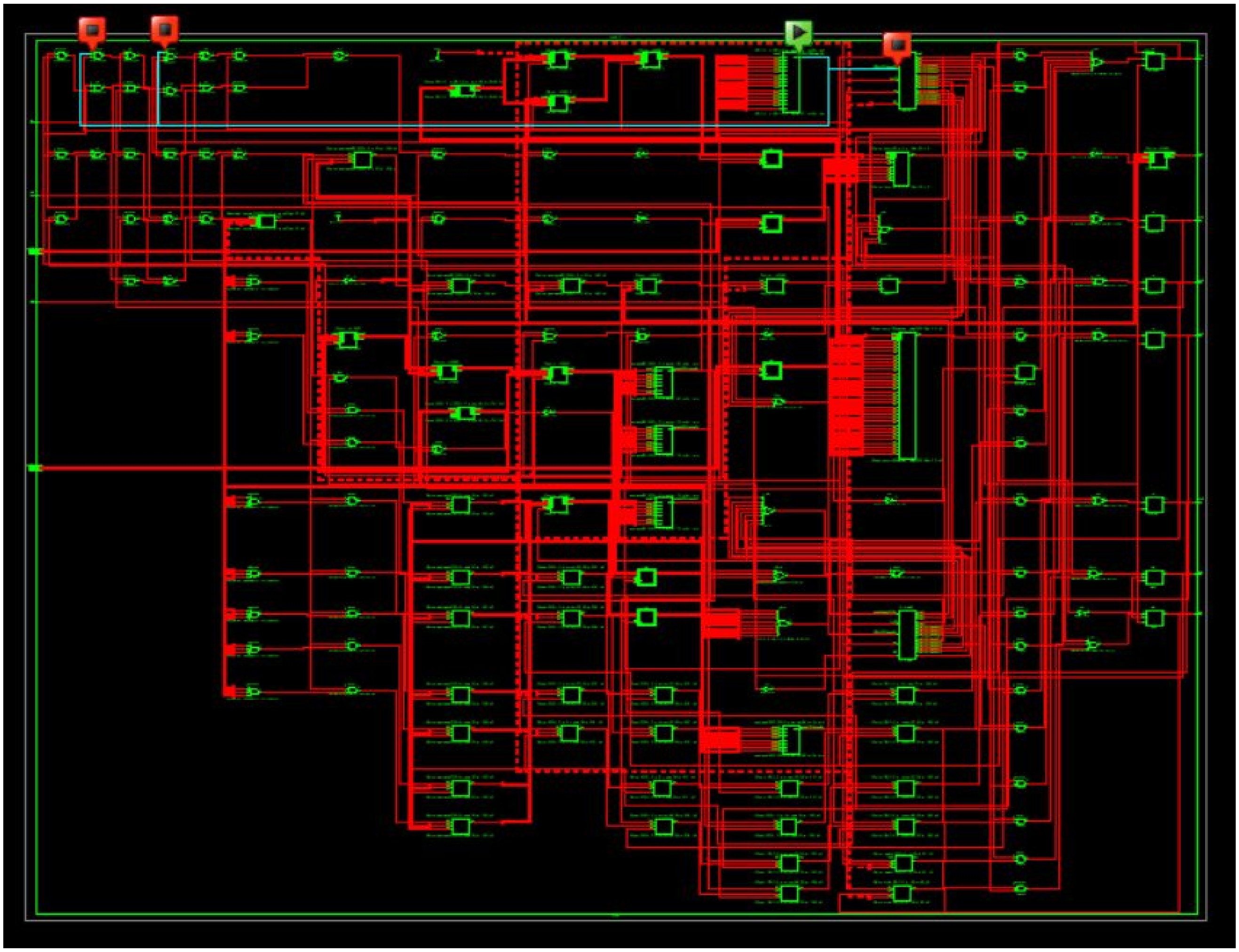
It has a master which guides slaves with two wires namely i2c_scl and i2c_sda which are clock and data_address lines which are kept at high(Vcc) unless data transmission start from master which pulls down the i2c_sda and then after i2c_scl will start tracing clock(external) and then make data transfer between master and slaves synchronous. Output1, 2, 3 & 4 are output terminals of slaves which trace the data given by master if it has been addressed by master.

When i2c_sda is pull down , then start bit of transmission starts and after a clock pulse next 7 bit address will be sent and then if address will be matched with any slave then acknowledgement bit will be received and shown on i2c_sda and read enable will be sent to slave and then after 32-bit data transfer will happen and again an acknowledgement bit will be sent to i2c_sda line to show data has been received and then stop bit will start and then it finishes . All this happens when only reset=0 and start =1 is given as input and all these have been coded using FSM.

Similarly four slave configurations have been set using FSM again to send ack and receive data if address matches otherwise ack will not be sent . However slave has only control over i2c_sda for acknowledgement only otherwise i2c_sda is always handled by master.

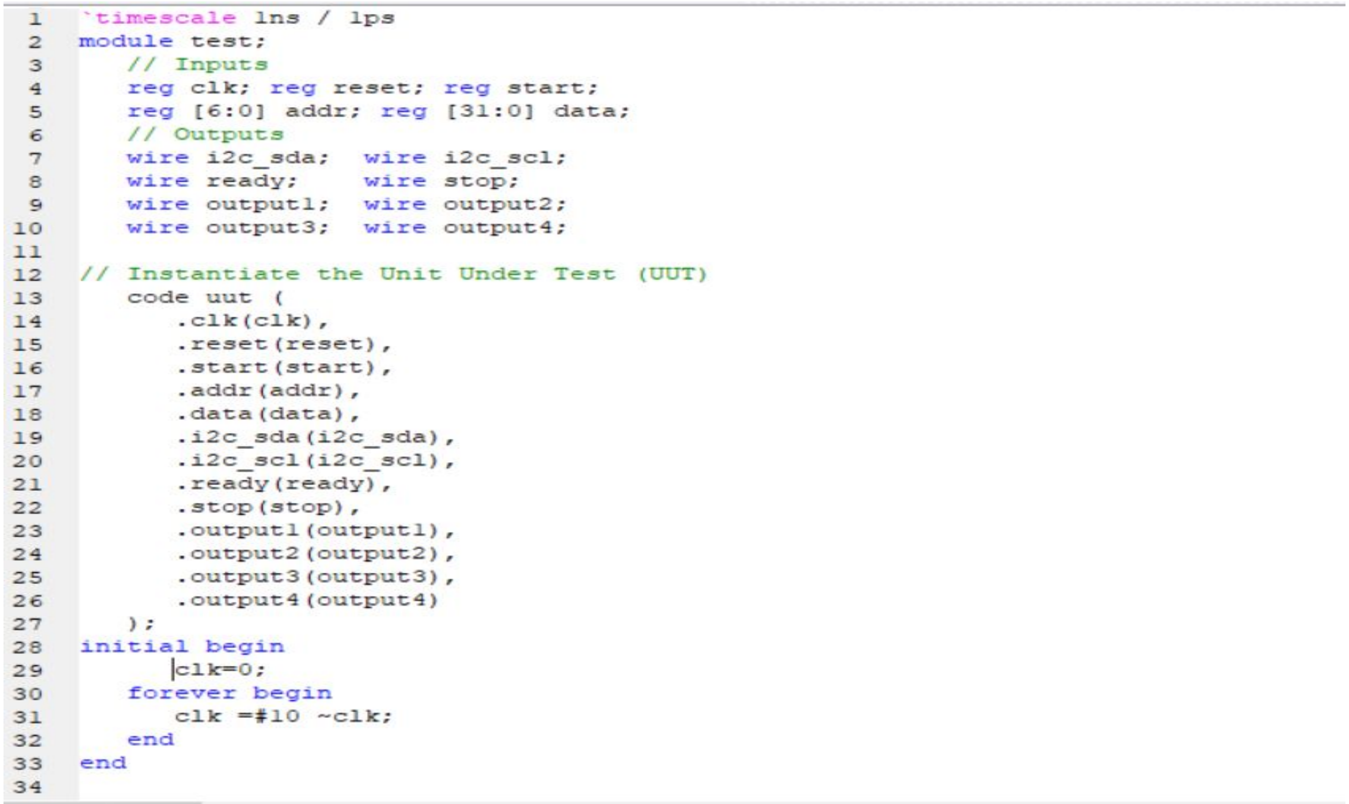
RTL:



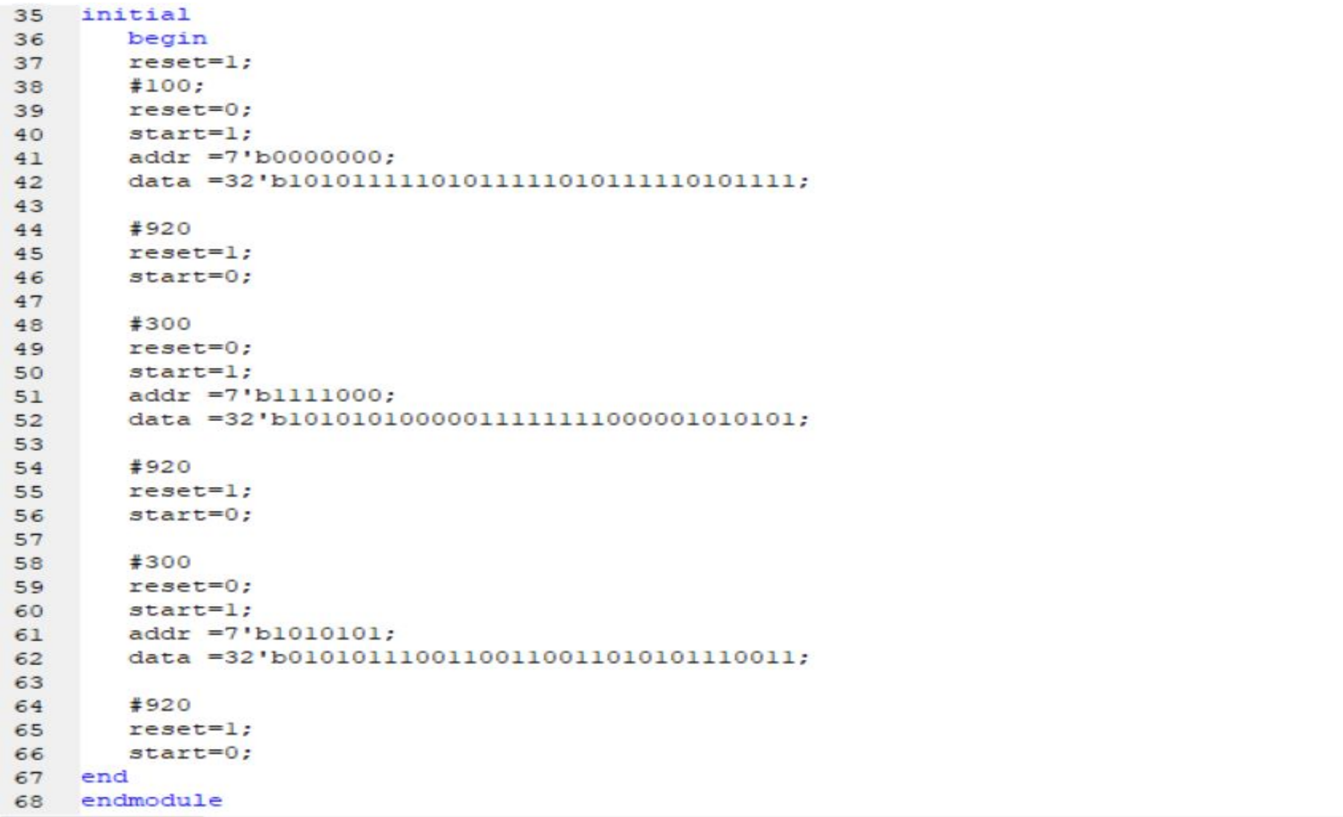


RTL view using Xilinx ISE design suite.

TEST BENCH:

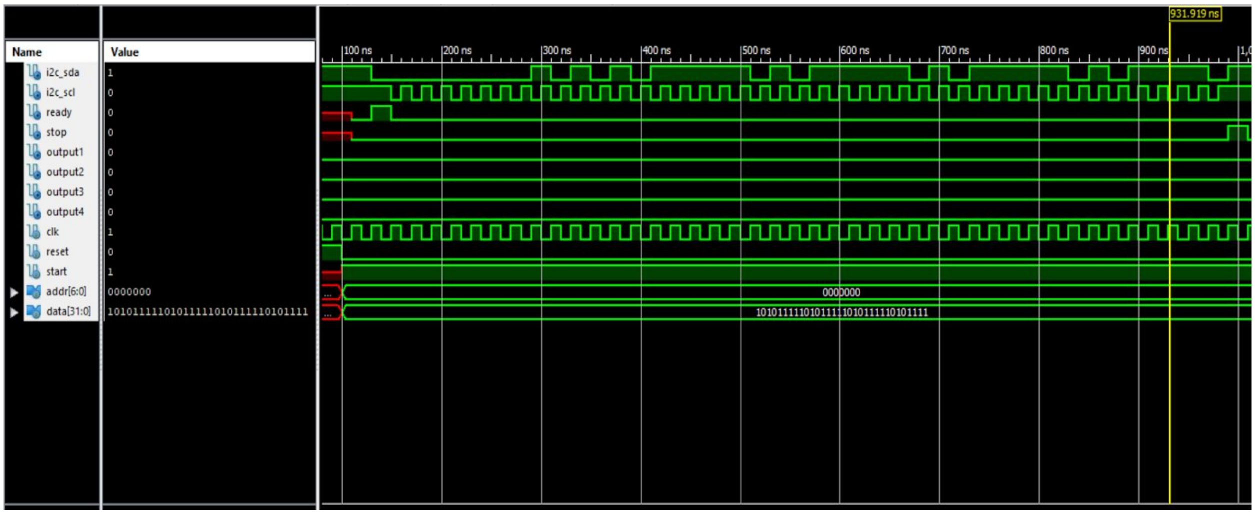


```
`timescale 1ns / 1ps
module test;
   // Inputs
   reg clk; reg reset; reg start;
   reg [6:0] addr; reg [31:0] data;
   // Outputs
   wire i2c_sda;  wire i2c_scl;
   wire ready;    wire stop;
   wire output1;  wire output2;
   wire output3;  wire output4;

// Instantiate the Unit Under Test (UUT)
   code uut (
      .clk(clk),
      .reset(reset),
      .start(start),
      .addr(addr),
      .data(data),
      .i2c_sda(i2c_sda),
      .i2c_scl(i2c_scl),
      .ready(ready),
      .stop(stop),
      .output1(output1),
      .output2(output2),
      .output3(output3),
      .output4(output4)
   );
initial begin
      clk=0;
   forever begin
      clk =#10 ~clk;
   end
end

```



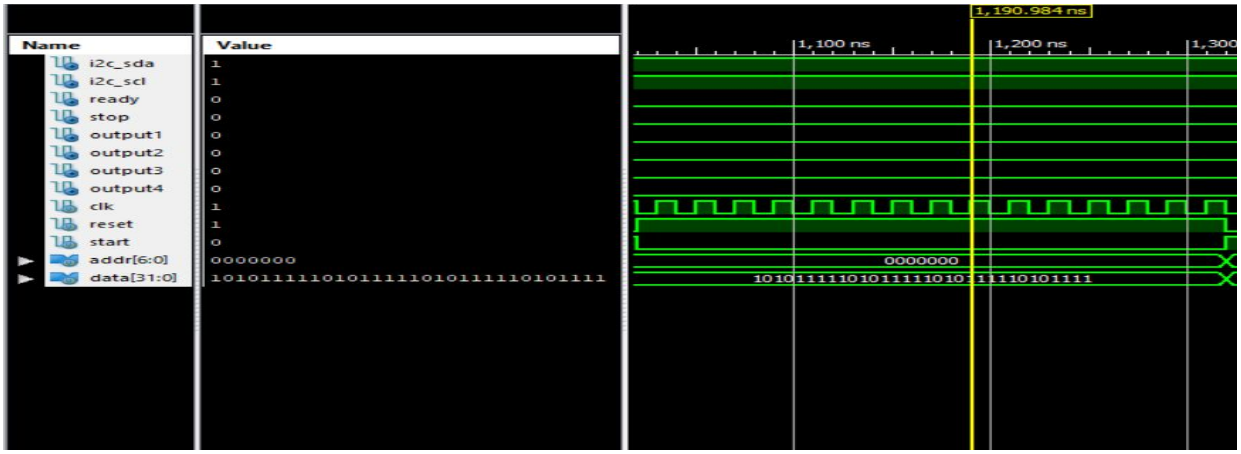
```
initial
   begin
   reset=1;
   #100;
   reset=0;
   start=1;
   addr =7'b0000000;
   data =32'b10101111101011111010111110101111;

   #920
   reset=1;
   start=0;

   #300
   reset=0;
   start=1;
   addr =7'b1111000;
   data =32'b101010100000011111111000001010101;

   #920
   reset=1;
   start=0;

   #300
   reset=0;
   start=1;
   addr =7'b1010101;
   data =32'b01010111001100110011010101110011;

   #920
   reset=1;
   start=0;
end
endmodule
```

OUTPUT DESCRIPTION FROM TEST BENCH

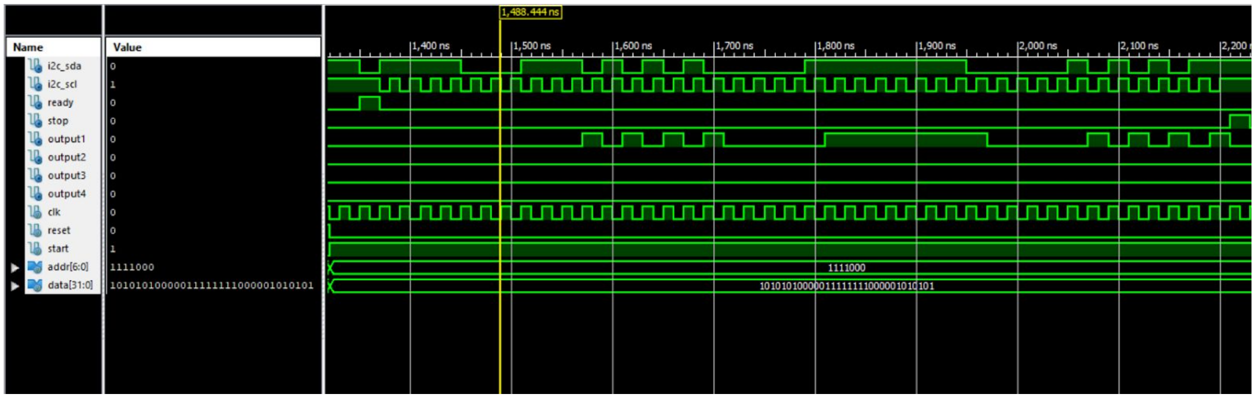


A). It is the case when reset is pulled down to 0 and start =1 and data and address is given from the test bench to start the operation . Here address =7'b0000000 which doesn't match with any defined slave address(refer code, line 163).Now ready is high to start operation of i2c. Hence address transfer is happening as it is controlled by master and then read enable is active high and sent by master shown is i2c_sda but as no address is matched ,hence next bit is ack from slave is not received and hence i2c_sda is 0 for a clock pulse and then 32-bit data is send as it is controlled by master and then again ack is 0 as no output terminal received it and then next stop bit is sent.

As clearly mentioned in theory, slaves control i2c_sda only for 2 pulse duration i.e. two acknowledgement one for address receiving and other for data receiving.Otherwise i2c_sda is always controlled by Master. Ready and stop signal gets high to 1 if it is ready and stop respectively with respect to protocol code.



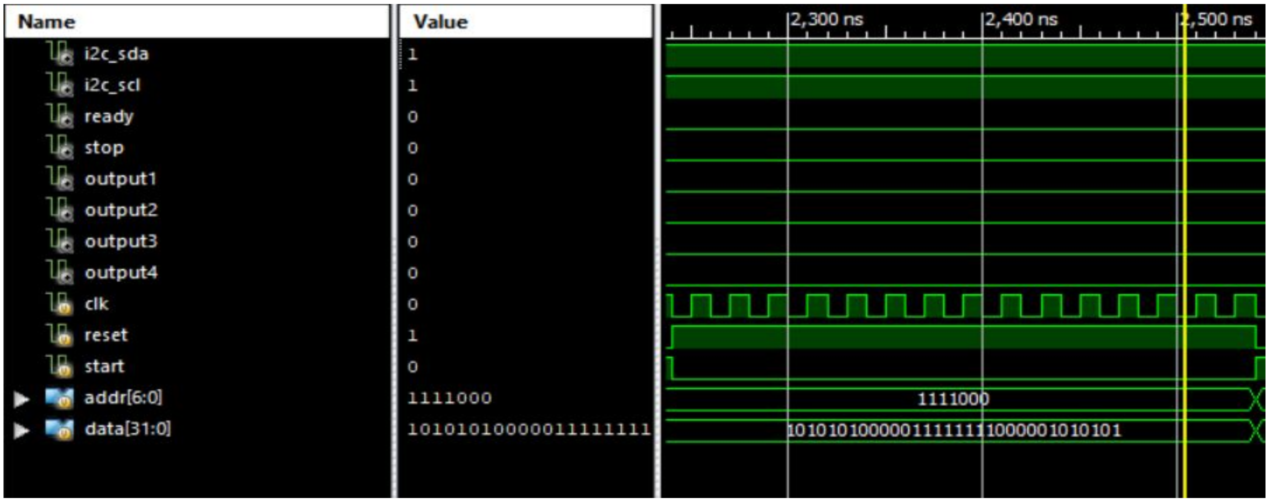
B). Now reset=1 and start=0; which does stop and makes the idle condition pull i2c_sda and i2c_scl as high(Vcc) as mentioned in protocol.



Name	Value
i2c_sda	0
i2c_scl	1
ready	0
stop	0
output1	0
output2	0
output3	0
output4	0
clk	0
reset	0
start	1
addr[6:0]	1111000
data[31:0]	10101010000011111111000001010101

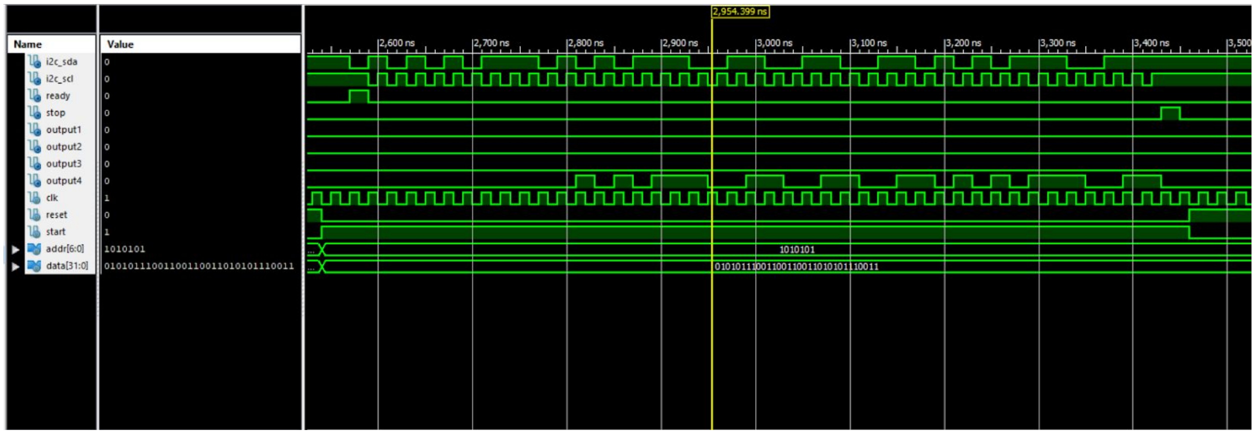
C). It is again now ,reset is pulled down to 0 and start =1 and data and address is given from the test bench to start the operation . Here address =7'b1111000 which does match with defined slave-1 address(refer code, line 163).Now ready is high to start operation of i2c. Hence address transfer is happening as it is controlled by master and then read enable is active high and sent by master shown is i2c_sda now as address is matched ,hence next bit is ack from slave is also received and hence i2c_sda is 1 for a clock pulse and then 32-bit data is send as it is controlled by master and then again ack is 1 as output terminal-1 received it and then next stop bit is sent.

And it is clearly seen that all 32-bit patterns are shown in Output1 serially as it is well known that it is serial protocol.One look away here is dat is shown after 1-clock pulse which is necessary to demonstrate as practically it is impossible for any device to produce output at the instant it received input and hence at after a pulse when master gives data, it shows on output -1 and thus 32- data transfer occurs serially and at last 32-bit ack is sent and then stop bit. Ready and stop signal gets high to 1 if it is ready and stops respectively with respect to protocol code.



Name	Value
i2c_sda	1
i2c_scl	1
ready	0
stop	0
output1	0
output2	0
output3	0
output4	0
clk	0
reset	1
start	0
addr[6:0]	1111000
data[31:0]	10101010000011111111

D). Now reset=1 and start=0; which does stop and makes the idle condition pull i2c_sda and i2c_scl as high(Vcc) as mentioned in protocol.



E). Now again ,reset is pulled down to 0 and start =1 and data and address is given from the test bench to start the operation . Here address =7'b1010101 which does match with defined slave-4 address(refer code, line 166).Now ready is high to start operation of i2c. Hence address transfer is happening as it is controlled by master and then read enable is active high and sent by master shown is i2c_sda now as address is matched ,hence next bit is ack from slave is also received and hence i2c_sda is 1 for a clock pulse and then 32-bit data is send as it is controlled by master and then again ack is 1 as output terminal-1 received it and then next stop bit is sent.

And it is clearly seen that all 32-bit patterns are shown in Output1 serially as it is well known that it is serial protocol.One look away here is data is shown after 1-clock pulse which is necessary to demonstrate as practically it is impossible for any device to produce output at the instant it received input and hence at after a pulse when master gives data, it shows on output -4 and thus 32- data transfer occurs serially and at last 32-bit ack is sent and then stop bit. Ready and stop signal gets high to 1 if it is ready and stops respectively with respect to protocol code.

Result:

Demonstrated the I2c protocol using Verilog as HDL and simulated it for various cases to verify it.

Conclusion:

1. Only two bus lines are required; a serial data line (SDA) and a serial clock line (SCL).

2. Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.

3. It is a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.

4. Serial, 32-bit oriented, bidirectional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode, up to 1 Mbit/s in Fast-mode Plus, or up to 3.4 Mbit/s in the High-speed mode.

5. I2C-bus compatible ICs not only assist designers, they also give a wide range of benefits to equipment manufacturers because:

a. The simple 2-wire serial I2C-bus minimizes interconnections so ICs have fewer pins and there are not so many PCB tracks; result — smaller and less expensive PCBs.
b. The completely integrated I2C-bus protocol eliminates the need for address decoders and other 'glue logic'.
c. The multi-master capability of the I2C-bus allows rapid testing and alignment of end-user equipment via external connections to an assembly line.
d. The availability of I2C-bus compatible ICs in various leadless packages reduces space requirements even more.

References:

1. https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

2. https://en.wikipedia.org/wiki/I%C2%B2C

3. https://www.electronicshub.org/basics-i2c-communication/

4. https://www.nxp.com/docs/en/user-guide/UM10204.pdf