Term Project Report on

I2C PROTOCOL: DESIGNING & VERIFICATION

B.Tech. EC
Sem. VIII

Submitted by

Janki Vaya	**Siddh Virani**	**Shivam Prasad**
EC095	**EC097**	**EC060**
18ECUOF004	**18ECUOG008**	**18ECUOS089**

Under the supervision of

Prof. Pinkesh Patel

Department of Electronics & Communication
Faculty of Technology
Dharmsinh Desai University
Nadiad

2021-22

Certificate

This is to certify that the project on I2C **PROTOCOL: DESIGNING AND VERIFICATION** and term work carried out in subject of Term Project is bonafide work of **Janki Vaya** (Roll no.: **EC095**) of **B. Tech**. Semester **VII** in the branch of **Electronics & Communication**, during the academic year **2021-22.**

Prof. Pinkesh Patel
Project Guide,
EC Department

Dr. Purvang Dalal
Head of Department
EC Department

Certificate

This is to certify that the project on **I2C PROTOCOL: DESIGNING AND VERIFICATION** and term work carried out in subject of Term Project is bonafide work of **Siddh Virani** (Roll no.: **EC097**) of **B. Tech**. Semester **VII** in the branch of **Electronics & Communication**, during the academic year **2021-22.**

Prof. Pinkesh Patel
Project Guide,
EC Department

Dr. Purvang Dalal
Head of Department
EC Department

Certificate

This is to certify that the project on **I2C PROTOCOL: DESIGNING AND VERIFICATION** and term work carried out in subject of Term Project is bonafide work of **Shivam Prasad** (Roll no.: **EC060**) of **B. Tech**. Semester **VII** in the branch of **Electronics & Communication**, during the academic year **2021-22.**

Prof. Pinkesh Patel
Project Guide,
EC Department

Dr. Purvang Dalal
Head of Department
EC Department

ACKNOWLEDGEMENT

Firstly, we offer our sincere gratitude towards our guide **Prof**. **Nisarg bhatt** & **Prof. Pinkesh Patel sir**, Assistant Professor, Electronics and Communication Department, Dharmsinh Desai Institute of Technology for their invaluable support, information, motivation and guidance given through this Project and for helping to establish our direction.

We express our deep sense of gratitude to our respected HOD and Professor, **Dr. Purvang Dalal**, EC Department, Dharmsinh Desai Institute of Technology, who has given us invaluable support and given opportunities to learn and develop technical skills. And has been a constant source of inspiration to us. We are thankful to all the faculty members and institution staff for their consistent support and guidance.

TABLE OF CONTENT

LIST OF FIGURES

ABSTRACT

The project we have undertaken is **"I2C Protocol: Designing & Verification".** I2C is abbreviation Inter-Integrated Circuit. It is most popular serial communication protocol developed by Phillips in late 1980s.It is used mainly in communication among modules and sensors. The idea of this project is to design and verify the RTL design with help of System Verilog Language. Verification of design is essential part of VLSI design hierarchy. Fabrication of process is costly and time consuming so verification is required. To full fill the requirement system Verilog is used for verification of any complex design. System Verilog is very flexible and objective language which is consist of Object Oriented (OOP) Concepts, Randomization, Assertion and Coverage methods. This projects shows how test bench can be written to verify I2C design with OOP concepts and modules of classes. To perform these actions, a powerful tool is essential, this projects is written on "EDA Play Ground" platform. It is open source, cloud based, consist of multiple commercial simulators, wave form compatibility and many more.

CHAPTER - 1 INTRODUCTION

In the Electronic world, communication protocols are the backbone links for an embedded system, thus they are very important for us Seekers to understand what they are, why they are used, and the difference between communication protocols.

Wired communication protocols are simply a set of rules that allow two or more entities of a communication system to transmit information via physical medium. Therefore, the syntax, semantics, and synchronization of communication and possible error recovery methods between communication systems are all defined by the term “protocol”. Protocols can be implemented by both hardware and software or a combination of both. Further, each protocol has its own application area.

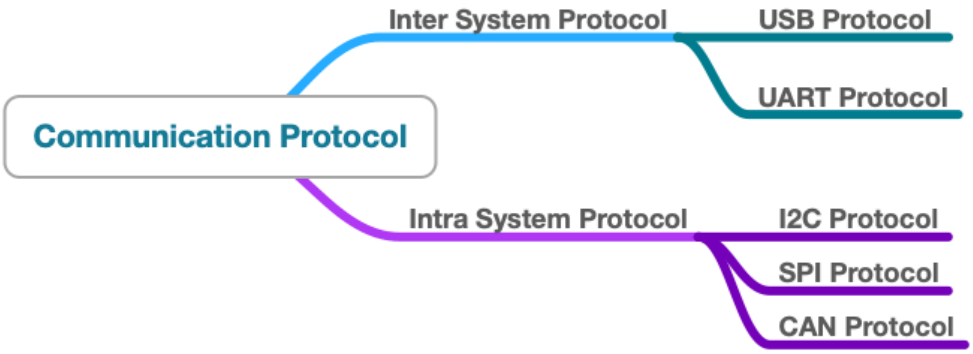

Figure 1.1 Various Communication Protocols

In figure 1.1, General diagram of Communication Protocol is shown. Communication protocols are divided in two parts, Inter System Protocol and Intra System Protocol. Inter System protocols are the protocol which helps in flexible communication among other devices\systems. Unlikely, Intra system protocols are used within systems modules. We will discuss about only I2C Protocol From onwards.

In following chapters and subsection we will show the all the tools and languages required to design and verify via writing test bench.

1.1 Inter Integrated Circuit (I2C) Communication Protocol

(I2C) Inter Integrated Circuit, pronounces “eye-two-see” or “eye-squared-see”, combines the best features of SPI and UART. With I2C, you are able to connect multiple slaves to a single master (just like SPI) or having multiple masters controlling single or even multiple slaves. This is extremely useful when you are logging data into a single LCD from more than one micro-controller.

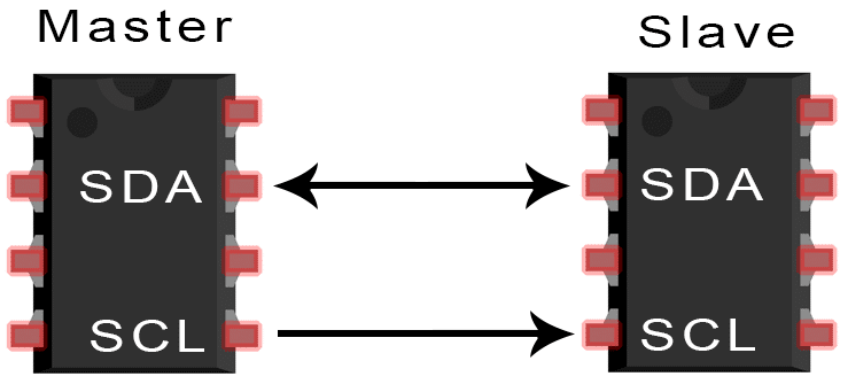

Figure 1.2 - I2C Configuration

It requires only two wires connecting all peripherals to the microcontroller. I2C requires two wires SDA (serial data line) and SCL (serial clock line) to carry information between devices. It is a master to a slave communication protocol. Each slave has a unique address. The master device sends the address of the target slave device and reads/writes the flag. The address matches any slave device that the device is ON, the remaining slave devices are disabled mode.

Once the address is match communication proceed between the master and that slave device and transmitting and receiving the data. The transmitter sends 8-bit data, the receiver replies 1-bit of acknowledgment. When the communication is completed master issues the stop condition. The I2C bus was developed by Philips Semiconductors (NXP). Its original purpose is to provide an easy way to connect CPU to peripherals chips.

Peripheral devices in embedded systems are often connected to the microcontroller as memory-mapped devices. I2C requires only two wires for connecting all the peripherals to the microcontroller. These active wires, called SDA and SCL, are both bidirectional. SDA line is a serial data line and the SCA line is a serial clock line.

1.2 Introduction to Verilog HDL

The Verilog Hardware Description Language (Verilog HDL) is a language that describes the behavior of electronic circuits, most commonly digital circuits. Verilog HDL is defined by IEEE standards. There are three common variants: Verilog 1995, Verilog 2001, and the recent SystemVerilog 2005.

A Verilog design consists of a hierarchy of modules. Modules encapsulate **design hierarchy**, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. However, the blocks themselves are executed concurrently, making Verilog a dataflow language.

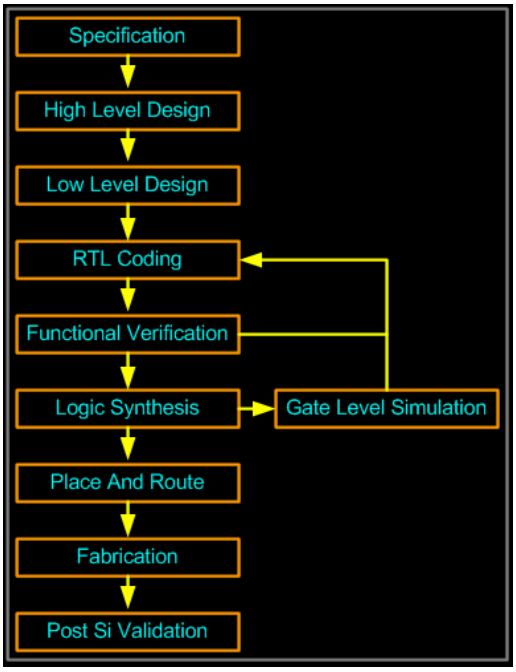

Figure 1.3 Design Hierarchy

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating or high impedance, undefined") and signal strengths (strong, weak, etc.). This system allows abstract modeling of shared signal lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

1.3 Verilog Abstraction Levels

Verilog supports designing at many different levels of abstraction. Three of them are very important:

- Behavioral Level
- Register-Transfer Level
- Gate Level

(a) Behavioral level:

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

(b) Register – Transfer Level:

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

(c) Gate Level:

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (‘0’, ‘1’, ‘X’, ‘Z’). The usable operations are predefined logic primitives (AND, OR, NOT etc. gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.

We will discuss about more in upcoming chapters.

1.4 Need of Verification

A primary purpose for functional verification is to detect failures so that bugs can be identified and corrected before it gets shipped to costumer. If RTL designer makes a mistake in designing or coding, this results as a bug in the Chip. If this bug is executed, in certain situations the system will produce wrong results, causing a failure. Not all mistakes will necessarily result in failures. The bug in the dead code will never result in failure. A single mistake may result in a wide range of failure symptoms. Not all bugs are caused by coding errors. There are possibilities that error may in the specification itself.

1.5 Introduction to System Verilog

System Verilog has been called the industry's first Hardware Description and Verification Language (HDVL), because it combines the features of Hardware Description Languages such as Verilog and VHDL with features from specialized Hardware Verification Languages, together with features from C and C++. System Verilog first became an official IEEE standard (IEEE 1800™) in 2005, was updated with IEEE 1800™ 2009, and is now in the process of being further refined under the guidance of Accellera as tool vendors and users gain experience with the practical implementation and application of the language.

System Verilog is an extension of Verilog with many such verification features that allow engineers to verify the design using complex test bench structures and random stimuli in simulation.

Nowadays designs are very much complex so we need robust test bench which can verify the every possibility of input and can test every environment.

1.6 Testbench

Test Bench mimic the environment in which the design will reside. It checks whether the RTL Implementation meets the design spec or not. This Environment creates invalid and unexpected as well as valid and expected conditions to test the design.

Test Bench is the simplest, fastest and easiest way of writing test benches. This became novice verification engineer choice. It is also slowest way to execute stimulus. Typically, linear test benches are written in the VHDL or Verilog. In this Test Bench, simple linear sequence of test vectors is mentioned. Stimulus code like this is easy to generate translating a vector file with a Perl script, for example. Small models like simple state machine or adder can be verified with this approach. The following code snippet shows linear Test Bench. The code snippet shows some input combination only. This is also bad for simulator performance as the simulator must evaluate and schedule a very large number of events. This reduces simulation performance in proportion to the size of the stimulus process.

How does a Verification engineer check whether the results obtained from the simulation match the original specification of the design? For simple test benches like the above, output is displayed in waveform window or messages are sent to terminal for visual checking. Visually checking is the oldest and most labor intensive technique. The quality of the verification depends on the determination and dedication of the individual who is doing the checking. It is not practical to verify a complex model merely by examining the waveform or text file. Whenever a change is made to the DUT to add a new feature or to fix a bug, same amount of effort needs to be deployed to check the simulation results.

A self-checking Test Bench checks expected results against actual results obtained from the simulation. Although Self-checking test benches require considerably more effort during the

initial test bench creation phase, this technique can dramatically reduce the amount of effort needed to re-check a design after a change has been made to the DUT. Debugging time is significantly shortened by useful error-tracking information that can be built into the Test Bench to show where a design fails.

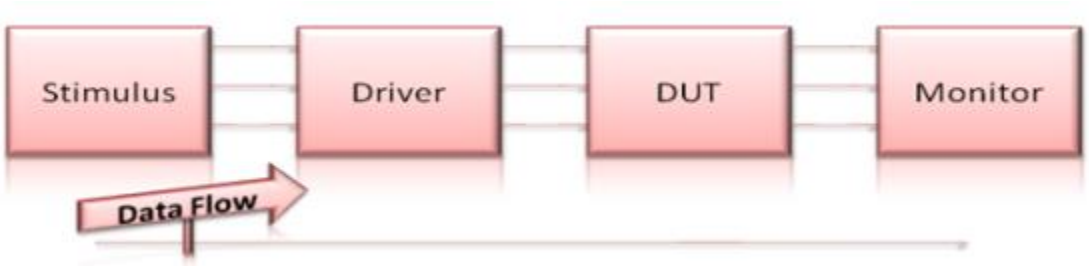

Figure 1.4 Self Checking Test Benches

A self-checking TestBench has two major parts, the input blocks and output blocks. Input block consist of stimulus and driver to drive the stimulus to DUT. The output block consists of monitor to collect the DUT outputs and verify them.

All the above approaches require the test writer to create an explicit test for each feature of the design. Verification approach in which each feature is written in a separate test case file is called directed verification.

1.7 Motivation

The aim of the project is to do verification of Designed I2C protocol. Since, we have learned Verilog HDL in earlier semester, we have sufficient experience with it for designing of digital circuits. So, it will help us to design I2C protocol. Similar way as per shown in figure 1.3, design hierarchy, Function Verification is required before manufacturing or fabrication of chip. As per discussed in previous section, for verification purpose System Verilog is helpful. We will be discussing about this new hardware verification language (HVL) in upcoming chapters.

CHAPTER - 2 DESIGNING OF I2C USING VERILOG HDL

In this chapter, we will see how I2C protocol can be design using Verilog and its features. First of all we will see basic understanding of I2C protocol.

2.1 I2C Protocol

I2C or I square C or Inter Integrated Circuit is half duplex and bidirectional data transfer protocol. It is Serial communication consisting of two dedicated lines, Serial Clock (SCL) & Serial Data (SDA). In figure 2.1, one example is shown. It is having two lines with several devices connected within it. The microcontroller can be act as a master and while others (LCD, A/D, D/A, EEPROM) are acting as slave.

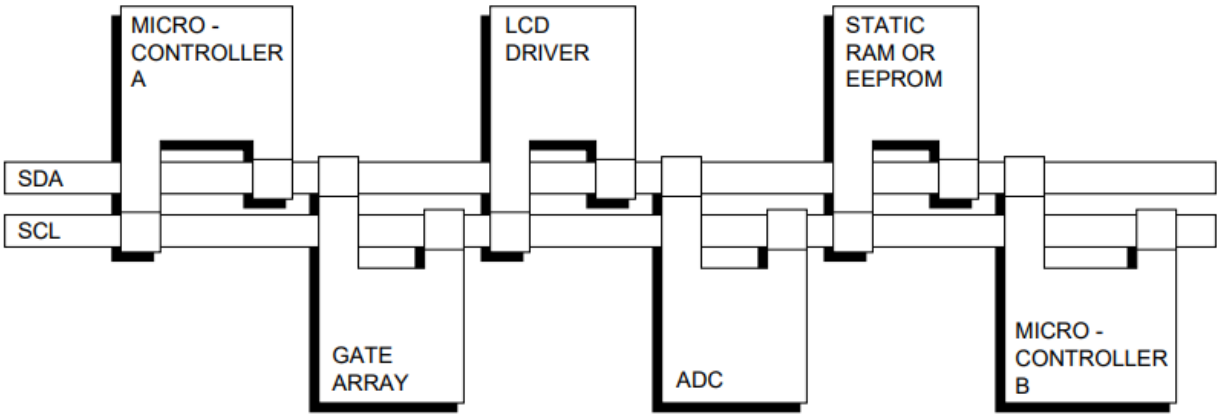

Figure 2.1 Master and Slave Configuration

This example highlights the master-slave and receiver-transmitter relationships found on the I2C-bus. Note that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows:

1. Suppose microcontroller A wants to send information to microcontroller B:
   a. microcontroller A (master), addresses microcontroller B (slave)
   b. microcontroller A (master-transmitter), sends data to microcontroller B (slave-receiver)
   c. microcontroller A terminates the transfer.

2. If microcontroller A wants to receive information from microcontroller B:
   a. microcontroller A (master) addresses microcontroller B (slave)
   b. microcontroller A (master-receiver) receives data from microcontroller B (slave-transmitter)

c. microcontroller A terminates the transfer

Even in this case, the master (microcontroller A) generates the timing and terminates the transfer. The possibility of connecting more than one microcontroller to the I2C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid the chaos that might ensue from such an event, an arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I2C interfaces to the I2C-bus.

Table 2.1 Applicability of I2C – Bus protocol features

Features	Configuration		
	Single master	Multi-master	Slave
START Condition	M	M	M
STOP Condition	M	M	M
Acknowledge	M	M	M
Arbitration	-	M	-
7-bit Slave Address	M	M	M

2.2 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 (see Figure 2.2). One clock pulse is generated for each data bit transferred.

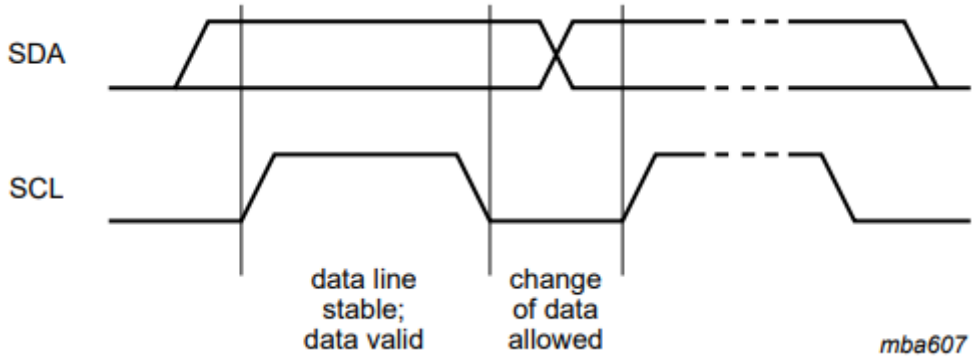

Figure 2.2 Data Validity

2.3 START and STOP Condition

All transactions begin with a START (S) and are terminated by a STOP (P) (see Figure 2.3). A HIGH to LOW transition on the SDA line while SCL is HIGH defines a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition. START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition.

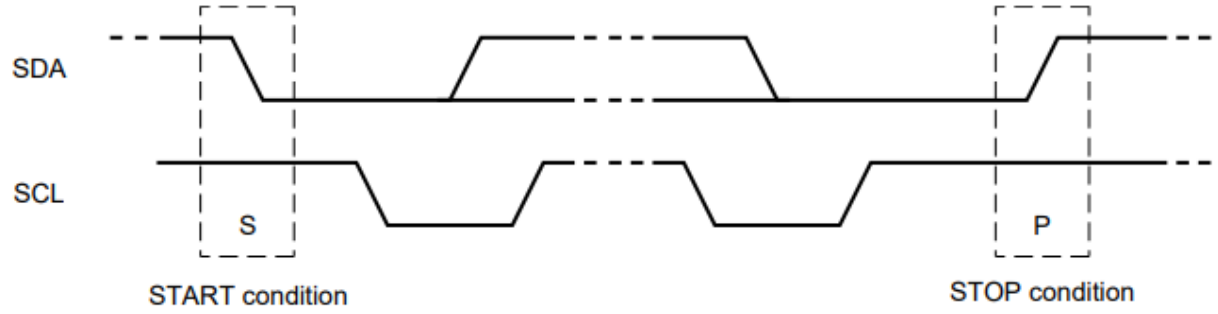

Figure 2.3 START and STOP Condition

The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical. For the remainder of this document, therefore, the S symbol is used as a generic term to represent both the START and repeated START conditions, unless Sr is particularly relevant

2.4 Byte Format

Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must be followed by an Acknowledge bit. Data is transferred with the Most Significant Bit (MSB) first (see Figure 2.4). If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

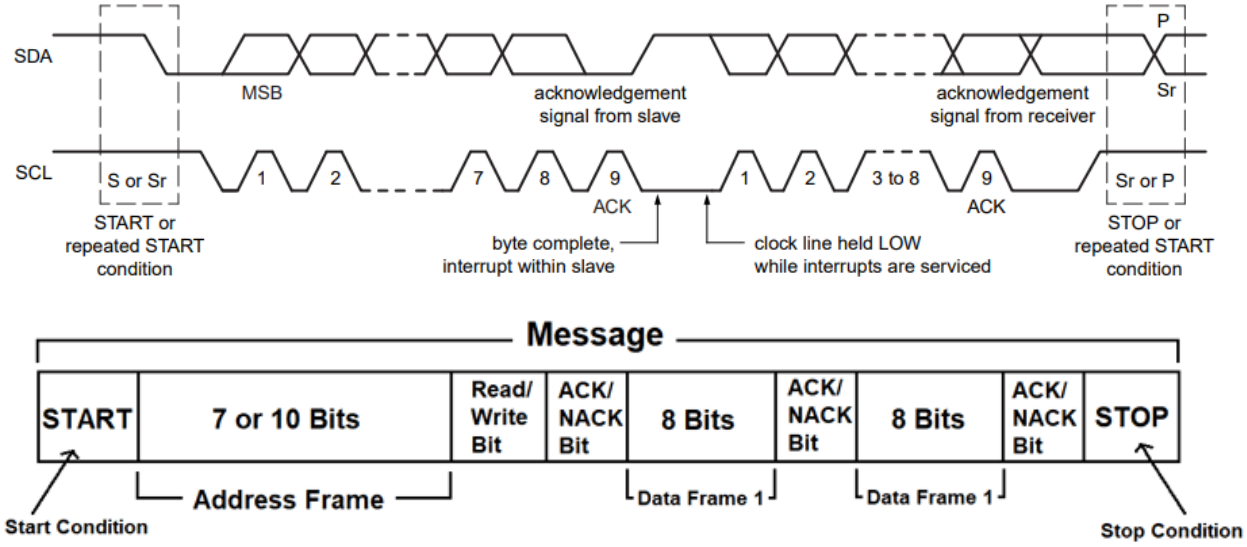

Figure 2.4 I2C Address and Data Format

2.5 Acknowledge (ACK) and Not Acknowledge (NACK)

The acknowledge takes place after every byte. The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. The master generates all clock pulses, including the acknowledge ninth clock pulse.

The Acknowledge signal is defined as follows: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse (see Figure 2.4). Set-up and hold times must also be taken into account.

When SDA remains HIGH during this ninth clock pulse, this is defined as the Not Acknowledge signal. The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer. There are five conditions that lead to the generation of a NACK.

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.

2.6 Arbitration

Arbitration, like synchronization, refers to a portion of the protocol required only if more than one master is used in the system. Slaves are not involved in the arbitration procedure. A master may start a transfer only if the bus is free. Two masters may generate a START condition within the minimum hold time of the START condition which results in a valid START condition on the bus. Arbitration is then required to determine which master will complete its transmission. Arbitration proceeds bit by bit. During every bit, while SCL is HIGH, each master checks to see if the SDA level matches what it has sent. This process may take many bits. Two masters can actually complete an entire transaction without error, as long as the transmissions are identical. The first time a master tries to send a HIGH, but detects that the SDA level is LOW, the master knows that it has lost the arbitration and turns off its SDA output driver. The other master goes on to complete its transaction.

No information is lost during the arbitration process. A master that loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration and must restart its transaction when the bus is free.

If a master also incorporates a slave function and it loses arbitration during the addressing stage, it is possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave mode.

Figure 8 shows the arbitration procedure for two masters. More may be involved depending on how many masters are connected to the bus. The moment there is a difference between the internal

data level of the master generating DATA1 and the actual level on the SDA line, the DATA1 output is switched off. This does not affect the data transfer initiated by the winning master.

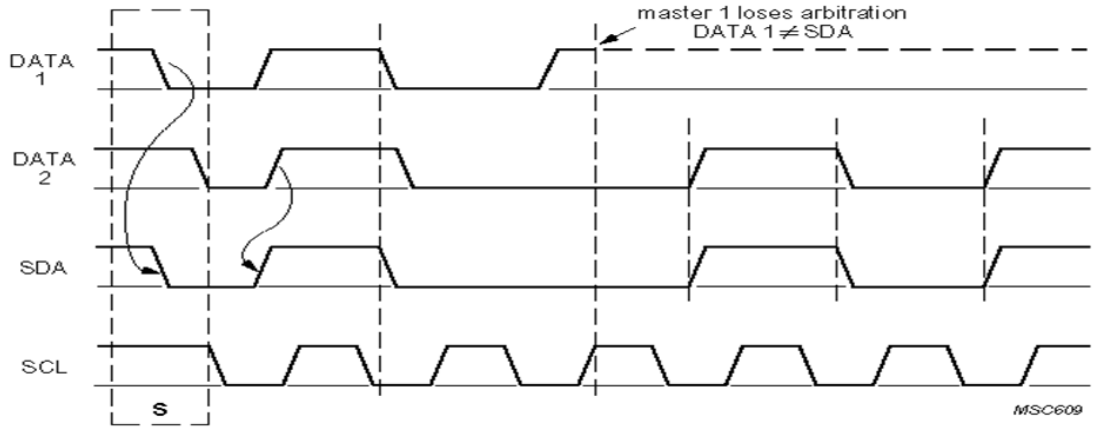

Figure 2.5 Arbitration

2.7 The slave address and R/W bit

Data transfers follow the format shown in Figure 2.6. After the START condition (S), a slave address is sent. This address is seven bits long followed by an eighth bit which is a data direction bit (R/W) — a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ). A data transfer is always terminated by a STOP condition (P) generated by the master. However, if a master still wishes to communicate on the bus, it can generate a repeated START condition (Sr) and address another slave without first generating a STOP condition. Various combinations of read/write formats are then possible within such a transfer.

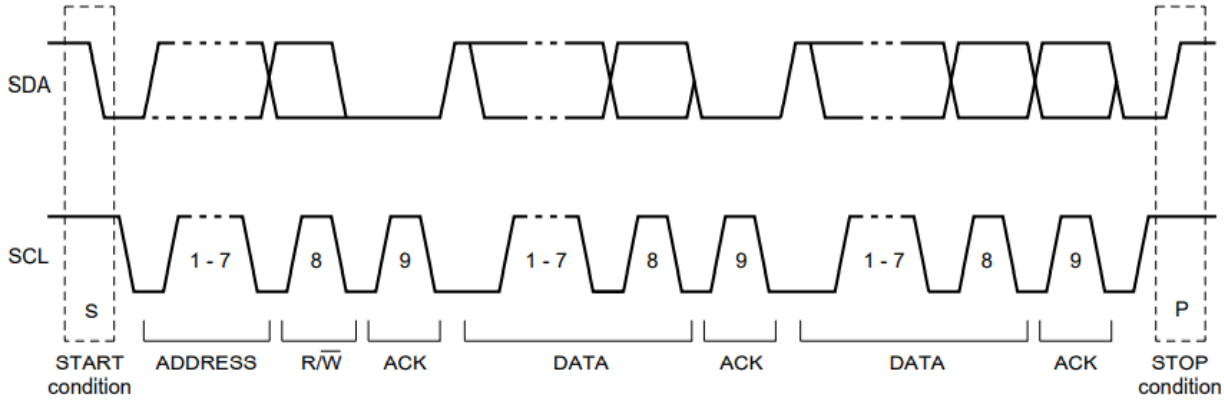

Figure 2.6 Data Transfer

Different Master-Slave Configuration has been shown in consecutive figure 2.7, 2.8 and 2.9.

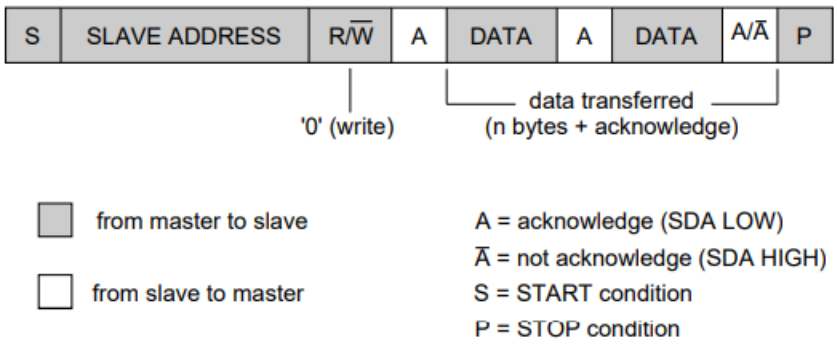

Figure 2.7 A master-transmitter addressing a slave receiver with a 7-bit address

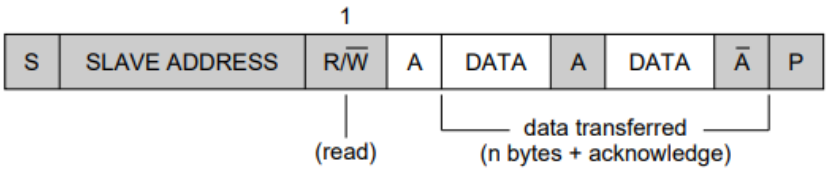

Figure 2.8 A master reads a slave immediately after the first byte

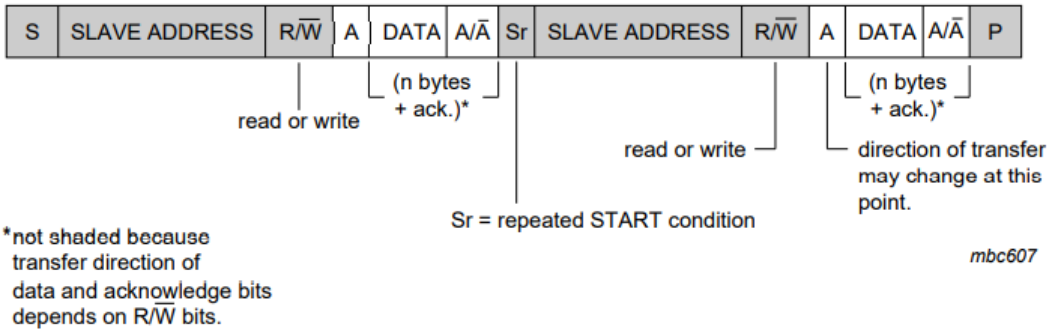

Figure 2.9 Combined format

2.8 Designing using Finite State Machine

In previous sections, we understood how I2C protocol works.it can be seen I2C works in several stages like, Start bit, address bits, acknowledge bits etc. In order to design protocol using Verilog HDL we can use finite state machines to elaborate whole design. The finite state machines (FSMs) are significant for understanding the decision making logic as well as control the digital systems. In the FSM, the outputs, as well as the next state, are a present state and the input function. This means that the selection of the next state mainly depends on the input value and strength lead to more compound system performance. Here in report we are assuming that readers are comfortable with Verilog HDL language.

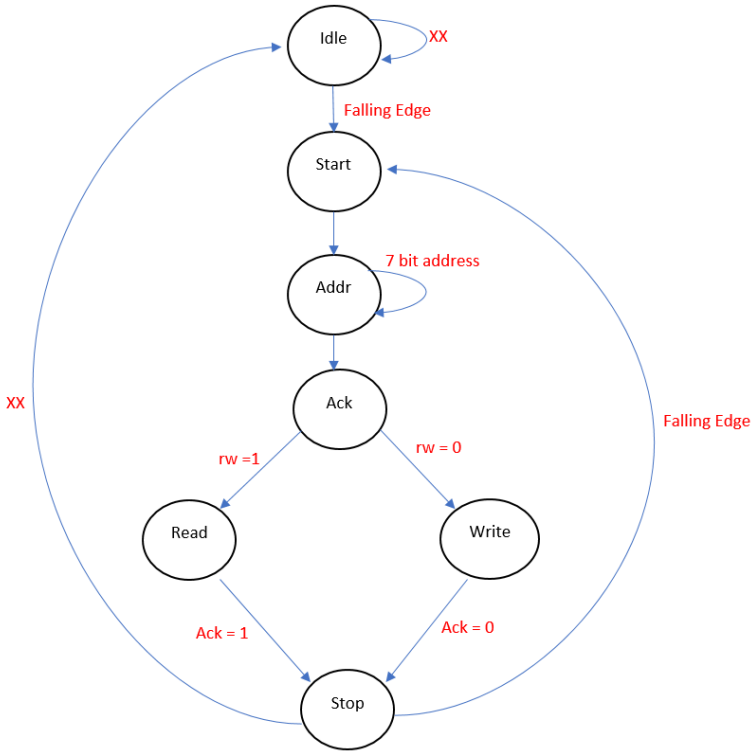

Figure 2.10 Finite State Machine for I2C Protocol

Where,

- **Idle** : No data on SDA line
- **Start** : Triggering of Address/Data Transfer
- **Addr** : Address fetching of slave
- **Ack** : Acknowledged the address
- **Read** : Master reads the data from SDA line
- **Write**: Master sends the data from sda line
- **Stop** : Stopping the data transfer

This FSM (shown in figure 2.10) can be understood by starting with idle state. Whenever state is in idle and it detects falling edge, it will enter in next state which is Start state. In Sequence, it will go to Addr state for 7 bit address detections, acknowledgment from slave (Ack state), Read or write state and repeat. In section 2.9, we will see the Verilog HDL code for implementation.

2.9 Verilog HDL Code

```
module AND(out,sda_1,sda_2);
  input sda_1,sda_2;
  output  out;
  assign out=sda_1&sda_2;
endmodule

module Slave(SDA,SCL,DATA_out,DATA_read,sample_sda,sample_sda1);
//inout triand SDA;
input triand SCL;
inout triand SDA;
output reg [7:0]DATA_out;
output reg [7:0] DATA_read;
output reg sample_sda;
output reg sample_sda1;

  assign sample_sda=SDA;
  assign sample_sda1=SDA;
  reg [3:0] IDLE                = 4'b0000;   // idle state 0
 // reg [3:0] START              = 4'b0001;   // start state 1
  reg [3:0] READ_ADDRESS     = 4'b0010;       // read address state 2
  reg [3:0] READ_WRITE         = 4'b0011;       // read_write state 3
  reg [3:0] READ               = 4'b0100;       // read data from slave state 4
  reg [3:0] DATA               = 4'b0101;       // write data to slave state 5
  reg [3:0] DATA_ACK    = 4'b0110;       // send ack to master state 6
  reg [3:0] READ_ACK    = 4'b0111;       // send ack to slave state 7
  reg [3:0] STOP               = 4'b1110;       // stop condition
  reg [3:0] ADDRESS_ACK        = 4'b1000;      // adress ack to master 8
```

```
  reg [3:0] state                 = 4'b0010;       // initial in read adress mode

  reg [6:0] slaveAddress    = 7'b1010_111;    // slave address is set as 0x28
  reg [6:0] addr                  = 7'b000_0000;    // adress register to store address
  reg [6:0] addressCounter        = 7'b000_0000;    //address counter

  reg [7:0] read_reg;      // = 8'b01010101;   // to send the data to master (during read operation)
  reg [7:0] data                  = 8'b0000_0000;  // so store the data that is sent by the master
  reg [6:0] dataCounter     = 7'b000_0000;   // data counter
  reg [6:0] readCounter     = 7'b000_0000;   // read counter

  reg readWrite                   = 1'b0;       // to store read write bit
  reg start                 = 0;          // start condition flag is set means that start condition is
occtred
  reg write_ack                   = 0;          // write_ack flag to access SDA line at time of
ACK
  reg sda_reg          = 0;         // data register while putting any data on the sda ny slave

  assign SDA = (write_ack == 1) ? sda_reg : 1'b1;   // putting data on SDA line

  always @(negedge SDA) begin    // to detect start condition
    if ((start == 0) && (SCL == 1) && (write_ack == 0))
    begin
              start <= 1;
       state <= READ_ADDRESS;// start flag is set
       addressCounter <= 0;  // reseting counters
              dataCounter <= 0;
       readCounter <= 0;
```

```
    end
end

always @(posedge SDA) begin   // to detect the stop condition
  if (SCL == 1 && (write_ack == 0))
    begin
      start <= 0;        // start flag is reset
              state <= READ_ADDRESS;  // initially READ_ADDRESS mode
      end
    end

always @(posedge SCL)     // main FSM
  begin
    if (start == 1)   // flag bit is set then operation will be perfomed
    begin
      case (state)
        READ_ADDRESS:
          begin
            write_ack = 0;
            addr[6-addressCounter] = SDA; //  address sampling

            if (addressCounter == 6)
              begin
                //$display("address %h =", addr);
                if(addr==slaveAddress )   // comparing the address
                  begin
                                state = READ_WRITE;  // go in to read_write cheking state
                  end
```

```
            else
              begin
                    state = IDLE;    // if address don't match then go in idle state
              end
          end
            addressCounter = addressCounter + 1;
      end


READ_WRITE:     // sampling of read_write  bit
  begin
       readWrite <= SDA;
        state <= ADDRESS_ACK;
   end
   ADDRESS_ACK:    // taking decision for  read or write state
     begin
       sda_reg <= 0;     // giving ack to master
       write_ack <= 1;   // taking the access of the sda line
       if(readWrite==0)
       begin
          state <= DATA;
          @(negedge SCL);
          write_ack <= 0; //giving acess back to master
       end
       else begin
          state <= READ;
          read_reg<=$urandom();
       end
```

```
        end
      DATA:
        begin
          write_ack <= 0;    // giving the access back to master
        dataCounter <= dataCounter + 1;
          if (dataCounter == 7)  // going in to ACK state
            begin
                state <= DATA_ACK;
            end
          else begin
                state <= DATA;
          end
             data[7-(dataCounter)] <= SDA;  // data storing in the data reg
    end
      READ:       // read state to read data from slave
          begin
            if(readCounter<=7)
              begin
                write_ack <= 1;   // tking access of SDA line
                sda_reg <= read_reg[7-readCounter];   // sending data to master bit by bit (first
msb)
                readCounter <= readCounter + 1;   // read
              end
            if (readCounter == 7)
            begin
               state <= READ_ACK;   // going to read ACK state
            end
          end
```

```
        DATA_ACK:
       begin
          state <= DATA;
          DATA_out<=data;
          dataCounter <= 0; //  reset the counter
          write_ack <= 1;  // taking access back from master
          sda_reg <= 0;  // giving the ACK
          @(negedge SCL); // taking access back at negative edge
          write_ack <= 0; //giving acess back to master
       end
       READ_ACK:
       begin
          write_ack <= 0;
          readCounter <= 0;
          if(SDA==0)   // cheking the ack of data read
          begin
                    DATA_read<=read_reg;
             state<=READ_ADDRESS;
          end
          @(negedge SCL);
          write_ack <= 1; //
       end
     endcase
   end
  end
endmodule
```

The code shown above, it is same replica of FSM shown in figure 2.10. We can see that all the necessary states are defined as **register (reg)** like, SCL, SDA, IDLE, READ, DATA, READ_ACK DATA_ACK etc.

To define FSM, we used features of Verilog HDL, **'SWITCH CASE'**. Switch case allows to switch the states of I2C operation. It gives flexibility to code to run particular state case in parallel with other tasks. Also, Non-blocking (<=) operators have been used to do parallel operation.

CHAPTER - 3 TESTBENCH USING SYSTEM VERILOG

3.1 Introduction to Verification

Verification is the process in which design is tested (or verified) against a given design specification before tape-out. This happens along with the development of the design and can start from the time the design architecture/microarchitecture definition happens.

The main goal of verification is to ensure the functional correctness of the design before the tape out and it is the most important aspect of the product development process as it is consuming 80% of the total product development time.

Functional Verification is defined as the process of verifying that an RTL (Synthesizable Verilog, VHDL, and System Verilog) design meets its specification from a functional perspective.

RTL Verification is usually divided into two discrete areas:

- Functional verification: It establishes that the design under test (DUT) implements the functionality of the specification correctly.

- Physical verification: It checks that the synthesis, implementation and back-end flow maintain the same functionality in their level of abstraction.

Effective verification requires **two critical elements**, both of which are promoted by SystemVerilog(which also supports OOP which makes verification of designs at a higher level of abstraction possible).

1) The verification environment must be set up to detect bugs as automatically as possible, which is where features like assertions, automated response checkers, and scoreboards come in handy.
2) it must be able to generate the proper stimulus to cause bugs to happen. Stimuli may be generated via either directed tests or constrained random techniques that exercise a wide range of scenarios with a relatively small amount of code.

3.2 System verilog testbench architecture

Purposes of testbench:

A testbench allows us to verify the functionality of a design through simulations. It is a container where the design is placed and driven with different input stimuli. The purpose of testbench is to determine the correctness of DUT which can be done using following steps.

1) Generate stimulus
2) Apply stimulus to the DUT
3) Capture the response
4) Check for correctness
5) Measure progress against the overall verification goals

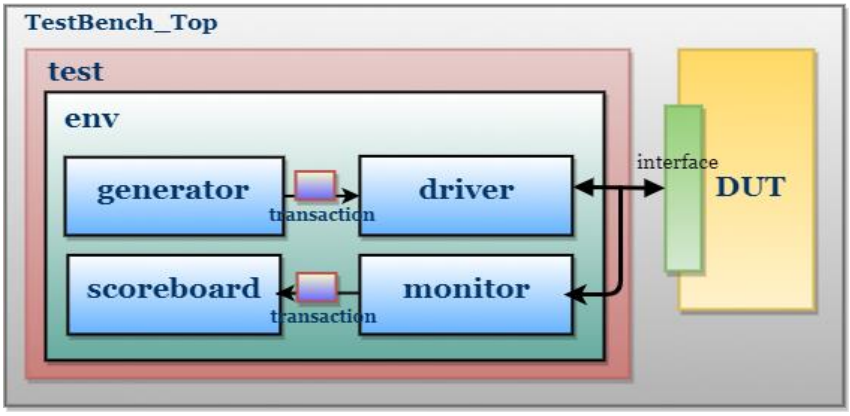

Fig 3.1 Testbench Architecture

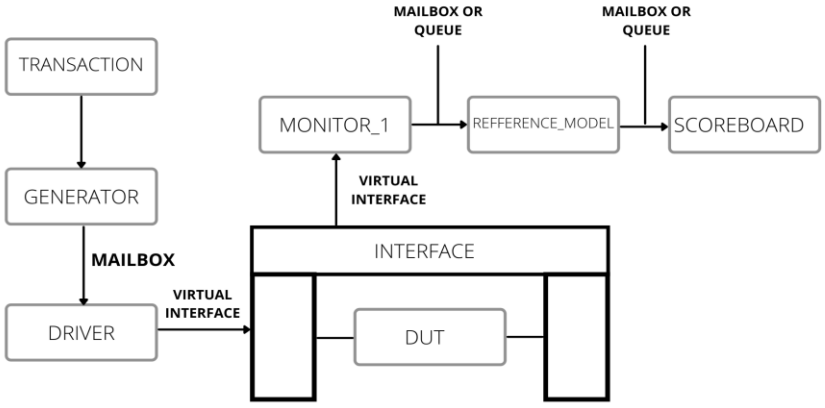

Fig 3.2 Flow of data/signal

In order to do this, the top level design module is instantiated within the test bench environment, and the design input/output ports are connected with the appropriate testbench component signals.

Interface

An interface is a way to encapsulate signals into a block.All the input/output port signals are bound together in a block (interface) which acts as a communication medium between DUT(Design Under Test) and testbench.

To specify the direction of the signal w.r.t module which uses interface instead of port list, modports are used.Modport restrict interface access within a module based on the direction declared.

i.e.

```
interface intf (input clk);
logic read, enable,
logic [7:0] addr,data;

modport dut (input read,enable,addr,output data);
modport tb (output read,enable,addr,input data);
endinterface :intf
```

Modports and interfaces by default do not specify any timing requirements or synchronization schemes between signals; Clocking block does exactly that.

A clocking block is a collection of signals synchronous with a particular block and helps to specify the timing requirements between clock and the signals.

i.e.

```
clocking cb @(posedge clk);
default input #10ns output #2ns;

output read,enable,addr;
input negedge data;
Endclocking
```

The second line specifies that by default all signals in the clocking block shall use a 10ns input skew and a 2ns output skew by default.Fourth line also contains negedge which overrides the skew ,so that data is sampled on the negedge of the clk.

Interfaces can contain tasks, functions, parameters, variables, functional coverage and assertions.This enable us to monitor and record the transactions via the interface within the block. It also becomes easier to connect regardless of the number of ports it has since that information is encapsulated in an interface.

Generator

The generator component generates stimuli which are sent to DUT by the driver. Stimulus generation is modeled to generate the stimulus based on the specification.

Stimulus generation can be directed or directed random or automatic and the user should have proper controllability from the test case. System Verilog provided construct to control the random generation distribution and order. Constraints defined in stimulus are combinational in nature whereas constraints defined in stimulus generators are sequential in nature.

Generally the generator should be able to generate every possible scenario and the user should be able to control the generation from directed and directed random test cases.

Driver

The drivers translate the operations produced by the generator into the actual inputs for the design under verification.

Generators create inputs at a high level of abstraction namely, as transactions and send them to the driver through the mailbox. The drivers convert this input into actual design inputs that the DUT can understand, as defined in the specification of the design interface.

When the driver has to drive some input values to the design , it simply has to call the predefined task in the interface without actually knowing the timing relation between these signals.The timing information defined within the task provided in the interface. This is the level of abstraction required to make testbenches more flexible and scalable.

Monitor

The DUT processes the input data and sends the results to the output pins. Monitor samples the interface signals and converts the signal level activity to transaction level and it sends the sampled transactions to Scoreboard via mailbox.

Monitor reports the protocol violation and identifies all the transactions.Monitors are two types, Passive and active. Passive monitors do not drive any signals. Active monitors can drive the DUT signals.

Scoreboard

The stimulus generator generated the random vectors and is sent to the DUT using drivers. These stimuli are stored in the scoreboard until the output comes out of the DUT. The Scoreboard can have a reference model which behaves the same way as the DUT.

Scoreboard basically compares the output of the reference model and the one that of DUT which it receives via mailbox from monitor. So ,if the DUT has a functional problem then the output from the DUT will not match with that of the reference model. So this way functional error can be detected and corrected further.Thus sometimes Scoreboard is referred as tracker.

Environment

Environment contains the instances of all the verification components and Component connectivity is also done.

It also contains the declarations of the virtual interfaces. Virtual interfaces are just handles(like pointers). When a virtual interface is declared, it only creates a handle. It does not create a real interface.

Constructor method should be declared with the virtual interfaces as arguments, so that when the object is created in the test case, new() method can pass the interface into the environment class where they are assigned to the local virtual interface handle. With this, the Environment class virtual interfaces are pointed to the physical interfaces which are declared in the top module.

Test

The test will instantiate an object of the environment and configure it the way the test wants.When we have more than hundreds of test it is not feasible to make direct changes to the environment for each test instead we want certain knobs/parameters in the environment that can be tweaked for the each test.

That way the test will have higher control over the stimulus generator and will be more effective.

Testbench_top

This is the topmost file, which connects the DUT and TestBench. It consists of DUT, Test and interface

3.3 EDA Playground:

EDA Playground gives engineers immediate hands-on exposure to simulating and synthesizing SystemVerilog, Verilog, VHDL, C++/SystemC, and other HDLs. All you need is a web browser.

- With a simple click, run your code and see console output in real time.
- View waves for your simulation using EPWave browser-based wave viewer.
- Save your code snippets (“Playgrounds”).
- Share your code and simulation results with a web link. Perfect for web forum discussions or emails. Great for asking questions or sharing your knowledge.

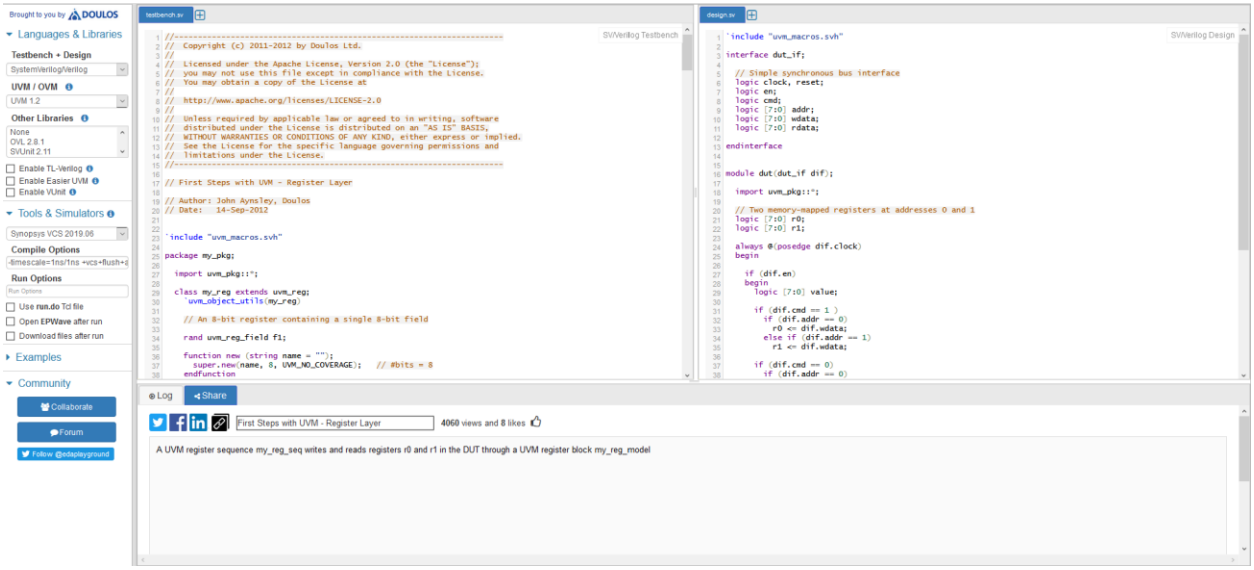

Figure 3.3 EDA Playground Testbench and Design editing Space

The main advantage of EDA is that it is Open source and cloud based platform. It allows beginners to do any kind of practice without taking burden of purchasing any tools. Also since it is cloud based and it offers cloud storage for whatever you have written in the code you can preserve it. Also one can share it with others makes them reliable.

3.4 Testbench Code:

As per discussed, System Verilog offers Object orientation programming concepts. So we can use class features of it. So by product we divided the testbench in several classes. First class is transection class, we discussed in section 3.2, it generates Random data packets for DUT. This can be achieved by using “rand” keyword.

Transaction class:

```
//creating the buch of inputs and binding it in class
`ifndef TRANS  //guard to avoid recompilation
`define TRANS

class transaction;
  bit [6:0]address= 7'b1010111;  //slave adress  the inputs
  bit [6:0]address1=7'b1011111;
  rand bit rw;   //randomizing the inputs
  rand bit rw1;
  rand bit [7:0] data;  //randomizing the inputs
  rand bit [7:0] data1;
  bit [7:0] r_data;
  bit [7:0] r_data1;
  bit[7:0] smp_data;
  bit[7:0] smp_data1;
  bit[7:0] smp_read;
  bit[7:0] smp_read1;

  function void display(string name);  //display function to observe the data
    $display("-------------------------");
    $display("- %s ",name);
    $display("-------------------------");
    $display("- address = %0d, r_w = %0b",address,rw);
    $display("- data = %0d",data);
    $display("- address1 = %0d, r_w1 = %0b",address1,rw1);
    $display("- data1 = %0d",data1);
    $display("-------------------------");
```

```
  if(rw)
     begin
     $display("- r_data=%d",r_data);
     $display("-------------------------");
   end
   if(rw1)
     begin
       $display("- r_data1 =%d",r_data1);
     $display("-------------------------");
    end
 endfunction
endclass //transection
`endif
```

Generator Class:

//generator class generates the data according to the defination of the transaction class and puts it into the mailboxes for driver

```
`ifndef GEN  //guard to avoid recompilation
`define GEN
class generator;

    mailbox gen2drv; // mailbox to connect the generator and driver
    transaction tr;  //instance of transaction class
    int no_tr;
  function new(mailbox gen2drv); //constructor for mailbox
       this.gen2drv=gen2drv;
    endfunction //new()
```

```
    event ended;
    task main();
      repeat (no_tr)begin  //no_tr times generates the input
        tr=new();  //constructiong thr tr object
        if(!tr.randomize())$fatal("gen:: trans randomization failed");  //cheking that randimization process is sucessfull or not
        tr.display("[generator]"); //display generated data
        gen2drv.put(tr);  //putting generated data into mailbox so it can reach at driver
        end
        ->ended; //triggering indicatesthe end of generation
    endtask

endclass //generator
`endif
```

Driver Class:

// driver devides buch of tinput(transection) according to the output of the dut it put the input data in the interface.

```
`ifndef DRV   //guard to avoid recompilation
`define DRV
class driver;
    mailbox gen2drv;  // mailbox to connect the generator and driver
    virtual intf d_intf; //interface instance to connect the testbech to the DUT
    int no_tr;
        bit m_select;
        bit reWr;         //read write bit selection
bit [7:0] data_select ; // data selection bit
bit [6:0]addbit;          // address selection bit
```

```
  function new(mailbox gen2drv,virtual intf d_intf,bit m_select);  //constructor for mailbox and interface
      this.d_intf=d_intf;
      this.gen2drv=gen2drv;
       this.m_select=m_select;
   endfunction //new()

   task reset;  // testing of reset

     wait(d_intf.rst);
      $display("[ DRIVER ] ----- Reset Started -----");
      d_intf.drv.cb_driver.sda <= 0;
      d_intf.drv.cb_driver.scl <= 0;
     wait(!d_intf.rst);
      $display("[ DRIVER ] ----- Reset Ended   -----");
   endtask

   task main;
      transaction tr; //instance of transaction class
      forever begin
        tr=new();
        gen2drv.get(tr);  //getting transection frommailbox of generator
       // d_intf.drv.cb_driver.sda <= 1; //starting condition
      //  d_intf.drv.cb_driver.scl <= 0;

        start_condition();
        address_send(tr);
         check_ack();
```

```
        if(tr.rw==1)
        begin
          fork
           generate_scl();  // to generate the scl (clock) for read operation parallaly
           read_byte(tr);
          join
            send_ack();
        end
        else begin
         data_send(tr);
          check_ack();
       data_send(tr);
         check_ack();

        end
        stop_condition();
         $finish;

      tr.display("[ Driver ]");  //displayig the content of input and output
        no_tr++;  //incrementing the number  of transection
     end
  endtask

task generate_scl();
  for(int i=0;i<8;i++)
    begin
       @(negedge d_intf.clk);
   d_intf.drv.cb_driver.scl <= 1;
```

```
      @(negedge d_intf.clk);
  d_intf.drv.cb_driver.scl <= 0;
    end
endtask

  task start_condition();
                        // @(negedge d_intf.clk);
          // $display("st_c = %t ",$time);
                         @(posedge d_intf.clk);
            d_intf.drv.cb_driver.sda <= 1; //starting condition
            d_intf.drv.cb_driver.scl <= 0;
              @(negedge d_intf.clk);
            d_intf.drv.cb_driver.scl <= 1;
              @(posedge d_intf.clk);
            d_intf.drv.cb_driver.sda <= 0;
              @(negedge d_intf.clk);    // address sending
            d_intf.drv.cb_driver.scl <= 0;

  endtask

task read_byte( transaction tr);
     for(int i=0;i<8;i++)
     begin
        @(posedge d_intf.clk);
      @(negedge d_intf.drv.cb_driver.scl);
     //  $display("time=%d",$time);
      tr.r_data[7-i]=d_intf.drv.cb_driver.SDA;
      //$display("databit=%b",tr.r_data[7-i]);
```

```
    end
endtask



task send_ack();
  @(posedge d_intf.clk);
  d_intf.drv.cb_driver.sda <= 0;
  @(negedge d_intf.clk);
  d_intf.drv.cb_driver.scl <= 1;
   @(negedge d_intf.clk);
  d_intf.drv.cb_driver.scl <= 0;
  @(posedge d_intf.clk);
   d_intf.drv.cb_driver.sda <= 1;
  @(posedge d_intf.clk);
endtask

task address_send(transaction tr);
  if(m_select)
   begin
      addbit <= tr.address1;
      reWr <= tr.rw1;
   end
  else
    begin
                addbit <= tr.address;
                reWr <= tr.rw;
```

```
end

 // $display("address send %t ",$time);
    @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= addbit[6];
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= addbit[5];
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= addbit[4];
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= addbit[3];
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
```

```
d_intf.drv.cb_driver.sda <= addbit[2];
   @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
   @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
   @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= addbit[1];
   @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
   @(negedge d_intf.clk)
d_intf.drv.cb_driver.scl <= 0;
   @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= addbit[0];
        @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
        @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
   @(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= reWr;
   @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
   @(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
                    @(posedge d_intf.clk);
            d_intf.drv.cb_driver.sda <= 1;
                //      $finish;
// wire end property
```

```
//for reading if we not put one cause problem...

 endtask
      //ack

 task check_ack();

          @(negedge d_intf.clk);
  //$display("ACK time=%d",$time);
          d_intf.drv.cb_driver.scl <= 1;
                              @(posedge d_intf.clk);
 // $display("_time=%t------value=%b",$time,d_intf.drv.cb_driver.SDA);
    wait(!d_intf.drv.cb_driver.SDA); #0;
                              @(negedge d_intf.clk);
                    //        $display("time=%d",$time);
           d_intf.drv.cb_driver.scl <= 0;
                              d_intf.drv.cb_driver.sda <= 0;
                              @(posedge d_intf.clk);
                              d_intf.drv.cb_driver.sda <= 1;
                              @(posedge d_intf.clk);
          //$display("end_time=%d",$time);
           //d_intf.drv.cb_driver.sda <= 0;
                    //$finish;

 endtask

 task data_send(transaction tr);
  if(m_select)
```

```
data_select = tr.data1;
else
data_select = tr.data;



    @(posedge d_intf.clk);
  //  $display("time_data_driver=%d",$time);
    d_intf.drv.cb_driver.sda <= data_select[7];
    @(negedge d_intf.clk);
    d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
    d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
    d_intf.drv.cb_driver.sda <= data_select[6];
    @(negedge d_intf.clk);
    d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
    d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
    d_intf.drv.cb_driver.sda <= data_select[5];
    @(negedge d_intf.clk);
    d_intf.drv.cb_driver.scl <= 1;
    @(negedge d_intf.clk);
    d_intf.drv.cb_driver.scl <= 0;
    @(posedge d_intf.clk);
    d_intf.drv.cb_driver.sda <= data_select[4];
    @(negedge d_intf.clk);
```

```
d_intf.drv.cb_driver.scl <= 1;
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
@(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= data_select[3];
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
@(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= data_select[2];
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
@(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= data_select[1];
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
@(posedge d_intf.clk);
d_intf.drv.cb_driver.sda <= data_select[0];
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 1;
@(negedge d_intf.clk);
d_intf.drv.cb_driver.scl <= 0;
@(posedge d_intf.clk);
```

```
                    d_intf.drv.cb_driver.sda <=0;
    endtask
    task stop_condition();
                    //$display("stop_condition=%d",$time);
                        d_intf.drv.cb_driver.sda <= 0;
        @(negedge d_intf.clk);
        d_intf.drv.cb_driver.scl <= 1;
        @(posedge d_intf.clk);
        d_intf.drv.cb_driver.sda <= 1;
        @(negedge d_intf.clk);
        d_intf.drv.cb_driver.scl <= 0;
                        //$display("stop_condition=%d",$time);
    endtask

endclass //driver
`endif
```

Interface:

```
//interface is connection between the DUT and the

// Code your testbench here
// or browse Examples
`ifndef INTF ////guard to avoid recompilation
`define INTF
interface intf(input logic clk,rst);  //interface argument we are providing the clk and rst externally
..
  logic scl; //interface connection
  logic sda;
  logic [7:0] out;
```

```
logic [7:0] read;
logic SDA ; // to sample the data
 // bit rst;
wire sda_out;
clocking cb_test @(clk);
      inout sda;
     output scl;
     input SDA;
endclocking

clocking cb_driver @(clk);  //clcking block for synchronization
      inout scl;
      inout sda;
      input out;
      input read;
      input SDA;
endclocking

clocking cb_monitor @(clk);  //clcking block for synchronization
  input scl;
  input sda;
  input out;
  input read;
  input SDA;
endclocking

//modport DUT ( input a,b,cin,clk,rst, output sum,carry );
modport drv (clocking cb_driver,input rst);  //modport is usefull to define the direction like input/output for the perticular connection
```

```
  modport mon (clocking cb_monitor,input rst ); //
  modport test (clocking cb_test);
endinterface
`endif
```

Monitor Class:

// monitor class is for to monitor the output of the dut according to the input ans send it into the scoreboard so it can check the correctness of the functionality

```
`ifndef MON   //guard to avoid recompilation
`define MON
class monitor;
    virtual intf m_intf;  //interface to get the input and output  information from the DUT
     mailbox mon2scb;  // maibox to send the monitored data in to the scoreboard

  function new(virtual intf m_intf,mailbox m_mbox);  //constructor for mailbox and interface
    this.m_intf = m_intf;
    this.mon2scb = m_mbox;
    endfunction //new()

    task main;
        transaction tr;  //instance of transaction class
      forever begin
          @(negedge m_intf.mon.cb_monitor.sda);
          if(m_intf.mon.cb_monitor.scl==1)
          begin
             tr=new();
             @(negedge m_intf.mon.cb_monitor.scl);
             for(int i=0;i<7;i++)begin
```

```
    @(negedge m_intf.mon.cb_monitor.scl);
    tr.address[6-i]= m_intf.mon.cb_monitor.sda;
end
@(negedge m_intf.mon.cb_monitor.scl);
 tr.rw = m_intf.mon.cb_monitor.sda;

@(negedge m_intf.mon.cb_monitor.scl);  //skip adress ack

if(tr.rw==0)begin
    for(int i=0;i<8;i++)begin
        @(negedge m_intf.mon.cb_monitor.scl);
        tr.data[7-i] = m_intf.mon.cb_monitor.sda;

    end
     @(negedge m_intf.mon.cb_monitor.scl);
    tr.smp_data = m_intf.mon.cb_monitor.out;

end
else begin
    for(int i=0;i<8;i++)begin
        @(negedge m_intf.mon.cb_monitor.scl);
      tr.r_data[7-i] = m_intf.mon.cb_monitor.SDA;
    end
    @(negedge m_intf.mon.cb_monitor.scl);
  @(posedge m_intf.mon.cb_monitor.scl);
    tr.smp_read=m_intf.mon.cb_monitor.read;
  $display("r_data=%d time=%d",tr.smp_read,$time);
end
```

```
            //skip data ack

            mon2scb.put(tr);
         end
        end
         //  tr.display("monitor");  //display the content of the monitor.
      endtask
  endclass //monitor
  `endif
```

Scoreboard Class:

```
// scoreboard is for to check the correctens of the operation perfomed by the DUT on inputs..

`ifndef SB ////guard to avoid recompilation
`define SB
class scoreboard;

   mailbox mon2scb;  //mailbox toget the data from the monitor

   int no_tr;  // no of trnasection for the scoreboard..
 function new(mailbox mon2scb);  //constructor  for mailbox.
   this.mon2scb=mon2scb;//
   endfunction //new()

   task main;
                  transaction tr;  //transaction instance for the scoreboard
```

```
 tr = new();

forever begin

 mon2scb.get(tr);

// compare logic.....
 if(tr.rw==0)
   begin
     if(tr.smp_data==tr.data)
       begin
         $display("------------------write_test_pass----------------");
         $display("actual data=%d   expected data=%d",tr.smp_data,tr.data);
         $display("-------------------------------------------");
        // $display("check_time=%d",$time);
       end
       else begin
         $display("***********write_test_fail***********");
         $display("actual data=%d   expected data=%d",tr.smp_data,tr.data);
       end
   end
 else
   begin
    if(tr.r_data==tr.smp_read)
       begin
         $display("------------------read_test_pass----------------");
         $display("actual data=%d   expected data=%d",tr.r_data,tr.smp_read);
         $display("-------------------------------------------");
```

```
            // $display("check_time=%d",$time);
           end
           else begin
             $display("**********read_test_fail**********");
             $display("actual data=%d   expected data=%d",tr.r_data,tr.smp_read);
           end
         end
       no_tr++;
      end

   endtask

endclass //scoreboard
`endif
```

Test Class:

```
// test case for teating the DUT

//import package files::*;
`include "environment.sv"

program rd_test(intf t_intf,intf t_intf1);  //  program module which is creating the test
   bit flag =0;
   environment env,env1;
   virtual intf tbm,tbm1;

initial begin
   tbm=t_intf;
```

```
  tbm1=t_intf1;
 env = new(t_intf,0);  //constructor for environment
  env.gen.no_tr=1; // number of transections are 7
 env1 = new(t_intf1,1);
        env1.gen.no_tr=1;
 fork
  begin :master1
        env.run();
  end//3in the environment
  begin :master2
        env1.run();
  end
 join_none
  forever
   begin
    @(tbm.clk)
    begin
       if(tbm.test.cb_test.sda!=tbm.test.cb_test.SDA                    ||
tbm1.test.cb_test.sda!=tbm1.test.cb_test.SDA)
         begin
          if(tbm.test.cb_test.sda==1 && flag==0)
            begin
              tbm.test.cb_test.sda<=1;
              tbm.test.cb_test.scl<=1;
              disable master1;
              flag=1;
              $display("arbitration of master 1");
            end
          else
```

```
            begin
              if(flag==0)
              begin
                tbm1.test.cb_test.sda<=1;
                tbm1.test.cb_test.scl<=1;
                disable master2;
                flag=1;
                $display("arbitration of master 2");
              end
            end
          end
      end
    end
end
endprogram
//$display(" sda = %b SDA = %b time = %t",tbm.sda,tbm1.SDA,$time);
```

Testbench Class:

```
`include "environment.sv" //including the file
`include "test.sv"
`include "interface.sv"
module tb;
bit clk;
bit rst;
   always #1250 clk=~clk;  //clock generator

initial begin
   rst=1;    //reseting the DUT
```

```
    #2500 rst=0;
end


  intf tb_intf(clk,rst);
  //intf tbm_intf(clk,rst);
//intf.DUT dut_intf(clk);
  intf tb_intf1(clk,rst);
  rd_test t(tb_intf,tb_intf1);  //creating the instance of the test and giving it's interface  as argument
    AND a(.out(tb_intf.sda_out),.sda_1(tb_intf.sda),.sda_2(tb_intf1.sda));
    Slave
DUT(.SCL(tb_intf.scl&tb_intf1.scl),.SDA(tb_intf.sda_out),.DATA_out(tb_intf.out),.DATA_rea
d(tb_intf.read),.sample_sda(tb_intf.SDA),.sample_sda1(tb_intf1.SDA));
    initial begin
  $dumpfile("dump.vcd");
  $dumpvars;
          end
  endmodule
```

3.5 Final Results – Waveforms:

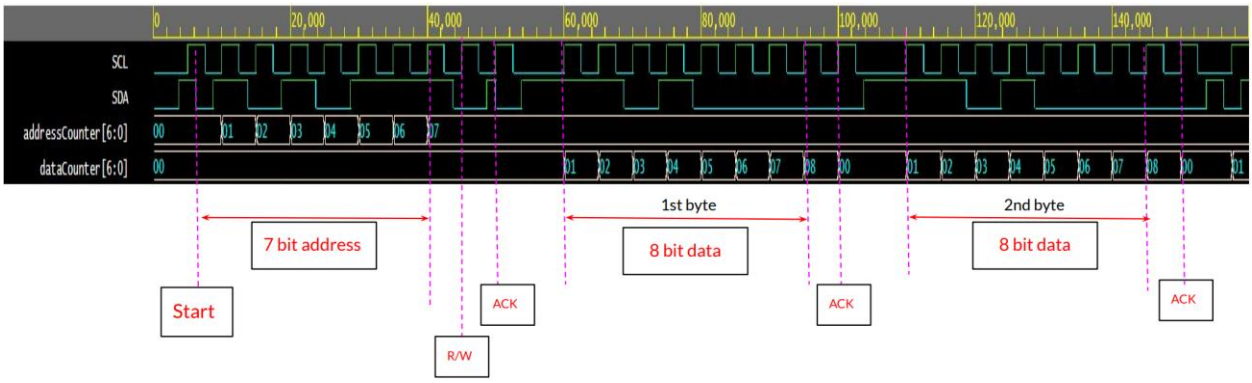

Figure 3.4 Successful Multi-byte Operation

Figure 3.4 shows multi-byte operation, It can be seen that design follows exact I2C Protocol. First of all, whenever protocol detects fall edge during high level of SCL, protocol consider it as a START condition. After that master is sending 7 bit address on the SDA line. Once address matched with particular slave, slave has to acknowledge with low level logic. In figure 3.4, it can be seen that R/W bit and ACK, both are low logic that means master is in write mode and slave has been selected which available on SDA line. Now after receiving acknowledge from slave, master will send 8 bit data on SDA line. After receiving 8 bit data slave will acknowledge it to master. Similarly process will continue until master creates STOP Condition.

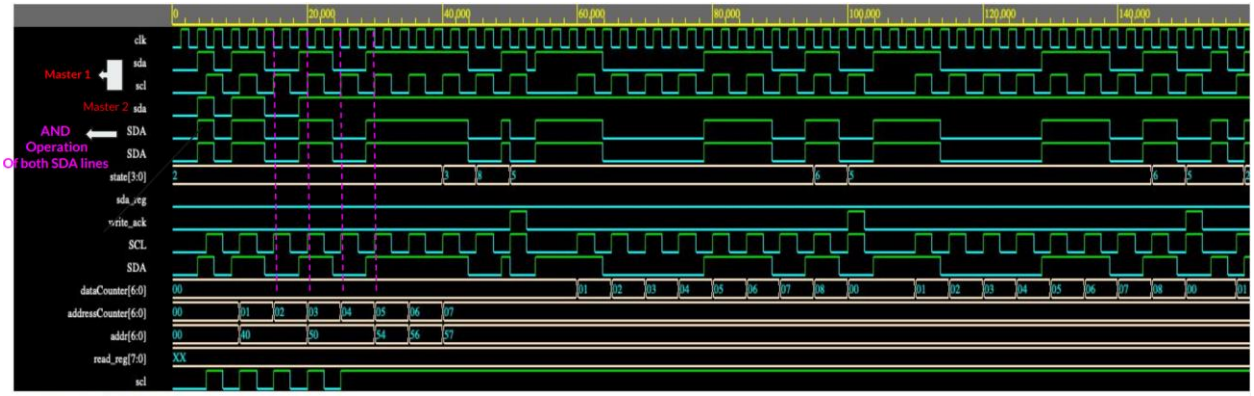

Figure 3.5 Multi master Capability – Write Operation with Arbitration

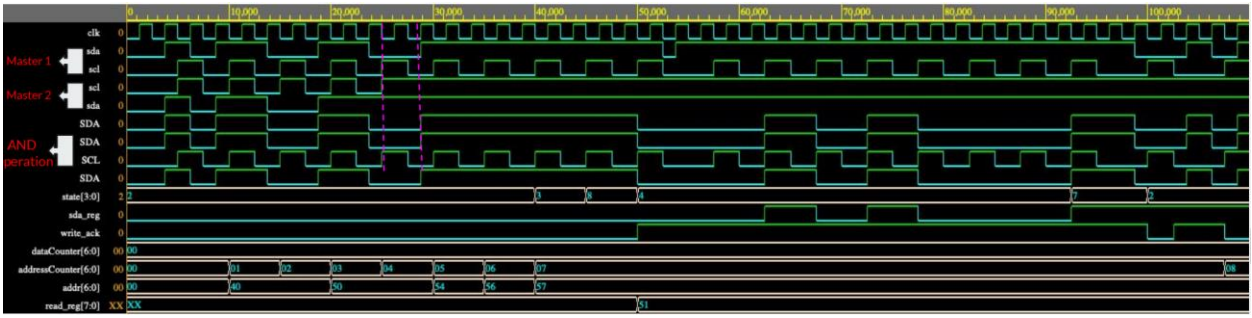

Figure 3.6 Multi master Capability – Read Operation with Arbitration

Figure 3.5 shows write operation, when two masters are detecting SDA line idle simultaneously. They both try to send the data. So to avoid it, Arbitration can be implemented. Figure shows that both masters SDA line gets logical AND operation and according to that one master gets complete control on SDA line. (Highlighted in figure 3.5) same way figure 3.6 shows read operation.

TROUBLESHOOTING

Problem:-How both master and slave can drive the bus same time
Solution: - In the I2C protocol we are using the SDA line to receive and transmit data. So master and slave both can drive the bus (one at a time) so it should be bidirectional. So that we can define it as INOUT in design as well as in TB

Problem: - how to perform wire end operation
Solution: - Here the i2c protocol follows the END(wire) functionality so that we need to use wire (triend, wand) data type for that. It will perform an operation between two simultaneous drives which is happening at both ends of the wire. In i2c if we are not performing the operation on the bus make the bus high and leave it.

Problem: - not able to drive the data on sda line in design
Solution: - our SDA line is wire (net type) so that without using an assign statement we cannot do that. For that we used the reg to change the data of sda and assign that reg to wire. We can also use(pull0,pull1) syntax. There are some other that we can see in the verilog concept document. Solution found on:

NOTE: - Wait keyword is level sensitive (Not edge) (issue was in ack task function)

Problem: - whatever we add in ack function after the wait statement that was reflecting before the wait operation.
Solution: - there was a problem with the wait statement in driver ack function that whatever we write with that was executing parallely(means waiting process and changing the data process happens concurrently). To solve that issue we can use separate wait statements and never use other operations with that. Otherwise problems will occur in the synchronization.

Note: - Always use non-blocking (<=) in driver class.

Problem: - No *.vcd file found. EPWave will not open. Did you use '$dumpfile("dump.vcd") ; $dumpvars;'? Even though we have included syntax for that
Solution: - it means that our simulation ends at 0 time. So in this kind of scenario look at the code and rectify the problem.

Note: - In interface, the clocking block should be negative as well as positive edge sensitivity. If we are using negative as well as positive otherwise we would not be able to use negative edges or whatever we have not specified in the sensitivity list that also need to include in the clocking block.

Problem: After the ack cycle of the clock, the data bit should change before the next SCL clock cycle comes. But it was not happening according our written design

Issue: SDA line is changing at the same pose edge of SCL

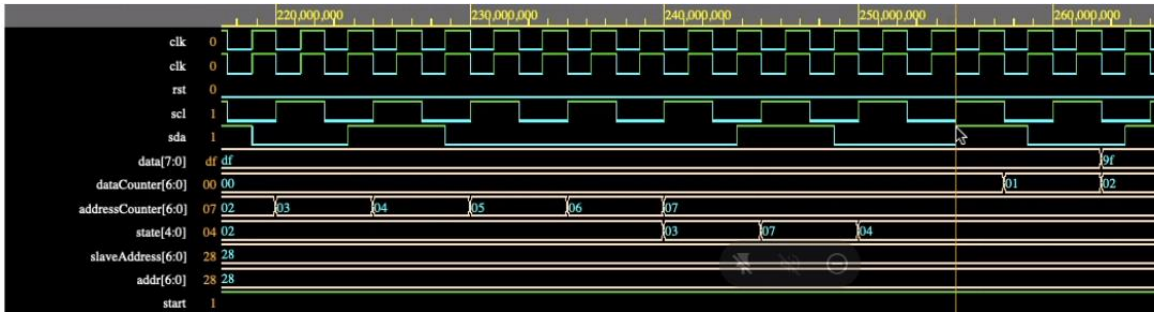

Figure 4.1 Solved

Solution:

We added one another reference clock which can be used as data manipulation. By sampling the data at the negative edge of SCL.
Whenever you define a module, define a reference clock with SCL, so data changing can take place very easily with reference clk.

Problem: When slave is reading multiple bytes from SDA line, after ACK of address clock cycle it is detecting next data bit as stop condition. Also data change is not taking place at a low level of SCL.
Solution: - We were sampling the data at neg edge, and from the reference figure, then to solve the problem we sampled data at posedge. But at the time of ack we take back control from slave at neg edge of ack cycle

Note:- According to the protocol sampling of the ACK of write operation is at the posedge and everything should happen at posedge according to the fig so now we want access back after the negedge to the master …

Problem: - in read operation we have the same problem because in that slave is driving the data so it will drive the data at posedge so the stop condition is occurring at that time

Solution :- now in that reading operation we will not check the start and stop condition because it is the responsibility of the master to generate it so at that time we will check that \who is driving the bus if wartyback==0 then only check for the start and stop condition

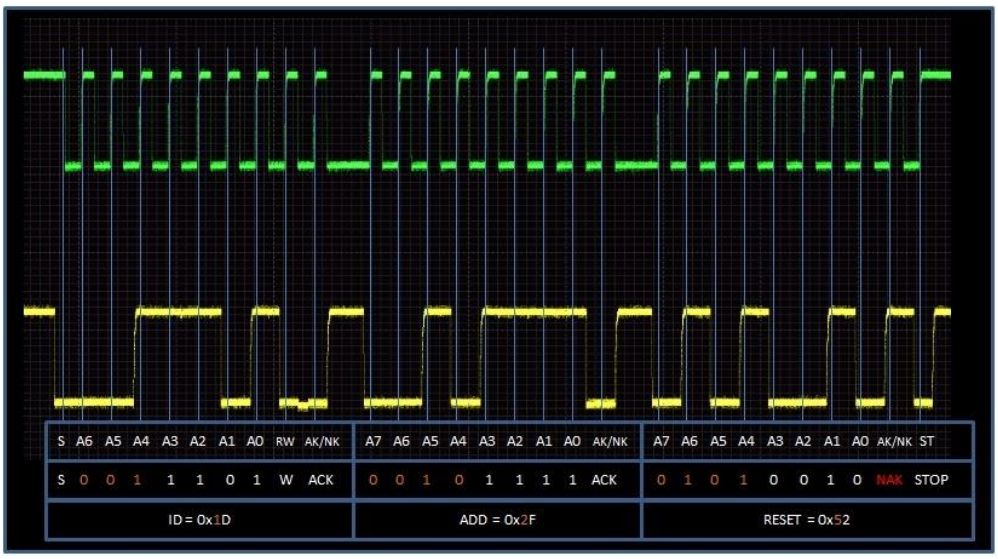

Figure 4.2 Actually I2C Protocol response on DSO

Problem: - While writing the randomized constraints for multi slave, error pops up Illegal combination of structural drivers in this case we wanted to implement multiple driver and wire end properties. But we cannot drive one wire simultaneously with two drivers. If we do that then it performs original operation and gives priority to logic 1.

Issue with Problem: we are trying to do arbitration for Multimeter, so we were doing "AND" Operation of SCL of both masters. For that we were trying to write and operation inside the driver class. But after simulation it throws an error that, we cannot use "assign" statement in a **class**

After some Googling we found that: (NOTE)

No. you cannot use an always block inside any procedural code, including a task. An always block implements the following two concepts:

- **It creates a process thread by execution of the procedural code within the block.**
- **Once the procedural block completes, it repeats execution of the procedural block indefinitely. That process continues until the end of the simulation.**

An initial block does only implements the first concept. Once the procedural block completes, that process terminates.

Solution: - If you need an infinite loop, you can use the procedural forever looping statement. That can be used anywhere a procedural statement is allowed, including inside a task.

Final solution: - Multi Master Capability

Idea: We were doing AND operation in Testbench for master switching, instead of that we should define AND operation in design itself And give the output to the dut.

Here every environment is acting like a master.

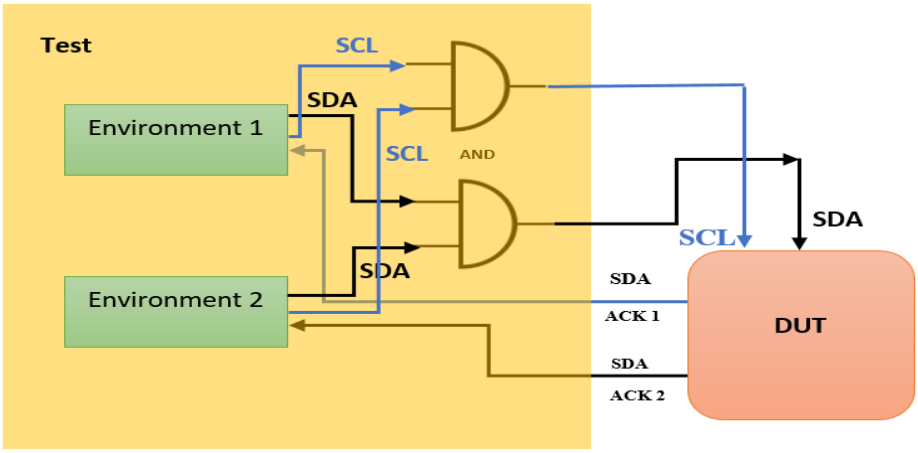

Figure 4.3 Arbitration using AND operation

Problem:- Now here SDA is bidirectional so through the AND gate we can not sample whatever data that DUT wants to send to the master.
Solution:- so we are defining two more signals as output signal SDA(sampling 1) And SDA(sampling 2) they are the same signal as SDA so that we can send the same data to both masters for observation as you can see in the above fig.

To perform and operation there is two way

1. We can do anding in instantiation of DUT like shown below

```
Slave  DUT(.SCL(tb_intf.scl&tb_intf1.scl),.SDA(tb_intf.sda&tb_intf1.sda),
        .DATA_out(tb_intf.out), .DATA_read(tb_intf.read),
      .sample_sda(tb_intf.SDA),  .sample_sda1(tb_intf1.SDA));
```

2. We can define module for and and instantiate in the TB and send the output of that in DUT like shown below

```
AND a(.out(tb_intf.sda_out),.sda_1(tb_intf.sda),.sda_2(tb_intf1.sda));

Slave   DUT(.SCL(tb_intf.scl&tb_intf1.scl),. SDA(tb_intf.sda_out),
.DATA_out(tb_intf.out), .DATA_read(tb_intf.read), .sample_sda(tb_intf.SDA),
.sample_sda1(tb_intf1.SDA));
```

Problem: - How to perform arbitration

Earlier version:-

```
task automatic run;
      fork
        begin

          pre_test(); //run all test one by one
            test();
            post_test();
            $finish;
        end
      forever
      begin
        @(posedge e_intf.clk);
        $display("%b sda = %b SDA = %b time = %t",m_select,e_intf.sda,e_intf.SDA,$time);
        if(e_intf.env.cb_env.sda!=e_intf.env.cb_env.SDA)
          begin
            $display("quit time = %t",$time);
            e_intf.env.cb_env.sda<=1'b1;
              disable run;
          end
      end
      join
     endtask
```

Two instantiations of the environment were there. Arbitration logic (method of leaving or breaking the task) were defined in the definition of the environment but when we run both environment together both were leaving(disabling) together even though there was a arbitration of only one environment

To solve this problem we refer to these sites and find the solution.

Implementing Parallel Processing and Fine Control in Design Verification

System Verilog defines a built-in class “Process” that provides users with fine control over the processes. It allows users to define variables of type process and pass them through tasks.

The methods provided in process class to let users control the processes are,

self ();
kill ();

```
await ();
suspend ();
resume ();
status ();
```

New version :-

We found that we can't disable the task from the task itself if we want to get our desired output then will have to put the arbitration logic somewhere else so that we can run parallely the both environment and also the process for checking of both SDA lines for the arbitration. For that we define arbitration in the TEST as shown below.

```
fork
   begin :master1
     env.run();
   end//3in the environment
   begin :master2
     env1.run();
   end
  join_none
   forever
    begin
      @(tbm.clk)
      begin
If ( tbm.test.cb_test.sda != tbm.test.cb_test.SDA || tbm1.test.cb_test.sda != tbm1.test.cb_test.SDA )
          begin
            if(tbm.test.cb_test.sda==1 && flag==0)
              begin
                tbm.test.cb_test.sda<=1;
                tbm.test.cb_test.scl<=1;
                disable master1;
                flag=1;
                $display("arbitration of master 1");
              end
            else
              begin
                if(flag==0)
                begin
                  tbm1.test.cb_test.sda<=1;
                  tbm1.test.cb_test.scl<=1;
                  disable master2;
                  flag=1;
                  $display("arbitration of master 2");
```

```
            end
          end
        end
      end
    end
```

/*Caution: if we apply this idea , we have to take care that we have defined a testbench such that "whenever the mode of data **writing/reading** switches, the SDA line should be high", after applying this idea to this AND operation may affect the overall result.*/ i don't understand this whatever you have written.

FUTURE SCOPE

Here as a part of term project we have done up to basic designing of I2C slave that includes functionality of multiple byte write and single byte read by master.
At the verification side we cover the testing of multiple write and single byte read functionality as well as multimaster capability of i2c communication(at same clock frequency).

So in the future the project can be extended and below mentioned things can be added.

1. In I2C slave design buffers can be added so that the peripheral can store multiple bytes.
2. In I2C multiple byte read functionality can be added.
3. In I2C design 10 bit mode also can bi added.
4. In verification , 10 bit capability verification can be done.
5. In verification, multimaster capability at different clock frequencies (clock stretching) can be added.
6. And there is also scope of improvement in the structure of the testbench.

CONCLUSION

The continuous growth in complexity of electronic designs requires a modern, systematic, and automated approach to creating test benches. The cost of fixing a bug grows by tenfold as a project moves from each step of specification to RTL coding, gate synthesis, fabrication, and finally into the user's hands. Directed tests only test one feature at a time and cannot create the complex stimulus and configurations that the device would be subjected to in the real world. To produce robust designs, you must use constrained-random stimulus combined with functional coverage to create the widest possible range of stimuli.

Here, After I2C designing, design worked quite responsive but after testing it by using Test bench we found many functional bugs and logic errors (as discussed in Trouble shooting). System Verilog provided very flexible test bench writing with OOPs concepts. Also system Verilog features with new vast variation of data types. Some data types have 4 states (1, 0, X, Z) and some have 2 states (1, 0). Also system Verilog allows dynamic memory allocation of arrays in run time of the code. These features were not available in Verilog HDL. In Verilog, for inserting verification test vectors we had to define everything manually but here system Verilog, we can define constrained and randomized input test vectors which reduces redundancy of code.

REFERENCES

[1] System Verilog for Verification by Chris Spear & Greg Tumbush 3rd edition

ISBN 978-1-4614-0714-0, Springer New York Dordrecht Heidelberg London

[2] UM10204 I2C –Bus Specification and User Manual

Rev. 6 — 4 April 2014, NXP Semiconductors

[3] Open source System Verilog Tutorials - Testbench.in

[4] Simulating Platform: EDA Playground by DOULOS

[5] Verilog HDL by Samir Palnitkar

SunSoft Press 1996

[6] For solving majority bugs and finding new System Verilog Concepts

SIEMENS – Verification Academy (Forum)