P - 1396

SIMULATION AND SYNTHESIS OF INTER-INTEGRATED CIRCUIT BUS

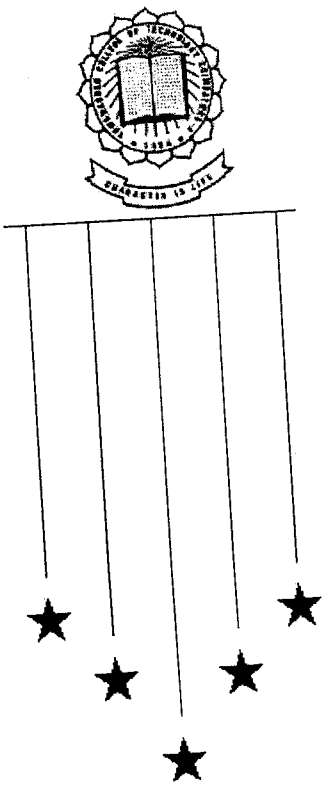
PROJECT REPORT 2002 - 03

Submitted by,

THARANI.B.S

MANJU.C.S

JYOTHIKA.M



Under the Guidance of

Mrs.SHANTHI. B.E., M.S.,

Submitted in partial fulfillment

Of the requirement for the degree of

BACHELOR OF ENGINEERING in the

Electronics & Communication Engg., Branch of the

BHARATHIAR UNIVERSITY,

COIMBATORE.

Department of Electronics & Communication Engineering

KUMARAGURU COLLEGE OF TECHNOLOGY

COIMBATORE - 641006

Department of Electronics & communication Engineering

KUMARAGURU COLLEGE OF TECHNOLOGY

Coimbatore - 641006

PROJECT REPORT 2002 – 03

CERTIFICATE

This is to certify that the report entitled "SIMULATION AND SYNTHESIS OF INTER INTEGRATED BUS" has been submitted by

JYOTHIKA.M, Roll.No. 99ECE13,

MANJU.C.S, Roll.No. 99ECE21,

THARANI.B.S, Roll.No. 99ECE56

in partial fulfilment for the award of BACHELOR OF ENGINEERING in Electronics and communication Engineering branch of the Bharathiyar University,Coimbatore-641006 during the academic year 2002-03.

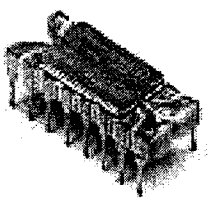
Guide

Head of Department

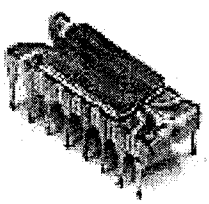
The candidates were examined by us in the Project work Viva-voce Examination held on 18.3.2003 and their University Register Numbers are 9927D0130, 9927D0138 and 9927D0173.

Internal Examiner

External Examiner



CERTIFICATE



ACKNOWLEDGEMENT

ACKNOWLEDGEMENT

In all our humility and sincerity, we express our gratitude to all those whohave helped us for the completion of this project.

We use this opportunity to thank our Principal., Dr.K.K.PADHMANABHAN, B.Sc(Engg.)., M.E., Ph.D., Kumaraguru college of technology, coimbatore-6, for his kind patronage.

We express our sincere gratitude to Prof.MUTHURAMAN RAMASAMY, M.E., M.I.S.T.E., M.I.E.E.E., M.I.E., C.(Engg)., M.B.M.E.S.I., (Ph.D), H.O.D of Electronics and Communication Engineering Department for the enormous help and guidance rendered by him to complete our project successfully.

We are grateful and immensely indebted to our project guide Sr.Lecturer., Mrs.M.SHANTI, M.S., a benevolent personality and Asst.Prof. Mr.S.GOVINDARAJU, M.E., a charismatic personality, for their able guidance and motivation, in situations, when our project was in rough waters.

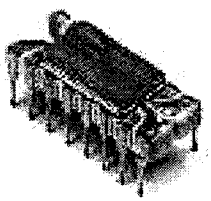
We are proud indeed to thank and be grateful to our class advisor, Lecturer., Mrs.G.AMIRTHA GOWRI, B.E., an agile and dynamic person for her concern and motivation through the years.

We are very much grateful to all staff members of Electronics and Communication Engineering Department for their kind support during the course of our project.

We take this opportunity to thank Mr.BALAMURUGAN, B.E. and

Mr.SAI VIJAYANAND, B.E., who are the men behind us in making us do the project. We are indeed greatly indebted to them.

Words can but in a meager way, our thanks to all lab assistants, who have put in a lot of efforts in order to give us the right kind of tools and other important facilities. Last but not the least we wish to express our humble thanks to the staffs of various department and ofcourse our fellow classmates who made our lives on campus a cherishable experience which would be indelibly imprinted in our hearts for posterity.



SYNOPSIS

SYNOPSIS

Our project work entitled "Simulation and synthesis of inter integrated circuit bus" has been developed using VERILOG HDL. Our project deals with the design of a simple bi-directional two-wire bus efficient for inter-IC control. All I2C compatible devices incorporate an on-chip interface, which allows them to communicate directly with each other via the I2C –bus. This design concept solves many interfacing problems.

DESIGN DESCRIPTION:

This project describes in detail the design of an I2C-Bus Model, having master models and slave models. The buses are the Serial Data Line (SDA) and the Serial Clock Line (SCL). Serial data bus carries information from one device to another, in synchronization with the clock in the serial clock line (SCL). The main process involved in I2C-bus is arbitration.

Arbitration process: (Designed with devices having different clock rates and same clock rate)

Arbitration takes place on the SDA line, while the SCL is at the HIGH level, in such a way that the master which transmits a HIGH level, while another master is transmitting a LOW level will switch off its DATA output stage because the level on the bus doesn't correspond to its own level.

Design description order:

The design description is ordered like,

- Detail design description of the I2C Master.
- Detail design description of the I2C Slave.
- Clock module.
- Test bench.
- Top module.

- **Master :**

The master initiates the transfer by issuing a START condition. It provides the clock for data transfer through SCL. It sends the slave 7-bit address together with READ/WRITE signal.

- **Slave:**

The slave module is an EEPROM and it acknowledges the data bytes send by the master and sends data when master acts as the receiver.

- **Clock generation module:**

This block generates the SCL clock while the master is initiating a data transfer in the Bus. The clock division number for all three modes of operation is given as a parameter to this block and should be given at the time of instantiation of the model.

- **Test bench:**

The external signals START and RESET are given to the system through the test bench module.

- **Topmodule:**

This module instantiates all other modules and connects all other modules through wires.

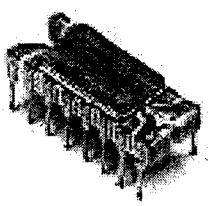
Mode of working:

Our project has been designed for the standard mode. The Standard mode works at a bit rate of 100Kbit/s.

Requirements:

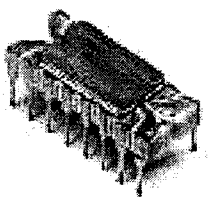
Simulator : Model Sim for simulation.

Synthesizer : Leonardo Spectrum.



CONTENTS

CONTENTS



INTRODUCTION

CHAPTER-1

1. INTRODUCTION:

The specifications of serial EEPROM and I2c BUS are matched ie., they are compatible and hence simulation can be done considering all aspects.

The Inter-Integrated Controller Bus is a bi-directional serial bus, which transmits one bit at a time. It is a multi master bus which can avoid collision and provides for arbitration when more than one master is connected.

The serial EEPROM basically consists of 128- 8 bit registers. Data can be transmitted or received in a serial manner. 7 bits of address are required to address 128 registers. These data are sent from the I2C Bus.

The above two can be interconnected and data transfer is accomplished. Additional chips can be added to the I2C Bus. The simulation is first accomplished for the read and writes cycle. The cycle is then extended to sequential read and sequential write cycle. The read cycle consists of transfer of 8 bits of address and then retrieval of 8 bits of clocked data. Write cycle consists of transfer of 8 bits of address and then serially clocked data. The sequential cycle consists of 32 bits of data and address bytes.

The necessity of faster speed can be achieved using a combination of serial EEPROM and I2C Bus. As I2C Bus is compatible with the serial EEPROM, the combination improves system performance.

1.1 PROLOGUE TO VERILOG HDL:

With the advent of VLSI (Very Large Scale Integration) technology designers could design chips with more than 100,000 transistors. Because of the complexity of the circuits, it was not possible to verify these circuits on a Breadboard. Computer aided techniques became critical for verification and design of VLSI circuits. The designers were now building blocks and then derive higher-level blocks from them. This process would continue till they build the top- level block, logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip. One such simulation language developed by Gateway Design Automation is VERILOG.

1.1.1 POPULARITY OF VERILOG HDL:

VERILOG HDL has evolved as a standard description language. VERILOG HDL offers many useful features for hardware design.

- VERILOG is general-purpose hardware description language that is easy to learn and to use. It is similar to the C Programming language.
- VERILOG HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware in terms of switches, gates, Register Transfer level (RTL) or behavioral code. Also, a designer needs only one language for stimulus and hierarchical design.
- All fabrication vendors provide VERILOG HDL libraries for post logic synthesis simulation. Thus, designing a chip in VERILOG HDL provides the widest choice of vendors.

- The programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of VERILOG. Designers can customize a VERILOG HDL simulator to their needs with the PLI.

1.2 TRENDS IN HDL's:

The speed and complexity of digital circuits has increased rapidly. Designers respond by designing at higher levels of abstraction. Designers have to think only in terms of functionality. CAD tools take care of the implementation details. With designer assistance, CAD tools have become sophisticated enough to do a optimum implementation.

The most popular trend currently is to design in HDL at an RTL level, because logic synthesis tools can create gate-level net lists from RTL level design. Behavioral synthesis has recently emerged. As those tools improve, designers will be able to design directly in terms of algorithms and behavior of the circuit and then use CAD tools to do the translation and optimization in each phase of the design. Behavioral modeling will be used more and more as behavioral synthesis matures, until then RTL design will remain popular.

A trend that is emerging for system-level design is a mixed bottom-up methodology where the designers use either existing VERILOG HDL modules, basic building blocks or vendor supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules.

1.3 DESIGN METHODOLOGIES:

There are two basic types of digital methodologies:

- A top-down design methodology and a bottom -up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further sub-divide the sub-blocks until we come to leaf cells, which are cells that cannot further be divided.
- In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells using these building blocks. These cells can then be used for higher-level block in the design.

Typically, a combination of top-down and bottom-up flows is used. Design architects define the specification of the top-level block. Logic designers decide architects how the design should be structured by breaking up the functionality into blocks and sub-blocks. At the same time, circuit designers are designing optimized circuits for leaf-level cells. They build higher-level cells by using these leaf cells. The flow meets at the intermediate point where the switch-level circuit designers have created a library of leaf cells by using switches, and the logic level designers have designed from designed from top-down until all modules are defined in terms of leaf cells.

1.4 MODULES AND INSTANCES:

VERILOG provides the concept of a module. A module is the basic building block in VERILOG. A module can be an element or a collection of lower-level design blocks. Typically elements are grouped into modules to provide common functionality that is used at many places in the design. A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting rest of the design.

In VERILOG, the keyword module declares a module. A corresponding end module must appear at the end of the module definition. Each module must have a module name, which is the identifier for the module terminal list, which describes the input and output terminals of the module.

VERILOG is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The module behaves identically with the external environment irrespective of the level of abstraction at which the nodule is described. The internals to describe a module can be changed without any change in the environment.

A module provides a template from which actual objects are created. When a module is invoked, VERILOG creates a unique object from the template. Each object has its own name, variables, parameters and I\o interfaces. The process of creating objects from a module template is called instantiation and the objects are called instances.

In VERILOG it is illegal to nest modules. One module definition cannot contain another module definition within the module and end module statements. Instead, a module definition can incorporate copies of other modules by instantiating them. It is important not to confuse module definitions and instances of a module. Module definitions simply specify how the module will work, its internals and its interfaces. Modules must be instantiated for use in the design.

1.5 LEVELS OF ABSTRACTION:

- **Behavioral or algorithmic level:**

This is the highest level of abstraction provided by VERILOG HDL. A module can be implemented in terms of desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

- **Dataflow level:**

At this level the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

- **Gate level:**

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is very similar to describing a design in terms of a gate-level logic diagram.

- Switch level:

This is the lowest level of abstraction provided by VERILOG. A module can be implemented in terms of switches, storage nodes and the interconnections between them. Design at this level requires knowledge of switch-level implementation details. VERILOG allows the designer to mix all the four levels of abstraction in a design. In the digital community the term register transfer level (RTL) is frequently used for a VERILOG description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools.

If a design contains four modules, VERILOG allows each of the modules to be written at a different level of abstraction. As the design matures, most modules are replaced with gate level implementations.

As a rule, higher the level of abstraction, more the flexibility and technology independent design. As one goes lower toward switch level design, the design becomes technology dependent and inflexible. A small modification can cause a significant number of changes in the design.

1.6 COMPONENTS OF SIMULATION:

Once a design block is completed, it must be tested. The functionality of the design a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block .We further sub-divide the sub-blocks until we come to leaf cells, which are cells that cannot further be divided. Applying stimulus and checking

results can test Block. We call this block the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block can be written in VERILOG. A separate language is not required to describe stimulus. The stimulus block is also commonly called a test bench. Different test benches can be used to thoroughly test the design block.

Two styles of stimulus applications are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block.

In the second style of applying stimulus, are to instantiate both the stimulus and design blocks in a top-level dummy module .The stimulus block interact with the design block only through an interface. The function of top-level block is simply to instantiate the design and stimulus blocks.

1.7 BASICS OF VERILOG LANGUAGE:

A VERILOG identifier including the names of variables may contain any sequence of letters, digits, a dollar sign '$' and the underscore '-' symbol. The first character of an identifier must be a letter or underscore; it cannot be a dollar sign '$' for example. We cannot use characters such as '-', brackets, or '#'(for active-low signals) in VERILOG names (escaped identifiers are an exception).

1.7.1 VERILOG LOGIC VALUES:

VERILOG has a predefined logic value system or value that

uses four logic values: '0', '1', 'x', 'z' (lower case 'x' and 'z'). The value 'x' represents an uninitialized or an unknown logic value-an unknown value is either '1', '0', 'z' or a value that is in state of change. The logic value 'z' represents a high impedance value, which is usually treated as 'x' value.

VERILOG uses a more complicated internal logic-value system in order to resolve conflicts between different drivers on the same mode. This hidden logic-value system is useful for switch-level simulation, but for most ASIC simulation.

1.7.2 VERILOG DATA TYPES:

There are several data types in VERILOG -all except one need to be stated before it can be used. The two main data types are nets and registers. Nets are further divided into several net types. The most common and important are wire and tri (which are identical); supply 1 and supply 0(which are equivalent to the positive and negative power supplies respectively). The wire data type is analogous to a wire in an ASIC.A wire cannot hold or store a value. An assignment statement must continuously drive a wire. The default initial value for a wire is 'z' there are also integer, time, event, and real data types.

A single bit wire or register is a scalar. It can also be declared a wire or register as vector with a range of bits. In some situations we may use implicit declaration for a scalar wire. It is the only data type that need not be declared. Explicit declaration must be used for a vector wire or any reg.

1.7.3 OTHER WIRE TYPES:

The following are the other types of wire in VERILOG:

- Wand, Wor, Triand and Trior model wired logic. Wiring or dotting the outputs of two gates generates a logic function (EEPROM for example).
- Tri0 and Tri1 model resistive connections to VSS or VDD
- Trireg is like a wire but associates some capacitance with the net, so it can model charge storage.

1.7.4 NUMBERS:

Constant numbers are integers or real constants. Integer constants are written as

width 'radix value'

where width and radix are optional. The radix (or base) indicates the type number:

decimal (d or D), hex (h or H), octal (o or O), binary (b or B).

A number may be sized or unsized. The length of an unsized number is implementation dependent. '1' or '0' numbers can be used as they cannot be identifiers, but we must write them as

1'bx and 1'bz for 'x' and 'z'

1.7.5 NEGATIVE NUMBERS:

Integer numbers are signed (two's complement) or unsigned. VERILOG only "keeps track " of the sign of a negative constant if it is

(1) Assigned to an integer

(2) Assigned to a parameter without using a base (essentially the same thing).

In other cases (the bit representation may be identified to the signed number), a negative constant is treated as an unsigned number. Once VERILOG "loses" the sign, keeping track of signed numbers becomes difficult.

1.7.6 STRINGS:

Strings can be stored in reg. The width of the register variables must be large enough to hold the string. Each character in the strings takes up 8-bits (1 byte). If the width of the register is greater than the size of the string, VERILOG fills bits to the left of the string with zeros. If the register width is smaller than the string width, VERILOG truncates the leftmost bits of the string. It is always safe to declare a string that is slightly wider than necessary.

1.7.7 OPERATORS:

An expression uses any of the three types of operators: Unary operators, binary operators and single ternary operators. The VERILOG operators are similar to those in the C programming language-except there is an auto increment (++) or auto decrement (- -) in VERILOG.

1.7.8 IDENTIFIER:

Every module instance, signal or variable is defined with an

Identifier. A particular identifier has a unique place in the design hierarchy. Hierarchical name referencing allows denoting every identifier in the designed hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots for each level of hierarchy. Thus any identifier can be addressed from any place in the design by simply specifying the complete hierarchical name of that identifier.

The top-level module is called root module because it is not instantiated anywhere. It is the starting point. To assign a unique name to an identifier, start from the top-level module and trace the trace the party along the designed hierarchy to the desired identifier. Hierarchical name referencing assigns a unique name to each identifier. To assign hierarchical name, use the module name for root module and instance names for all module instances below the root module. Each identifier in the design is uniquely specified by its hierarchal path name, to display the level of hierarchy use the special character %m in the $display task.

1.7.9 CONTINUOUS ASSIGNMENT:

The continuous assignment is used to drive a value on to a net. A continuous assignment replaces gates in the description of the circuits and describes the circuit at a higher level of abstraction. A continuous assignment statement starts with a keyword "assign". Continuous assignments have the following characteristics:

1. The left hand side of an assignment must always be a scalar or vector net or a concatenation of a scalar and vector and vector nets. It cannot be a scalar or vector register.
2. Continuous assignments are always active. The assignment expression is evaluated as soon as one of the right hand side operands change and the value is assigned to left hand side net.
3. The operands on the right hand side can be registers or nets or function calls. Registers or nets can be scalars or vectors.
4. Delay values can be specified for assignments in terms of time units. Delay values are used to control the time when a net is assigned the evaluated value. This feature is similar to specifying delays for gates. It is very useful in modeling timing behavior in real circuits.

1.7.10 IMPLICIT CONTINUOUS ASSIGNMENT:

Instead of declaring a net and then writing a continuous assignment on the net, VERILOG provides a shortcut by which the continuous assignment can be placed on a net when it is declared. There can be only one implicit declaration assignment per net because the net is declared only once.

1.7.11 DELAYS:

Delays values control the time between the changes in a right hand operand and when the new value is assigned to the left hand side. Three ways of

specifying delays in continuous assignment statements are regular assignment delay, implicit continuous assignment delay and net declaration delay.

1.7.12 REGULAR ASSIGNMENT DELAY:

First method is to assign a delay value in a continuous assignment statement. The delay value is specified after the keyword "ASSIGN". Any change in values in the expression will result in a delay of ten time units before recompilation of the expression. The expression changes value again before ten time units when the result propagates, the values at the time of recompilation are considered. This property is called inertial delay. An input pulse that is shorter than the delay of the assignment statement does not propagate to the output.

1.7.13 IMPLICIT CONTINUOUS ASSIGNMENT DELAY:

An equivalent method is to use an implicit continuous assignment to specify both a delay and an assignment on the net.

E.g.: assign #10 out = in1 & in2;

1.7.14 NET DECLARATION DELAY:

A delay can be specified on a net when it is declared without putting a continuous assignment on the net. If a delay is specified on a net, then any value change applied to the net is delayed accordingly.

1.8 EXAMPLES:

A design can be represented in terms of gates, data flow or a behavioral description. We consider the example of a 4-1Multiplexer. This can be designed from the logic diagram by direct translation into a VERILOG description. The same can be deigned in terms of data flow.

4-1 MULTIPLEXER:

We can use assignment statements instead of gates to model the logic equations of the multiplexer or can use conditional operators in the dataflow level.

GATE LEVEL MODELLING:

```
// module 4 to 1 mux
module mux4_to_1(out,i0,i1,i2,i3,s1,s2)
//port declarations
  output        out;
  input     i0,i1,i2,i3;
  input      s1,s0;
//internal wire declaration
  wire    s1n,s0n;
  wire   y0,y1,y2,y3;
//gate instantiation
  not(s1n,s1);
  not(s0n,s0);
//3 input and gates instantiation
```

```
  and (y0,i0,s1n,s0n);
  and (y1,i1,s1n,s0);
  and (y2,i2,s1,s0n);
  and (y3,i3,s1,s0);
or(out,y0,y1,y2,y3);
endmodule
//defn of stimulus module
  module stimulus;
//decl of variables to be connected
 reg in0,in1,in2,in3;
 reg s1,s0;
 wire output;
//instantiate the multiplexer
mux4_to_1mm(output,in0,in1,in2,in3,s1,s0);
//stimulate the inputs
initial begin
        //set inputs
         in0=1;in1=0;in2=1;in3=0;
        //choose in0
         s1=0;s0=0;
         #1 $display("s1=%b,s0=%b,output=%b\n",s1,s0,output);
        //choose in1
         s1=0;s0=1;
```

```
        #1 $display("s1=%b,s0=%b,output=%b\n",s1,s0,output);
        //choose in2
         s1=1;s0=0;
         #1 $display("s1=%b,s0=%b,output=%b\n",s1,s0,output);
         //choose in3
         s1=1;s0=1;
         #1 $display("s1=%b,s0=%b,output=%b\n",s1,s0,output);
end
endmodule
```

DATA FLOW MODELLING:

```
//module multiplexer 4 to 1
module mux(out,i0,i1,i2,i3,s1,s0);
//port declaration
output out;
input i0,i1,i2,i3;
input s1,s0;
assign out = (~s1 & ~s0 & i0);
             (~s1 & ~s0 &i1);
             (~s1 &~s0 &i2);
             (~s1 & ~s0 &i3);
endmodule
```

1.9 STRUCTURED PROCEDURES:

There are two structured procedure statements in VERILOG: always and initial. These statements are the two most basic statements in behavioral modeling. All other behavioral statements can appear only inside these structured procedure statements.

VERILOG is a concurrent programming language unlike C, which is sequential in nature. Each always and initial statement represents a separate activity flow in VERILOG. Each activity flows starts at simulation time 0. The statements always and initial cannot be nested.

1.9.1 INITIAL STATEMENT:

An initial block starts at time 0,executes exactly once during a simulation if there are multiple initial blocks, each block starts to execute concurrently at time 0. Each block finishes execution of tasks independently. Multiple statements can be grouped between Begin and End keywords. The following is an example of a program using initial statements:

```
module stimulus ;
reg x,z,c,v;
initial begin
        #5  x = 1'b1;
        #10 y = 1'b0;
end
initial
```

```
    # 50 $finish
endmodule
```

1.9.2 ALWAYS STATEMENT:

The ALWAYS statements starts at time zero and executes the statements in the always block enclosed with begin and end keywords. This statement is used to model a block of activity that is repeated continuously in a digital circuit.

```
module clock;
reg clock;
initial
    clock = 1'b0;
always
    # 10 clock = ~ clock;
initial
    #100 $finish;
endmodule
```

The above always block executes for every 10-time units and the clock is generated y toggling its value at every specific time unit.

1.9.3 PROCEDURAL ASSIGNMENTS:

These are used to update values; there are two types of assignments.

1.10 BLOCKING AND NON-BLOCKING STATEMENTS:

Blocking statements are executed in the order they are specified in a sequential block whereas a non-blocking statement allows scheduling of assignments without blocking execution of the statements that follow in a sequential block, a<= operator is used to specify non blocking assignments. The simulator schedules a non- blocking assignment statement to execute and continues to the next statement in the block without waiting for the non-blocking statement to complete execution.

1.11 LOOPS:

There are four types of looping statements, which includes: while, for repeat and forever. The while loop executes until the while loop becomes false. If the loop is entered when the while expression is false, the loop executed is not executed at all. If multiple statements are to be executed, they must be grouped typically using keywords begin and end.

The for loop contain three parts:

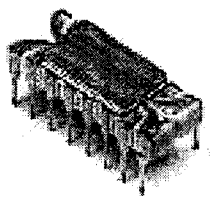
1. An initial condition
2. A check to see if the terminating condition is true.
3. A procedural assignment to change value of the control variable.

For loops are used for the verification part of the simulation.

The forever loop does not contain any expression and executes until the $finish task is encountered. Forever loops are used in conjunctions with timing control constructs. If timing control constructs are not used, the simulator would execute

this statement infinitely without advancing simulation time and rest of the design would never be executed.

```
//forever loop instead of always block
    reg clock;
    initial begin
            clock   = 1'b0;
            forever # 10 clock = ~clock;
    end
     endmodule
```



SERIAL EEPROM

CHAPTER-2

2. SERIAL EEPROM

2.1 FEATURES:

- **Low voltage and standard voltage operation**
  - **5.0 (V_{CC} = 4.5v to 5.5v)**
  - **2.7 (V_{CC} = 2.7v to 5.5v)**
  - **2.5 (V_{CC} = 2.5v to 5.5v)**
  - **1.8 (V_{CC} = 1.8v to 5.5v)**
- **Internally organized 128 * 8**
- **Two-wire serial interface, fully I2C-bus compatible**
- **Bi directional data transfer protocol**
- **100 KHz (1.8 v, 2.5 v, 2.7 v) and 400KHz (5v) compatibility**
- **8-byte page write mode**
- **Self-timed write cycle (10 ms max)**
- **High reliability**
  - **Endurance : 1 million write cycles**
  - **Data retention : 100 years**
  - **ESD protection : Greater than 3000v**
- **Automotive grade and extended temperature devices available**
- **8-pin PDIP, 8-pin MSOP, 8-pin TSSOP and 8-pin JEDEC SOI packages.**
- **Hardware WRITE control versions**
- **Byte and multi-byte WRITE (up to 4 bytes)**

- **Byte, random and sequential READ modes**
- **Automatic address incrementing**
- **Enhanced ESD/LATCH-UP performances**

2.2 DESCRIPTION:

This specification covers a range of 2K bits I2C EEPROM products. The EEPROM we have used for our design is ST24C02, where C stands for standard version. (Ref fig 2.1 in appendix logic diagram)

Signal names:

E0 – E2	**Chip enable inputs**
SDA	**Serial data address input/output**
SCL	**Serial clock**
MODE	**Multi-byte/page write mode (C version)**
V_{CC}	**Supply voltage**
V_{SS}	**Ground**

The ST24C02 is 2K bit electrically erasable programmable memories (EEPROM), organized as 256 * 8 bits. They are manufactured in SGS-THOMSON's Hi-endurance advanced CMOS technology which guarantees an endurance of one million erase/write cycles with a data retention of 10 years. The memories operate with a power supply value as low as 2.5V.

Both plastic dual-in-line and plastic small outline packages are available.

The memories are compatible with the I2C standard, two wire serial interface, which uses a bi-directional data bus and serial clock. The memories carry a built-in 4 bit, unique device identification code (1010) corresponding to the I2C-bus definition. This is used together with 3 chip enable inputs (E2, E1, E0) so that up to 8 * 2K devices may be attached to the I2C-bus and selected individually. The memories behave as a slave device in the I2C protocol with all memory operations synchronized by the serial clock. READ and WRITE operations are initiated by a START condition generated by the bus master. The START condition is followed by a stream of 7 bits (identification code 1010), plus one READ/WRITE bit and terminated by an acknowledge bit. When writing data to the memory it responds to the 8 bits received by asserting an acknowledge bit during the 9^{th} bit time. When the bus master reads data, it acknowledges the receipt of the data bytes in the same way. Data transfers are terminated with a STOP condition. The device is optimized for use in many industrial and commercial applications where low power and low voltage operation are essential. The ST24C02 is accessed via a 2-wire serial interface; in addition, the entire family is available in 5.0v (4.5V to 5.5V), 2.7 (2.7V to 5.5V), 2.5 (2.5V to 5.5V) and 1.8 (1.8V to 5.5V) versions.

2.3 ABSOLUTE MAXIMUM RATINGS:

- Ambient operating temperature :
  - -40°C to +85°C (grade 6)
  - 0°C to +70°C (grade 1)
- Storage temperature : -65°C to +150°C

- **Input and output voltages : -0.3 to 6.5V**
- **Supply voltage : -0.6 to 6.5V**

Notice: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions beyond those indicated in the operating sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

2.4 POWER ON RESET :

V_{CC} locks out write protect. In order to prevent data corruption and inadvertent write operations during power up, a Power ON Reset (POR) circuit is implemented. Until the V_{CC} voltage has reached the POR threshold value, the internal reset is active, all operations are disabled and the device will not respond to any command. In the same way, when V_{CC} drops down from the operating voltage to below the POR threshold value, all operations are disabled and the device will not respond to any command. A stable V_{CC} must be applied before applying any logic signal.

2.5 SIGNAL DISCRIPTION:

2.5.1 Serial Clock Line (SCL) :

The SCL input pin is used to synchronize all data in and out of the memory. A resistor can be connected from the SCL line to V_{CC} to act as a pull

2.5.2 Serial Data Line (SDA) :

The SDA pin is bi-directional and is used to transfer data or out of the memory. It is an open drain output that may be wire-OR'ed with other open drain or open collector signals on the bus. A resistor must be connected from the SDA bus line to V_{CC} to act as pull up.

2.5.3 Chip Enable (E2 – E0):

These chip enable inputs are used to set the 3 least significant bits (b3, b2, b1) of the 7-bit device select code. These inputs may be driven dynamically or tied to V_{CC} or V_{SS} to establish the device select code.

2.5.4 Mode (MODE):

The MODE input is available on pin 7 and may be driven dynamically. It must be at V_{IL} or V_{IH} for the byte write mode, V_{IH} for multi-byte write mode or V_{IL} for page write mode. When unconnected, the MODE input is internally read as a V_{IH} (multi-byte write mode).

2.6 DEVICE OPERATION:

2.6.1 I2C-BUS BACKGROUND:

The ST24C02 support the I2C-bus protocol. This protocol defines an device that sends data on to the bus as a transmitter and any device that reads the data as a receiver. The device that controls the data transfer is known as the master and the other as the slave. The master will always initiate a data transfer and will

provide the serial clock for synchronization. The ST24C02 are always slave devices in all communications.

2.6.2 START CONDITION :

START is defined by a high to low transition of the SDA with line while the clock SCL is stable in the HIGH state. A START condition must precede any command for data transfer. Except during a programming cycle, the ST24C02 continuously monitor the SDA and SCL signals for a START condition and will not respond unless one is given.

2.6.3 STOP CONDITION:

A LOW to HIGH transition of the SDA line defines STOP while the clock SCL is stable in the HIGH state. A STOP condition terminates communication between the ST24C02 and the master bus. A STOP condition at the end of a READ command forces the standby state. A STOP condition at the end of a WRITE command triggers the internal EEPROM WRITE cycle.

2.6.4 ACKNOWLEDGE BIT (ACK) :

An acknowledge signal is used to indicate a successful data transfer. The bus transmitter, either master or slave, will release the SDA after sending 8-bits of data. During the 9^{th} clock pulse period the receiver pulls the SDA bus low to acknowledge the receipt of the 8-bits of.

2.6.5 STANDBY MODE:

The ST24C02 features a low power standby mode, which is enabled:

(a) Upon power-up and

(b) After the receipt of the STOP bit and the completion of any internal operations.

2.6.6 DATA INPUT:

During data input the ST24C02 sample the SDA bus signal on the rising edge of the clock SCL. Note that for correct device operation the SDA signal must be stable during the clock LOW to HIGH transition and the data must change only when the SCL line is low.

2.6.7 MEMORY ADDRESSING:

To start communicating between the bus master and the slave ST24C02, the master must initiate a START condition. Following this, the master sends on to the SDA bus line 8-bits (MSB first) corresponding to the device select code (7-bits) and a READ or WRITE bit.

The 4 most significant bits of the device select code are the device type identifier, corresponding to the I2C bus definition. For these memories the 4-bits are fixed as 1010b. The following 3 bits identify the specific memory on the bus. They are matched to the chi enable signals E2, E1, E0. Thus up to 8 * 2K memories can be connected on the same bus giving a memory capacity total of 16K bits. After

a START condition any memory on the bus will identify the device code and compare the following 3 bits to its chip enable E2, E1, and E0.

The 8th bit sent is the READ or WRITE bit (RW'), this bit is set to '1' for READ and '0' for WRITE operations. If a match is found, the corresponding memory will acknowledge the identification on the SDA bus during the 9th bit time.

2.6.8 WRITE OPERATIONS:

The multi-byte WRITE mode (only available on the St24C02 versions) is selected when the MODE pin is at V_{IH} and the page write mode when the MODE pin is at V_{IL}. The MODE pin may be driven dynamically with the CMOS input levels.

Following a START condition the master sends a device select code with the RW' bit reset to '0'. The memory acknowledges this and waits for a byte address. The byte address of 8-bits provides access to 256 bytes of the memory. After receipt of the byte address the device again responds with acknowledge. For the ST24C02 versions, any WRITE command with WC' = 1 will not modify the memory content.

2.6.8.1 Byte Write:

In the byte WRITE mode the master sends one data byte, which is acknowledged by the memory. The master then terminates the transfer by generating a STOP condition. The WRITE mode is independent of the state of the MODE pin, which could be left floating if only this mode was to be used. However, it

is not a recommended operating mode, as pin has to be connected to either V_{IH} or V_{IL}, to minimize the stand-by current.

2.6.8.2 MULTI-BYTE WRITE:

For the multi-bye WRITE mode, the MODE pin must be at V_{IH}. The multi-byte WRITE mode can be started from any address in the memory. The master sends from one up to 4 bytes of data, which are each acknowledged by the memory. The master generating a STOP condition terminates the transfer. The duration of the WRITE cycle is 10 ms maximum except when bytes are accessed on rows (that is have different values for the 6 most significant address bits A7-A2), the programming time is then doubled to a maximum of 20ms. Writing more than 4 bytes in the multi-byte WRITE mode may modify data bytes in an adjacent row (one row is 8 bytes long). However the multi-byte WRITE can properly write up to 8 consecutive bytes only if the first address of these 8 bytes is the first address of the row, the following bytes written in the 7 following bytes of this same row.

2.6.8.3 Page Write:

For the page WRITE mode, the MODE pin must be at VIL. The page WRITE mode allows up to 8 bytes to be written in a single write cycle, provided that they are all located in the same row in the memory: that is the 5 most significant memory address bits (A7-A3) are the same. The master sends from one up to 8 bytes of data, which are each acknowledged by the memory. After each byte is transferred, the internal byte address counter (3 least significant bits only) is

incremented. The master generating a STOP condition terminates the transfer. Care must be taken to avoid address counter "roll-over" which could result in data being overwritten. Note that for any WRITE mode the generation by the master of the STOP condition starts the internal memory program cycle. All inputs are disabled until the completion of this cycle and the memory will not respond to any request.

2.6.8.4 MINIMIZING SYSTEM DELAYS BY POLLING ON ACK:

During the internal WRITE cycle the memory disconnects itself from the bus in order to copy the data from the internal latches to the memory cells. The sequence for polling is as follows:

Initial condition: a WRITE is in progress.

Step 1: The master issues a START condition followed by a device select byte.

Step 2: If the memory is busy with the internal WRITE cycle no ACK will be returned and the master goes back to step 1. If the memory has terminated the internal WRITE cycle it will respond with an ACK indicating that the memory is READY to receive the second part of the next instruction.

2.6.9 READ OPERATIONS:

Read operations are independent of the state of the mode pin. On delivery the memory content is set at all ones.

2.6.9.1 CURRENT ADDRESS READ:

The memory has an internal byte address counter. Each time a byte is read this counter is incremented. For the current address READ mode following a START condition the master sends a memory address with RW' bit set to '1'. The memory acknowledges this and outputs the byte addressed by the internal byte address counter. This counter is then incremented. The master does NOT acknowledge the byte output but terminates the transfer with a STOP condition.

2.6.9.2 RANDOM ADDRESS READ:

A dummy WRITE is performed to load the address into the address counter. This is followed by another START condition from the master and the byte address is repeated with RW' set to '1'. The memory acknowledges this and outputs the byte addressed. The master does NOT acknowledge the byte output but terminates the transfer with a STOP condition.

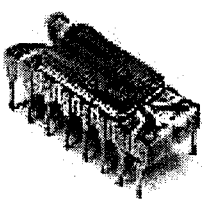
2.6.9.3 SEQUENTIAL READ:

This mode can be initiated with either a current address READ or a random address READ. However in this case the master DOES acknowledge the data byte output and the memory continues to output the next byte in sequence. To terminate the stream of bytes the master must NOT acknowledge the last byte output. But must generate a STOP condition. The output data is from consecutive byte addresses with the internal byte address counter automatically incremented

after each byte output. After a count of the last memory address the address counter will "roll-over" and the memory will continue to output data.

2.6.9.4 ACKNOWLEDGE IN READ MODE:

In all READ modes the ST24C02 wait for acknowledge during the 9^{th} bit time. If the master does not pull the SDA line low during this time the ST24C02 terminates the data transfer and switches to a stand-by state.



I2C-BUS

CHAPTER-3

3.THE INTER INTEGRATED CIRCUIT BUS

3.1VERSIONS 1.0 – 1992:

This version of I2C – bus specification includes the following modifications:

- **Programming of a slave address by software has been omitted.**
- **The " low speed mode " has been omitted.**
- **The fast mode is added. This allows a four-fold increase of the bit rate up to 400Kbit/s.**
- **10 – bit addressing is added. This allows 1024 additional slave addresses.**

3.2VERSION 2.0 – 1998 :

This updated version of the I2C bus specification meets the requirements of Higher bus speeds and lower supply and includes the following modifications:

- **The high – speed (Hs-mode) mode is added. This allows an increase in the bit rate up to 3.4Mbit/s. Hs-mode devices can be mixed with Fast and Standard-mode devices on the I2C bus system with bit rates from 0 to 3.4Mbit/s.**
- **The low output level and hysteresis of devices with a supply voltage of 2V and below has been adapted to meet the required noise margins and to remain compatible with the higher supply voltage devices.**

- **The 0.6V at 6mA requirement for the output stages of fast-mode devices has been omitted.**
- **Bus voltage-related levels replace the fixed input levels for new devices.**
- **Application information for bi-directional level shifter is added.**

3.3 VERSION 2.1 2000: (The version which we have designed)

Version 2.1 of the I2C-bus specification includes the following modifications:

- **After a repeated start condition in Hs-mode, it is possible to stretch the clock signal SCLH.**
- **Some timing parameters in Hs-mode have been relaxed.**

3.4 THE I2C-BUS BENEFITS:

In consumer electronics, telecommunications and industrial electronics, there are often many similarities between seemingly unrelated designs. For example, nearly every system includes:

- **Some intelligent control, usually a single-chip micro controller.**
- **General-purpose circuits like LCD drivers, remote I/O ports, RAM, EEPROM, or data converters.**
- **Application-oriented circuits such as digital tuning and signal processing circuits for radio and video systems, or DTMF generators for telephones with tone dialing.**

To exploit these similarities to the benefit of both systems designers

And equipment manufacturers, as well as o maximize hardware efficiency and circuit simplicity; Philips developed a simple 2-wire bus for efficient inter-IC control. This bus is called the Inter IC or I2C bus. At present, Philip's IC range includes more than 150 CMOS and bipolar I2C-bus compatible types for performing functions in all the three of the previously mentioned categories. All the I2C compatible devices incorporate an on-chip interface, which allows them to communicate directly with each other via the I2C-bus. This design concept solves the many interfacing problems encountered when designing digital control circuits.

3.4.1 FEATURES OF THE I2C-BUS :

- Only two bus lines are required:
  - A serial data line (SDA) and
  - A serial clock line (SCL)
- 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.
- It's a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- Serial, 8-bit oriented, bi-directional data transfers can be made at up to 100Kbit/s in the standard-mode, up to 400Kbit/s in the Fast-mode, or up to 3.4Mbit/s in the High-speed mode.

- **On-chip filtering rejects spikes on the bus data line to preserve data integrity.**
- **The number of IC's that can be connected to the bus is limited only by a maximum bus capacitance of 400pF. (Fig.1 shows 2 ex's of I2C)**

3.4.2 DESIGNER BENEFITS:

I2C compatible IC's allow a system design to rapidly progress, directly from a functional block diagram to a prototype. Moreover, since they 'clip' directly on to the I2C-bus without any additional external interfacing, they allow a prototype system to be modified or upgraded simply by 'clipping' or 'unclipping' IC's to or from the bus.

Here are some of the features of I2c-bus compatible IC's, which are particularly attractive to designers:

- **Functional blocks on the block diagram correspond with the actual IC's; designs proceed rapidly from block diagram to find schematic.**
- **No need to design bus interface because the I2C-bus is already integrated on-chip.**
- **Integrated addressing and data-transfer protocol allow systems to be completely software-defined.**
- **The same IC types can often be used in many different applications.**
- **Designers-time reduces as designers quickly become familiar with the frequently used functional blocks represented by I2C-bus compatible IC's.**

- **IC's can be added to or removed from a system without affecting any other circuits on the bus.**
- **Fault diagnosis and debugging are simple; malfunctions can be immediately traced,**
- **Software development time can be reduced by assembling a library of reusable software modules.**

In addition to these advantages, the CMOS IC's in the I2C-bus compatible range offer designers special features, which are particularly attractive for portable equipment and battery-backed systems. They all have:

- **Extremely low current consumption**
- **High noise immunity**
- **Wide supply voltage range**
- **Wide operating temperature range.**

3.4.3 MANUFACTURER BENEFITS:

I2C-bus compatible IC's don't only assist designers; they also give a wide range of benefits to equipment manufacture because:

- **The simple 2-wire serial I2c-bus minimizes interconnections so IC's have fewer pins and there are not so many PCB tracks; result-smaller and less expensive PCB's.**
- **The completely integrated I2C-bus protocol eliminates the need for address decoders and other 'glue-logic'.**

- **The multi-master capability of the I2C-bus allows rapid testing and alignment of end-user equipment via external connections to an assembly line.**
- **The availability of I2C-bus compatible IC's in SO (small outline), VSO (very small outline) as well as DIL packages reduces space requirements even more.**

These are just some of the benefits. In addition, I2C-bus compatible IC's increase system design flexibility by allowing simple construction of equipment variants and easy upgrading to keep designs up-to-date. In this way, an entire family of equipment can be developed around a basic model. Upgrades for new equipment, or enhanced-feature models (i.e. extended memory, remote control, etc.)Can then be produced simply by clipping the appropriate IC's on to the bus. If a larger ROM is needed, it's simply a matter of selecting a micro controller which supercede the older ones, it's easy to add new features to equipment or to increase its performance by simply unclipping the outdated IC from the bus and clipping on its successor.

3.5 INTRODUCTION TO THE I2C-BUS SPECIFICATION:

For 8-bit oriented digital control applications, such as those requiring micro controllers, certain design criteria can be established:

- **A complete system usually consists of at least one micro controller and other peripheral devices such as memories and I/O expanders.**

- **The cost of connecting the various devices within the system must be minimized.**
- **A system that performs a control function doesn't require high-speed data transfer.**
- **Overall efficiency depends on the devices chosen and the nature of the interconnecting bus structure.**

To produce a system to satisfy these criteria, a serial bus structure is needed. Although serial buses don't have the throughput capability of parallel buses, they do require less wiring and fewer IC connecting pins. However, a bus is not a merely an interconnecting wire, it embodies all the formats and procedures for communication within the system.

Devices communicating with each other on a serial bus must have some form of protocol, which avoids all possibilities of confusion, data loss and blockage of information. Fast devices must be able to communicate with slow devices. The system must not be dependent on the devices connected to it; otherwise modifications or improvements would be impossible. A procedure has also to be devised to decide which device will be in control of the bus and when. And, different devices with different clock speeds are connected to the bus; the bus clock source must be defined. All these criteria are involved in the specification of the I2C-bus.

3.6 THE I2C-BUS CONCEPT:

The I2C-bus supports any IC fabrication process (NMOS, CMOS, BIPOLAR). Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus. Each device is recognized by a unique

address (whether it's a micro controller, LCD driver, memory or keyboard interface) and can operate as either a transmitter or receiver, depending on the function of the device. Obviously an LCD driver is only a receiver, whereas a memory can both receive and transmit data. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers (Refer table 1). A master is the device which initiates a data transfer on the bus and generates the clock signals to permit the transfer. At that time, any device addressed is considered a slave.

TABLE1: DEFINITION OF I2C-BUS TERMINOLOGY:

1. TERM	DESCRIPTION
Transmitter	The device, which sends data to the bus.
Receiver	The device, which receives data from the bus.
Master	The device, which initiates a transfer, generates clock signals and terminates a transfer.
Slave	The device addressed by a master.
Multi-master	More than one master can attempt to control the bus at the same time without corrupting the message.
Arbitration	Procedure to ensure that, if more than one master simultaneously tries to control the bus, only one is allowed to do so and the winning message is not corrupted.
Synchronization	Procedure to synchronize the clock signals of two or more devices.

The I2C-bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. As masters are usually micro controllers, let's consider the case of a data transfer between two micro controllers connected to the I2C-bus. (See fig.2)

This highlights the master-slave and receiver-transmitter relationships to be found on the I2C-bus. It should be noted 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:

- **Suppose micro controller A wants to send information to micro controller B.**
  - **Micro controller A (master), addresses micro controller B (slave).**
  - **Micro controllers A (master-transmitter), send data to micro controller B (slave-receiver).**
  - **Micro controller A terminates the transfer.**
- **If micro controller A wants to receive information from micro controller B:**
  - **Micro controller A (master) addresses micro controller B (slave).**
  - **Micro controller A (master-receiver) receives data from micro controller B (slave-transmitter).**
  - **Micro controller A transmits the transfer.**

Even in this case, the master (micro controller A) generates the timing and terminates the transfer.

The possibility of connecting more than one micro controller to the I2C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid this chaos that might ensue from such an event-procedure arbitration procedure has been developed. This procedure relies on the wired-AND connection of all I2C interfaces to the I2C-bus.

If two or more masters try to put information on to the bus, the first to procedure a 'one' when the other produces a 'zero' will loose the arbitration. The clock signals during arbitration are a synchronized combination of the clocks generated by the masters using the wired-AND connection to the SCL line (for more detailed information concerning arbitration see section 8.)

Generation of clock signals on the I2C-bus is always the responsibility of master devices; each master generates its own clock signals when transferring data on the bus. Bus clock signals from a master can only be altered when they are stretched by a slow-slave device holding-down the clock line, or by another master, arbitration occurs.

3.7 GENERAL CHARACTERISTICS OF I2C-BUS:

Both SDA and SCL are bi-directional lines, connected to a positive supply voltage via a current-source or pull-up resistor (see fig.3). When the bus is free both lines are HIGH. The output stages of the devices connected to the bus must have an open-drain or open-collector to perform the wired-AND function. Data on the I2C-bus can be transferred at rates of up to 100Kbit/s in the standard-mode, up to 400Kbit/s in the Fast-mode, or up to 3.4Mbit/s in the High-speed mode. The

number of interfaces connected to the bus is solely dependent on the bus capacitance limit of 400pF.

3.8 BIT TRANSFER:

Due to the variety of different technology devices (CMOS, NMOS, BIPOLAR), which can be connected to the I2C-bus, the levels of the logical '0' (LOW) and '1' (HIGH) are not fixed and depend on the associated level of V_{DD}. One clock pulse is generated for each data bit transferred.

3.8.1 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 fig 4).

3.8.2 START AND STOP CONDITIONS:

Within the procedure of the I2C-bus, unique situations arise which are defined as START (S) and STOP (P) conditions (see fig.5).

A HIGH to LOW transition on the SDA line while SCL is HIGH is one such unique case. This situation indicates a START condition.

START and STOP conditions 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. This bus situation is specified in figure.

The bus stays busy if a repeated START (S_R) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (S_R) conditions are functionally same (see figure 10). For the remainder of this document, therefore, the S symbol will be used as a generic term to represent both START and repeated START conditions, unless S_R is particularly relevant.

Detection of START and STOP conditions by the devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, micro controllers with no such interface have to sample the SDA line at least twice per clock period to sense the transition.

3.9 TRANSFERRING DATA:

3.9.1 BYTE FORMAT:

Every byte put on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first (see figure 6). If a slave can't 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 SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another bye of data and releases clock line SCL.

In some cases, it's permitted to use a different format from the I2C-bus format (For CBUS compatible devices for example). A message, which starts with such an address, can be terminated by generation of a STOP condition, even

during the transmission of a byte. In this case, no acknowledge is generated (see section 10.1.3).

3.9.2 ACKNOWLEDGE:

Data transfers with acknowledge is obligatory. The master generates the acknowledge-related clock pulse. The transmitter releases the SDA line (HIGH) during the acknowledge clock pulse.

The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse (see figure 7). Of course, set-up and hold times must also be taken into account.

Usually, a receiver which has been addressed is obliged to generate acknowledge after each byte has been received, except when the message starts with a CBUS address.

When slave doesn't acknowledge the slave address (for example, it's unable to receive or transmit because it's performing some real-time function) , the data line must be left HIGH by the slave. The master can then generate either a STOP condition either to abort the transfer, or a repeated START condition to start a new transfer.

If a slave-receiver does acknowledge the slave address but some time later in the transfer cannot receive any more data bytes, the master must again abort the transfer. The slave generating the not-acknowledge indicates this on the

first byte to follow. The slave leaves the data line HIGH and the master generates a STOP or a repeated START condition.

If a master-receiver is involved in a transfer, it must signal the end of data to the slave-transmitter by not generating acknowledge on the last byte that was clocked out of the slave. The slave-master must release the data line to allow the master to generate a STOP or repeated START condition.

3.10 ARBITRATION AND SYNCHRONIZATION:

3.10.1 Synchronization:

All masters generate their own clock on the SCL line to transfer messages on the I2C-bus. Data is only valid during the high period of the clock. A defined clock is therefore needed for the bit-by-bit arbitration procedure to take place.

Clock synchronization is performed using wired AND connection of I2C interfaces on the SCL line. This means that a HIGH to LOW transition on the SCL line will cause the devices concerned to start counting off their LOW period and once a device clock has gone LOW, it will hold the SCL line in that state until the clock HIGH state is reached. However the LOW to HIGH transition of this clock may not change the state the of the SCL line if another clock is still within its LOW period. The SCL line will therefore be held LOW by the devices with the longest LOW period. Devices with shorter LOW periods enter a HIGH wait-state during this time (see fig.8).

When all devices concerned have counted off their LOW period, the clock line will be released and go HIGH. There will then be no difference between the device clocks and the state of the SCL line, and all the devices will start counting their HIGH periods. The first device to complete its HIGH period will again pull the SCL line LOW. In this way, a synchronized SCL clock is generated with its LOW period determined by the device with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period.

3.10.2 ARBITRATION:

A master may start a transfer only if the bus is free. Two or more masters may generate a START condition within the minimum hold time (t_{HD}; SDA) of the START condition, which results in a defined START condition to the bus (see fig 9).

Arbitration takes place on the SDA line, while the SCL line is at the HIGH level, in such a way that the master which transmits a HIGH level, while another master is transmitting a LOW level will switch off its DATA output stage because the level on the bus doesn't correspond to its own level. Arbitration can continue for many bits. Its first stage is comparison of the address bits. If the masters are each trying to address the same device, arbitration continues with comparison of the data-bits if they are master-transmitter, or acknowledge-bits if they are master-receiver. Because the winning master determines address and data information on I 2 C-bus, 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. As an Hs-mode master has a unique 8-bit master code, it will always finish the arbitration during the first byte. If a master also incorporates a slave function and it loses arbitration during the addressing stage, it's possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave mode. Figure 9 shows the arbitration procedure for two masters. Of course, 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 DATA 1 and the actual level on the SDA line, its data output is switched off, which means that a HIGH output level is then connected to the bus. This will not affect the data transfer initiated by the winning master (see fig.10).

Since control of I 2 C-bus is decided solely on the address or master code and data sent by competing masters, there is no central master, nor any order of priority on the bus. Special attention must be paid if, during a serial transfer, arbitration procedure is still in progress at the moment when a repeated START condition or a STOP condition is transmitted to I 2 C-bus. If it's possible for such a situation to occur, the masters involved must send this repeated START condition or STOP condition at the same position in the format frame. In other words, arbitration isn't allowed between:

- A repeated START condition and a data bit
- A STOP condition and a data bit
- A repeated START condition and a STOP condition.

Slaves are not involved in the arbitration procedure.

3.10.3 USE OF THE CLOCK SYNCHRONIZING MECHANISM AS A HANDSHAKE:

In addition to being used during the arbitration procedure, the clock synchronization mechanism can be used to enable receivers to cope with fast data transfers, on either a byte level or a bit level. On the byte level, a device may be able to receive bytes of data at a fast rate, but needs more time to store a received byte or prepare another byte to be transmitted. Slaves can then hold the SCL line LOW after reception and acknowledgment of a byte to force the master into a wait state until the slave is ready for the next byte transfer in a type of handshake procedure (see fig.6).

On the bit level, a device such as a micro controller with or without limited hardware for I 2C-bus, can slow down the bus clock by extending each clock LOW period. The speed of any master is thereby adapted to the internal operating rate of this device. In Hs-mode, this handshake feature can only be used on byte level.

3.11 FORMATS WITH 7-BIT ADDRESSES:

Data transfers follow the format shown in Fig.10. After the START condition (S), a slave address is sent. This address is 7 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 conditi

(S_R) and address another slave without first generating a STOP condition (see fig11). Various combinations of read/write formats are then possible within such a transfer.

Possible data transfer formats are:

- Master-transmitter transmits to slave-receiver. The transfer direction is not changed (see Fig.11).
- Master reads slave immediately after first byte (see Fig.12). At the moment of the first acknowledge, the master- transmitter becomes a master- receiver and the slave-receiver becomes a slave-transmitter. This first acknowledge is still generated by the slave. The STOP condition is generated by the master, which has previously sent a not-acknowledge (A).
- Combined format (see Fig.13). During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed. If a master receiver sends a repeated START condition, it has previously sent a not-acknowledge (A).

NOTES:

1. Combined formats can be used, for example, to control a serial memory. During the first data byte, the internal memory location has to be written. After the START condition and slave address is repeated data can be transferred.
2. All decisions on auto-increment or decrement of previously accessed memory locations etc; are taken by the designer of the device.
3. Each byte is followed by an acknowledgment bit as indicated by the A or A blocks in the sequence.

4. I 2 C-bus compatible devices must reset their bus logic on receipt of a START or repeated START condition such that they all anticipate the sending of a slave address, even if these START conditions are not positioned according to the proper format.

5. A START condition immediately followed by a STOP condition (void message) is an illegal format.

3.12 7-BIT ADDRESSING:

The addressing procedure for the I2C-bus is such that the first byte after the START condition usually determines which slave the master will select (see fig.13).The exception is the 'general call' address, which can address all devices. When this address is used, all devices should, in theory, respond with acknowledge. However, devices can be made to ignore this address. The second byte of the general call address then defines the action to be taken.

3.12.1 DEFINITION OF THE BITS IN THE FIRST BYTE:

The first seven bits of the first byte make up the slave address (see Fig.14). The eighth bit is the LSB (least significant bit). It determines the direction of the message. A 'zero' in the least significant position of the first byte means that the master will write information to a selected slave. A 'one' in this position means that the master will read information from the slave.

When an address is sent, each device in a system compares the first seven bits after the START condition with its address. If they match, the device considers itself addressed by the master as a slave-receiver or slave-transmitter,

depending on the R/W bit. A slave address can be made-up of a fixed and a programmable part. Since it's likely that there will be several identical devices in a system, the programmable part of the slave address enables the maximum possible number of such devices to be connected to the I2C-bus. The number of programmable address bits of a device depends on the number of pins available. For example, if a device has 4 fixed and 3 programmable address bits, a total of 8 identical devices can be connected to the same bus.

The I2C-bus committee coordinates allocation of I2C addresses. Further information can be obtained from the Philips representatives listed on the back cover. Two groups of eight addresses (0000XXX and 1111XXX) are reserved for the purposes shown in Table 2. The bit combination 11110XX of the slave address is reserved for 10-bit addressing (see Section 14).

Notes:

1. No device is allowed to acknowledge at the reception of the START byte.
2. The CBUS address has been reserved to enable the inter-mixing of CBUS compatible and I 2 C-bus compatible devices in the same system. I2C-bus compatible devices are not allowed to respond on reception of this address.
3. The address reserved for a different bus format is included to enable I 2 C and other protocols to be mixed. Only I 2 C-bus compatible devices that can work with such formats and protocols are allowed to respond to this address.

3.12.2 GENERAL CALL ADDRESS:

The general call address is for addressing every device connected to the I2C-bus. However, if a device doesn't need any of the data supplied within the general call structure, it can ignore this address by not issuing an acknowledgment. If a device does require data from a general call address, it will acknowledge this address and behave as a slave- receiver. Every slave- receiver will acknowledge the second and following bytes capable of handling this data. A slave, which cannot process one of these bytes must ignore it by not-acknowledging. The meaning of the general call address is always specified in the second byte (see Fig.15).

There are two cases to consider:

- When the least significant bit B is a 'zero'.
- When the least significant bit B is a 'one'.

When bit B is a 'zero'; the second byte has the following definition:

- 00000110 (H'06'). Reset and write programmable part of slave address by hardware. On receiving this 2-byte sequence, all devices designed to respond to the general call address will reset and take in the programmable part of their address. Pre-cautions have to be taken to ensure that a device is not pulling down the SDA or SCL line after applying the supply voltage, since these low levels would block the bus.
- 00000100 (H'04'). Write programmable part of slave address by hardware. All devices, which define the programmable part of their address by hardware (and which respond to the general call address) will latch this

programmable part at the reception of this two-byte sequence. The device will not reset.

- 00000000 (H'00'). This code is not allowed to be used as the second byte.

Sequences of programming procedure are published in the appropriate device data sheets. The remaining codes have not been fixed and devices must ignore them. When bit B is a 'one'; the 2-byte sequence is a 'hardware general call'. This means that a hardware master device, such as a keyboard scanner, which cannot be programmed to transmit a desired slave address, transmits the sequence. Since hardware master doesn't know in advance to which device the message has to be transferred, it can only generate this hardware general call and its own address - identifying itself to the system (see Fig.16). The seven bits remaining in the second byte contain the address of the hardware master. This address is recognized by an intelligent device (e.g. a micro controller) connected to the bus, which will then direct the information from the hardware master. If the hardware master can also act as a slave, the slave address is identical to the master address.

In some systems, an alternative could be that the hardware master transmitter is set in the slave-receiver mode after the system reset. In this way, a system configuring master can tell the hardware master- transmitter (which is now in slave-receiver mode) to which address data must be sent. After this programming procedure, the hardware master remains in the master-transmitter mode.

3.13 EXTENSIONS TO THE STANDARD-MODE I2C-BUS SPECIFICATION:

The Standard-mode I 2 C-bus specifications, with its data transfer rate of up to 100kbit/s and 7-bit addressing, has been in existence since the beginning of the 1980's. To meet the demands for higher speeds, as well as make available more slave address for the growing number of new devices, the Standard-mode I 2 C-bus specifications was upgraded over the years and today is available with the following extensions:

- **Fast-mode, with a bit rate up to 400kbit/s.**
- **High-speed mode (Hs-mode), with a bit rate up to 3.4Mbit/s.**
- **10-bit addressing, which allows the use of up to 1024 additional slave addresses.**

There are two main reasons for extending the regular I 2 C-bus specification:

- **Many of today's applications need to transfer large amounts of serial data and require bit rates far in excess of 100Kbit/s (Standard-mode), or even 400Kbit/s (Fast-mode). As a result of continuing improvements in semiconductor technologies, I 2 C-bus devices are now available with bit rates of up to 3.4Mbit/s (Hs-mode) without any noticeable increases in the manufacturing cost of the interface circuitry.**
- **As most of the 112 addresses available with the 7-bit addressing scheme were soon allocated, it became apparent that more address combinations were required to prevent problems with the allocation of slave addresses for new devices. This problem was resolved with the new 10-bit addressing scheme, which allowed about a tenfold increase in available addresses.**

New slave devices with a Fast- or Hs-mode I 2 C-bus interfaces can have a 7- or a 10-bit slave address. If possible, a 7-bit address is preferred as it is the cheapest hardware solution and results in the shortest message length. Devices with 7- and 10-bit addresses can be mixed in the same I 2 C-bus systems regardless of whether it is an F/S- or Hs-mode system. Both existing and future masters can generate either 7- or 10-bit addresses.

3.14 FAST-MODE:

With the Fast-mode I 2 C-bus specification, the protocol, format, logic levels and maximum capacitive load for the SDA and SCL lines quoted in the Standard-mode I 2 C-bus specification are unchanged. New devices with an I2C-bus interface must meet at least the minimum requirements of the Fast- or Hs-mode specification (see Section 13). Fast-mode devices can receive and transmit at up to 400kbit/s. The minimum requirement is that they can synchronize with a 400kbit/s transfer; they can then prolong the LOW period of the SCL signal to slow down the transfer. Fast-mode devices are downward compatible and can communicate with Standard-mode devices in a 0 to 100kbit/s I 2 C-bus systems. As Standard-mode devices, however, are not upward compatible, they should not be incorporated in a Fast-mode I 2 C-bus systems as they cannot follow the higher transfer rate and unpredictable states would occur.

The Fast-mode I 2 C-bus specifications has the following additional features compared with the Standard-mode:

- **The maximum bit rate is increased to 400kbit/s.**
- **Timing of the serial data (SDA) and serial clock (SCL) signals has been adapted. There is no need for compatibility with other bus systems such as CBUS because they cannot operate at the increased bit rate.**
- **The inputs of Fast-mode devices incorporate spike suppression and a Schmitt trigger at the SDA and SCL inputs.**
- **The output buffers of Fast-mode devices incorporate slope control of the falling edges of the SDA and SCL signals.**
- **If the power supply to a Fast-mode device is switched off, the SDA and SCL I/O pins must be floating so that they don't obstruct the bus lines.**
- **The external pull-up devices connected to the bus lines must be adapted to accommodate the shorter maximum permissible rise time for the Fast-mode I2C-bus. For busloads up to 200pF, the pull-up device for each bus line can be a resistor; for busloads between 200pF and 400pF, the pull-up device can be a current source (3mA max.) or a switched resistor circuit (see Fig.43).**

3.15 Hs-MODE:

High-speed mode (Hs-mode) devices offer a quantum leap in I 2 C-bus transfer speeds. Hs-mode devices can transfer information at bit rates of up to 3.4Mbit/s, yet they remain fully downward compatible with Fast- or Standard-mode (F/S-mode) devices for bi-directional communication in a mixed-speed bus system. With the exception that arbitration and clock synchronization is not performed during the Hs-mode transfer, the same serial bus protocol and data format is

maintained as with the F/S-mode system. Depending on the application, new devices may have a Fast or Hs-mode I 2 C-bus interface, although Hs-mode devices are preferred as they can be designed-in to a greater number of applications.

3.15.1 HIGH SPEED TRANSFER:

To achieve a bit transfer of up to 3.4Mbit/s the following improvements have been made to the regular I 2 C-bus specification:

- Hs-mode master devices have an open-drain output buffer for the SDAH signal and a combination of an open-drain pull-down and current-source pull-up circuit on the SCLH output (1). This current-source circuit shortens the rise time of the SCLH signal. Only the current-source of one master is enabled at any one time, and only during Hs-mode.
- No arbitration or clock synchronization is performed during Hs-mode transfer in multi-master systems, which speeds-up bit handling capabilities. The arbitration procedure always finishes after a preceding master code transmission in F/S-mode.
- Hs-mode master devices generate a serial clock signal with a HIGH to LOW ratio of 1 to 2. This relieves the timing requirements for set-up and hold times.
- As an option, Hs-mode master devices can have a built-in bridge (1). During Hs-mode transfer, the high speed data (SDAH) and high-speed serial clock (SCLH) lines of Hs-mode devices are separated by this bridge from the SDA

and SCL lines of F/S-mode devices. This reduces the capacitive load of the SDAH and SCLH lines resulting in faster rise and fall times.

- The only difference between Hs-mode slave devices and F/S-mode slave devices is the speed at which they operate. Hs-mode slaves have open-drain output buffers on the SCLH and SDAH outputs. Optional pull-down transistors on the SCLH pin can be used to stretch the LOW level of the SCLH signal, although this is only allowed after the acknowledge bit in Hs-mode transfers.
- The inputs of Hs-mode devices incorporate spike suppression and a Schmitt trigger at the SDAH and SCLH inputs.
- The output buffers of Hs-mode devices incorporate slope control of the falling edges of the SDAH and SCLH signals.

Figure 20 shows the physical I 2 C-bus configuration in a system with only Hs-mode devices. Pins SDA and SCL on the master devices are only used in mixed-speed bus systems and are not connected in an Hs-mode only system. In such cases, these pins can be used for other functions.

Optional series resistors R_S protect the I/O stages of the I2C-bus devices from high-voltage spikes on the bus lines and minimize ringing and interference.

Pull-up resistors R_P maintain the SDAH and SCLH lines at a HIGH level when the bus is free and ensure the signals are pulled up from a LOW to a HIGH level within the required rise time. For higher capacitive bus-line loads (>100pF), the resistor R_P can be replaced by external current source pull-ups to

meet the rise time requirements. Unless proceeded by an acknowledge bit, the rise time of the SCLH clock pulses in Hs-mode transfers is shortened by the internal current-source pull-up circuit MCS of the active master.

3.16 ELECTRICAL SPECIFICATIONS:

For devices operating in standard mode, the I\O levels, I\O current, spike suppression, output slope control etc. are given in the table 1.

Electrical connections of I2C Bus devices to the bus lines

I2C bus devices with fixed input levels of 1.5v and 3v can each have their own appropriate supply voltage. Pull-up resistors (R_P) must be connected to a 5v +/- 10% supply (see fig.36). I2C bus devices with input levels related to V_{DD} must have one common supply line to which the pull-up resistor is also connected. When devices with fixed input levels are mixed with devices with input levels related to V_{DD}, the latter devices must be connected to one common supply line of 5v+/- 10% and must have pull-up resistors connected to their SDA and SCL pins. A series resistor (R_S) is also used for protection against high voltage spikes.

Maximum and minimum values of R_S and R_P for standard – mode I2C bus devices

The values of pull-up resistors depend on the following parameters:

1. Supply voltage
2. Bus capacitance
3. Number of connected devices

The supply voltage limits the minimum value of resistor R_P due to the specified minimum sink current of 3 ma at V_{OL} max = .4v for the output. The

required noise margin limits the maximum value of R_S. The bus capacitance limits the maximum value of R_P due to the specified rise time. The maximum HIGH-level input current of each I\O connection has a specified maximum value of 10microA.

3.17 BIDIRECTIONAL LEVEL SHIFTER FOR S-MODE I2C BUS SYSTEMS:

To interface these lower voltage circuits with existing 5v devices, a level shifter is needed. Such a level shifter must also be bi-directional without the need of a direction control signal. Connecting a discrete MOS-FET to each bus line can do this. It also isolates a powered-down bus section from the rest of the bus and protects the lower voltage side against high voltage spikes from the higher voltage side (see fig.45).

3.17.1 CONNECTING DEVICES WITH DIFFERENT LOGIC LEVELS:

Different voltage devices could be connected to the same bus by using pull-up resistors to the supply voltage line. By using a bi-directional level shifter, it is possible to interconnect two sections of an I2C bus system, with each section having a different logic levels. The left low voltage section has pull-up resistors and devices connected to a 3.3v supply voltage, the right high voltage section has pull-up resistors and devices connected to a 5v supply voltage.

The devices of each section have I\Os with supply voltage related logic input levels an open drain output configuration.

The level shifter for each bus line is identical and consists of one discrete N-channel enhancement MOSFET .The gates have to be connected to the

lowest supply voltage Vdd1, the sources to the bus lines of the lower voltage section and the drains to the bus lines of the higher voltage section. Many MOSFET's have the substrate internally connected with its source, if it is not the case an external connection should be made. Each MOSFET has an integral diode (n-p junction) between the drains and substrate.

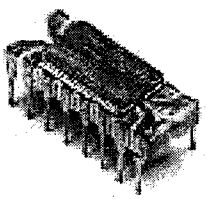
3.17.2 OPERATION OF THE LEVEL SHIFTER:

The following three states should be considered during the operation of the level shifter.

- No devices are pulling down the bus line. Its pull-up resistors Rp pull up the bus line of the lower voltage section to 3.3V. The gate and the source of the MOSFET are both at 3.3V. So its V_{GS} is below the threshold voltage and the MOSFET is not conducting. This allows the bus line at the higher voltage section to be pulled up by its pull-up resistor R_P to 5V. So the bus lines of both sections are HIGH but at different voltage levels.
- A 3.3 V device pulls down the bus line to a LOW level. The source of the MOS-FET also becomes LOW, while the gate stay at 3.3 V. VGS rises above the threshold and the MOS-FET starts to conduct. The bus line of the "higher-voltage" section is then also pulled down to a LOW level by the 3.3 V devices via the conducting MOS-FET. So the bus lines of both sections go LOW to the same voltage level.
- A 5 V device pulls down the bus line to a LOW level. The drain-substrate diode of the MOS-FET the "lower-voltage" section is pulled down until VGS

passes the threshold and the MOS-FET starts to conduct. The bus line of the "lower-voltage" section is then further pulled down to a LOW level by the 5V devices via the conducting MOS-FET. So the bus lines of both sections go LOW to the same voltage level.

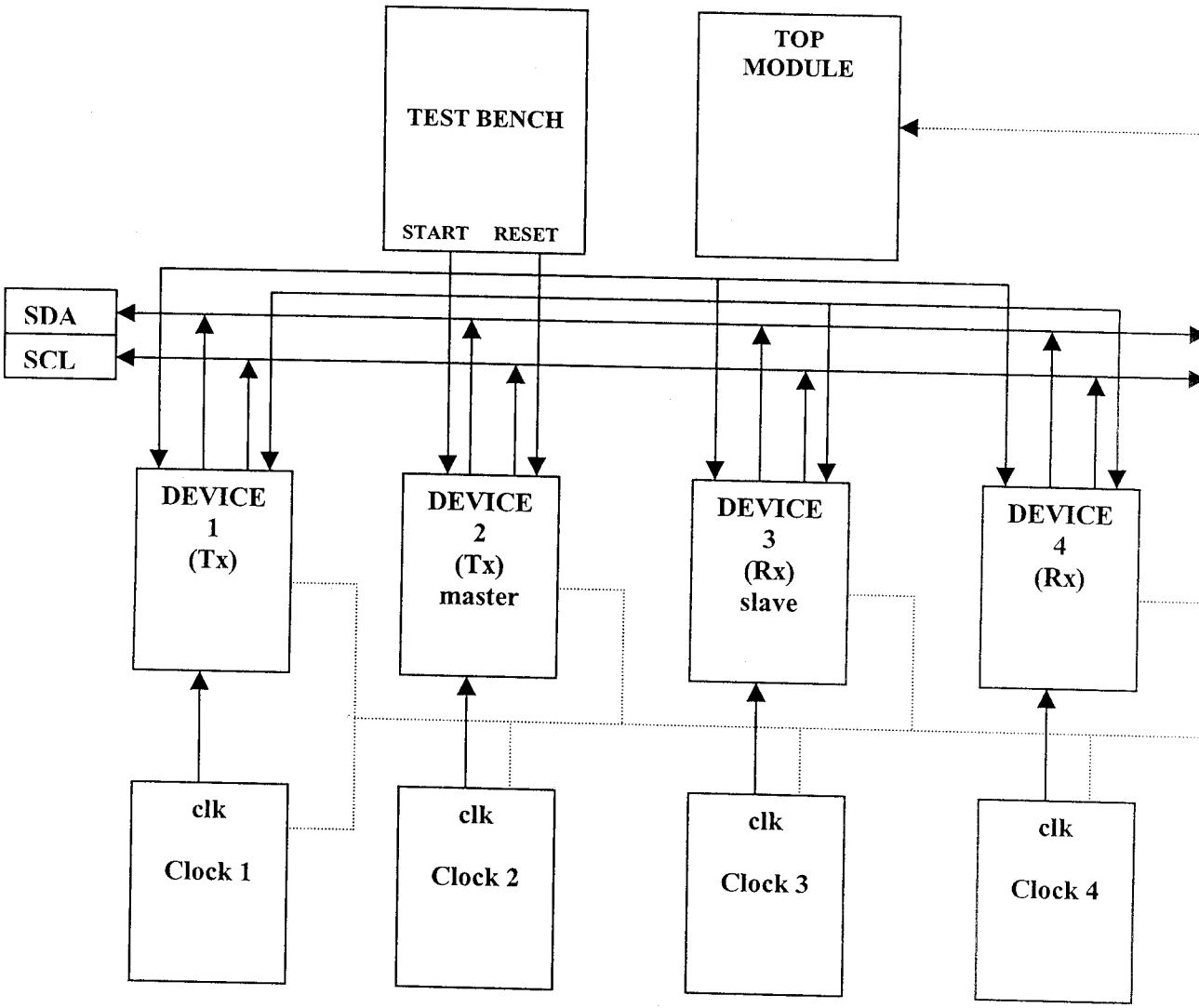
The three states show that the logic levels are transferred in both directions of the bus system, independent of the driving section. State 1 performs the level shift function. States 2 and 3 perform a "wired AND" function between the bus lines of both sections as required by I 2 C-bus specification. Supply voltages other than 3.3 V for V_{DD1} and 5 V for V_{DD2} can also be applied, e.g. 2 V for V_{DD1} and 10 V for V_{DD2} are feasible. In normal operation VDD2 must be equal to or higher than V_{DD1} (V_{DD2} is allowed to fall below V_{DD1} during switching power on/off).



ARCHITECTURE

CHAPTER-4

4. ARCHITECTURE:



"..... " ⇨ **Denotes instantiation within top module**

4.1 Clock Module :

Each device is provided with a separate clock module representing the internal clock of the device. The entire data transfer on the I2C bus is based on the clock of the master. A device that wins arbitration assigns its' clock to the SCL line initiating a data transfer in the Bus.

4.2 Master Transmitter:

The transmitter (master) initiates the transfer by issuing start condition. It provides the clock for data transfer through SCL. It sends the receiver 7-bit address together with write signal. The addressed device responds with acknowledge. Here the addressed device is the EEPROM. The transmitter sends the byte address where the data is to be written in the next byte after the first acknowledge. The receiver acknowledges. The transmitter then sends the data to be written to the EEPROM. After receiving data EEPROM acknowledges and the transmitter issues stop condition detecting acknowledge.

4.3 Slave Receiver :

The slave is EEPROM. It gives the acknowledgement when the master addresses this device by pulling SDA low at the 9^{th} clock pulse.

4.4 Master Receiver:

The Master receiver receives the data from the slave by setting the last bit of the first byte transmitted to one.

4.5 Slave Transmitter :

The slave reads the last bit of the first byte send by the transmitter. If it is a '1' the slave transmits its' data till the master does not acknowledge it. On

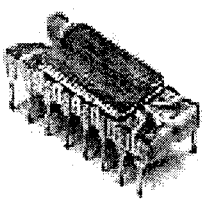
detecting the not acknowledge condition the slave transfers the control to Master to issue stop condition and the master issues the stop condition.

4.6 Test Bench:

The external signals for all the modules come from the test bench.

Top Module:

The top module instantiates all the modules and provides the required wire connections between the modules.



SIMULATION

CHAPTER-5

5. SIMULATION:

5.1. WHAT IS MODELSIM?

ModelSim simulates Verilog (and VHDL) source code. In contrast, the Xilinx and Altera simulators simulate from an EDIF file, which has advantages and disadvantages. One advantage: EDIF is more similar to the hardware. A strong disadvantage: the synthesizer removes or renames signals as it sees fit - it can be hard to recognize your own design. A big advantage of using ModelSim: it's very pedantic about suspicious coding style.

5.2 MODELSIM PROJECT:

A project is a collection entity for an HDL design under specification or test. At a minimum it has a root directory, a work library, and a session state that is stored in a .mpf file located in the project's root directory. With the exception of libraries, all components of a project reside in or below the project's root directory. A project can also have associated source files, libraries, and top-level design units.

5.3 MODELSIM LIBRARY:

A ModelSim Library is a directory created by ModelSim (using the vlib command or the Design > Create a New Library menu selection). This directory contains the

compiled objects from compiling VHDL or Verilog source files.

5.4 VSIM, VCOM, AND VLOG :

VSIM, VCOM and VLOG are tools used to compile and simulate Verilog and VHDL designs. VCOM compiles a VHDL source file into a library; VLOG compiles a Verilog source file into a library. Once a design is compiled, you invoke VSIM with the name of the top-level design unit (previously compiled). VSIM then loads the design components from their libraries and performs the simulation.

5.5 CREATING A PROJECT :

Create a project by selecting File > New > New Project from the Main window menu. The resulting dialog box allows you to create a project from scratch or by copying an existing project.

5.6 DESIGN COMPONENTS OF A PROJECT:

You must have a project open to work with it. To open a new project, select File > Open > New Project from the Main window menu (using the cd command at the DOS/UNIX prompt will not work). Once the project is open, you can create HDL source files by selecting File > New > New Source. When you create HDL files in the project's root directory, you are prompted to add them to the project. HDL files for a given project must reside at or below the project's root directory.

5.7 COMPILING THE PROJECT:

To compile your project's HDL source files with the project open, select Design > Compile from the Main window menu or the compile icon on the toolbar. Next, select all the files you want to compile. Each file will be compiled into your project's work library. You can also choose Design > Compile Project if you wish to recompile all of the previously compiled files in the current project.

5.8 SIMULATING THE PROJECT:

To simulate an open project, select Design > Load New Design from the Main window menu. On the Design tab of this dialog box, specify the top-level design unit for your project. On the VHDL and Verilog tabs, specify HDL specific simulator settings (these are described in the VSIM portion of the User's Reference Manual). On the SDF tab, you can specify settings relating to the annotation of design timing from an SDF file (optional).

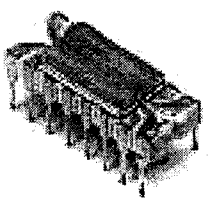
5.9 STOP WORKING ON A PROJECT:

Use the cd command to leave the project's working directory, or open another project by selecting File > Open > Open Project.

5.10 MODIFYING THE PROJECT SETTINGS:

There are four types of project settings: 1) Project wide settings-- these describe the makeup of the project; select Options > Edit Project. 2) Project

compiler settings--these specify HDL compiler options; select Options > Compile. 3) Project design simulation settings--these describe how a specific design unit is loaded and simulated; select Design > Load New Design. 4) Project simulation settings--these describe simulation specific behavior; select Options > Simulation.



SYNTHESIS

CHAPTER-6

6.SYNTHESIS:

6.1 INTRODUCTION TO LEONARDO SPECTRUM:

LeonardoSpectrum is a suite of high-level design tools for CPLD, FPGA, and ASIC synthesis design. LeonardoSpectrum offers design capture, VHDL and Verilog entry, register transfer level debug, constraint based optimization, timing analysis, encapsulated place and route, and schematic viewing.

A complete high-level design solution is here for Windows 95/98/NT, and for HP and Sun UNIX systems. LeonardoSpectrum provides three tool levels:

- LeonardoSpectrum Level 1
- LeonardoSpectrum Level 2
- LeonardoSpectrum Level 3

6.1.1 LeonardoSpectrum Level 1:

Level 1 is an easy-to-use, single FPGA technology, synthesis tool that uses the LeonardoSpectrum database. A logic designer need only select the input design and target technology and then click the Run button. A high-quality netlist is quickly produced.

6.1.2 LeonardoSpectrum Level 2:

Level 2 is an easy-to-use FPGA synthesis and timing analysis tool with back-annotation. A logic designer need only select the input design and target

technology and click the Run button. A high-quality netlist is quickly produced.

6.1.3.LeonardoSpectrum Level 3;

Level 3 is a versatile and interactive logic synthesis, optimization, and analysis tool developed to allow the use of technology-independent design methods for Field Programmable Gate Arrays (FPGAs), Complex Programmable Logic Devices (CPLDs) and ASICs. You can perform bottom-up design assembly with technology-mapped netlist. Hierarchy can be preserved, flattened, merged and dissolved or a block- by-block build can be done. Complex scripts can be written or captured and then used in command line and batch mode operations.

Level 3 utilizes the most powerful state-of-the-art optimization technology to guarantee high-quality results for any targeted FPGA technology.

6.6 SYNTHESIS WIZARD:

The SynthesisWizard is one of three ways to synthesize your design; Quick Setup and FlowTabs are the other two ways.

The SynthesisWizard consists of four steps that must be completed in the order presented. If you are a first-time user, then the SynthesisWizard is recommended to get you started right away. You can open by clicking on the Wizard hat icon, or click Flows->SynthesisWizard. Use the following defaults and accept displayed defaults to start synthesizing with LeonardoSpectrum:

- Technology: Altera FLEX 6k – Step 1
- Input Files: pseudorandom.vhd (demo) – Step 2

- **Global Constraints: 20 MHz – Step 3**
- **Output File and Finish: pseudorandom.edf (default) – Step 4**

6.3 RETARGETTING OUTPUT NETLIST:

Use these Quick Setup FlowTab steps to retarget a synthesized netlist from one technology to another technology. These steps assume you have already synthesized your design.

On Quick Setup FlowTab

1. **When input and output file windows are blank, click Open Files -> Set Input File(s) -> pseudonrandom.edf to open your output netlist in the Input file window.**

Note: On Set Input File(s), choose Files of type -> All files *.* to locate .edf files.

2. **Output file is: e:\exemplar\leospec\demo\pseudonrandom_1.edf**

Note: LeonardoSpectrum automatically adds "_1" to your retarget output netlist filename.

Input file is: pseudorandom.edf

3. **Click RMB on pseudonrandom.edf to open list:**
4. **Set Technology -> FPGA/CPLD -> Altera FLEX 6K. Click Altera FLEX 6K.**
5. **Click LMB on pseudonrandom.edf to bring file information into right window -> Source Technology: Altera FLEX 6K appears.**
6. **On Quick Setup Technologies: scroll to your target technology. For example, click on Xilinx 4000. Defaults change to Xilinx 4000.**

7. Click Run Flow. When run is complete, open schematic viewer and verify that cells are now Xilinx 4000.

6.4 PREPARATION OF AN OUTPUT NETLIST:

Use these Quick Setup FlowTab steps to synthesize a netlist for your technology.

1. On Quick Setup, run your design flow to generate an output netlist file with an .edf extension. For example, choose Altera FLEX 6K, 20 MHz, pseudorandom.vhd demo file.

Note: On Quick Setup:

Output File is: e:\exemplar\leospec\demo\pseudorandom.vhd

Input File is: pseudorandom.vhd

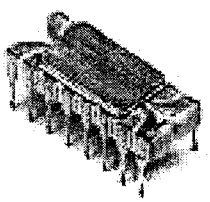
2. Click LMB on pseudorandom.vhd to bring file information into right window -> Source Technology: None appears.

3. Click Run Flow. When run is complete, open schematic viewer and verify that cells are Altera FLEX 6K.

4. Next, click RMB on pseudorandom.vhd input file to open list.

5. Click Remove to remove pseudorandom.vhd and, for example, e:\exemplar\leospec\demo\speudorandom.edf from Quick Setup.

6. You are now ready to retarget your output netlist to another technology. Continue to "How do I Retarget an Output Netlist?"



CODING

CHAPTER-7

7. CODING:

7.1 I2C WITH NOT ACKNOWLEDGE:

7.1.1 MASTER:

```
module i2c1(clk,start,sda,scl,reset);

 inout sda;
 inout scl;

 input start;
 input clk;
 input reset;

 reg      t1;                  // internal register to assign for sda
 reg [7:0] data;               // temp reg  to shift addr and data
 reg      data1;               // to assign last bit of address to check for read or write
 reg [3:0] cntr;               // counter to shift data
 reg      en;                  // enable bit to assign clk pulse for scl
 reg      en_bit;              // to make sda high after data transmission
 reg      flag_stop;           // to issue stop internally
 reg      flag_ack;            // to detect ack
 reg [1:0] cntr_stop;          // to issue stop
 reg      flag_rw;             // to detect rx or tx ,if==0 data is sent by tx else rx
 reg [3:0] cntr_ack;           // to give acknowledge
 reg      sda_ack;             // to pull sda low for acklg checking flag-rw
 reg      fg1_stop_low;        // to make stop zero if flag_start is at posedge clk
 reg [1:0] cntr_start;         // to detect start at posedge clk only
 reg      flag1_start;         // to activate internal start
 reg      flag2_start;         // to stop internal start
 reg      flag3_start;         // to stop internal start
 reg [7:0] data2;              // received data from rxr
 reg      sda_l2h;             // to pull sda low and then to make it high when SCL is high

 wire   stop;                  // to issue stop condition
 wire   flag_start;            // internal start

 assign flag_start = ((flag2_start == 1'b1 && flag3_start == 1'b0) || flag1_start ==
                                                                    1'b0) ? 1'b0 : 1'b1;

 assign sda      = (sda_l2h == 1'b0 && en_bit == 1'b0 && flag_start == 1'b1 &&
                                                        sda_ack == 1'b0 ) ? t1 : 1'bz ;
```

```
assign scl      = (en == 1'b1 && stop == 1'b0) ? clk : 1'b1;

assign stop     = (flag_stop == 1'b1 && fg1_stop_low == 1'b0) ? 1'b1 : 1'b0;

//START CONDITION:

always @ (negedge sda or posedge reset) begin

 if (reset == 1'b1)
   en = 1'b0;

 else begin

  if (scl == 1'b1 && stop == 1'b0)                    // used to assign clock
      en = 1'b1;

 end

end


always @ (negedge scl or posedge reset) begin        // to send data

 if (reset == 1'b1) begin
  t1      = 1'b0;
  cntr    = 4'b0000;
  data    = 8'b1010_1110;                            // addr of rxr with write active
low
  flag_rw  = 1'b0;
  data1    = data[0];
 end

 else begin

  if(flag_ack == 1'b1 && cntr_stop == 2'b11) begin   // similar to second reset
   data    = 8'b1010_1111;                           // addr of rxr with read = 1
   data1   = data[0];
  end

 if(stop == 1'b0) begin

  if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b01)
     data  = 8'b0111_0110;                           // byte address to write

  if (flag_ack == 1'b1 && flag_rw == 1'b1 && cntr_stop == 2'b01)
```

```
      data   = 8'b0111_0110;                          // byte address to read

    if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b10)
      data   = 8'b0111_0110;                          // first data sent

    if (cntr < 4'b1000) begin                         // to transmit data to sda
      t1   =  data[7];
      data =  data << 1'b1;
      cntr =  cntr + 4'b0001;
    end

    else if (cntr == 4'b1000) begin
      flag_rw =  data1;
      cntr    = 4'b0000;
      t1      = 1'b0;
    end
  end
 end
end

//data:

always @ (posedge scl or posedge reset) begin        // to receive data

 if (reset == 1'b1) begin
  cntr_stop = 2'b00;
  flag_ack  = 1'b0;
  cntr_ack  = 4'b0000;
  flag_stop = 1'b0;
  data2     = 8'b0000_0000;
 end

 else begin

   if (fg1_stop_low == 1'b1)
    flag_stop = 1'b0;

   if (flag_rw == 1'b0) begin

    if (cntr == 4'b0000 ) begin                       // to increment after each ack
     cntr_stop = cntr_stop + 2'b01;

      if (en_bit == 1'b1)
        flag_ack  = 1'b1;

       if (cntr_stop == 2'b11)
```

```
            flag_stop   = 1'b1;

        end

        else if (cntr_stop == 2'b11)
         cntr_stop      = 2'b00;
        end

      else if (flag_rw == 1'b1) begin

       if (cntr == 4'b0000) begin                          // to increment after each ack
        cntr_stop = cntr_stop + 2'b01;

         if (en_bit == 1'b1)
           flag_ack  = 1'b1;

          if (cntr_stop == 2'b11 && sda == 1'b1)
            flag_stop   = 1'b1;

         end

        else if (cntr_stop == 2'b11)
         cntr_stop      = 2'b00;
        end

       if (en_bit == 1'b0)
         flag_ack = 1'b0;

        if (flag_rw == 1'b1 && cntr_ack < 4'b1000 && cntr_stop == 2'b10 &&
                                                          flag_ack == 1'b0) begin
         data2    = data2 << 1'b1;
         data2[0] = sda;                              // to get the data from rxr
         cntr_ack = cntr_ack + 4'b0001;
        end

        else if (cntr_ack == 4'b1000)
         cntr_ack = cntr_ack + 4'b0001;

   end
end


always @ (posedge clk or posedge reset) begin
  if (reset == 1'b1) begin
    fg1_stop_low = 1'b0;
    cntr_start   = 2'b00;
```

```
    flag3_start  = 1'b0;
    flag1_start  = 1'b0;
    sda_l2h = 1'b0;
  end

  else begin
   if (flag_stop == 1'b1 && sda_ack == 1'b0 && flag_rw == 1'b1)
    sda_l2h = 1'b1;

   if (start == 1'b1)
     cntr_start = cntr_start + 2'b01;

   if (cntr_start == 2'b01) begin
     flag1_start = 1'b1;
     if (flag2_start == 1'b1)
         flag3_start = 1'b1;
   end

   if (flag2_start == 1'b0)
     flag3_start = 1'b0;

   if (start == 1'b0)
     cntr_start = 2'b00;

    if (flag2_start == 1'b1 && flag3_start == 1'b0)
      flag1_start = 1'b0;

    if (flag_stop == 1'b1) begin                    // to remove stop condition on start

     if (start == 1'b1 && cntr == 4'b0000)
       fg1_stop_low = 1'b1;
    end

    else if (fg1_stop_low == 1'b1 && flag_stop == 1'b0)
      fg1_stop_low = 1'b0;

  end
end

always @ (negedge clk or posedge reset) begin
  if (reset == 1'b1) begin
     en_bit  = 1'b0;
     sda_ack = 1'b0;

  end
```

```
  else begin

    if (cntr_stop == 2'b11 && flag_rw == 1'b1 && sda == 1'b1)
     sda_ack = 1'b0;                                  // to bring sda low for data ack

     if (sda_ack == 1'b0 && flag_rw == 1'b1 && (cntr_stop == 2'b10 || cntr_stop ==
                                                    2'b11) && cntr_ack < 4'b1000)
      sda_ack = 1'b1;

       if (cntr == 4'b1000) begin

      if (cntr_ack == 4'b0000)                      // to make sda high after addr txion
       en_bit = 1'b1;

    end

    if (flag_ack == 1'b1 && (cntr_stop == 2'b01 || cntr_stop == 2'b10))
          en_bit = 1'b0;

    if (flag1_start == 1'b0)                         // to give sda high imp after stop
     en_bit  = 1'b0;

  end
 end

 always @ (posedge sda or posedge reset) begin      // to issue stop condition

  if (reset == 1'b1)
    flag2_start = 1'b0;

  else begin

   if (scl == 1'b1)
    flag2_start  = 1'b1;

   else if (flag2_start ==1'b1)
    flag2_start = 1'b0;

  end

 end
endmodule
```

7.1.2 SLAVE:

```
module i2c1_rx(start,reset,clk,sda,scl);

 inout sda;
 inout scl;

 input clk;
 input start;
 input reset;

 reg [6:0] addr;            // to verify address
 reg [3:0] cntr_tb;         // to shift data and rx
 reg       rw;              // to receive lsb of addr
 reg [1:0] incr_tb;         // to distinguish data, addr and byte addr
 reg [7:0] tempdata_tb;     // to store the data
 reg [7:0] rx_data;         // data for rx
 reg       sda_det;         // temporary reg to shift to sda
 reg       fg_rw;           // detect rw in neg edge
 reg [3:0] bit_cntr;        // to make sda high after data txion
 reg       sda_high;        // to make sda high after data txion
 reg       fg_incr_tb;      // to make incr_tb zero after stop
 reg       fg1_start_tb;    // to initiate the start condition
 reg       fg2_start_tb;    // to terminate start
 reg [7:0] byte_addr;       // to assign byte addr
 reg       w1_en1;          // to make en1 high
 reg       w2_en1 ;         // to make en1 low

 reg [7:0] mem[255:0];  // to create memory for eeprom

 wire      fg_start_tb;     // instead of start
 wire      en1;             // to bring sda low for ack

 assign en1       = (w1_en1 == 1'b1 && w2_en1 == 1'b0) ? 1'b1 :1'b0;

 assign sda       = ((en1 == 1'b1 || fg_rw == 1'b1) && sda_high == 1'b0) ? sda_det :
                                                                          1'bz;

 assign fg_start_tb = (fg1_start_tb == 1'b1 && fg2_start_tb == 1'b0) ? 1'b1 : 1'b0;

 always @ (posedge scl or posedge reset) begin                  // to rx

  if (reset == 1'b1) begin
    cntr_tb    = 4'b0000;
    addr       = 7'b000_0000;
```

```
   rw         = 1'b0;
   tempdata_tb = 8'b0000_0000;
  end

  else begin

  if (fg_rw == 1'b0) begin                          // to detect  tx or rx

    if (fg_start_tb == 1'b1) begin
     if (incr_tb == 1'b0) begin                     // to detect addr or data

      if (cntr_tb < 4'b0111) begin                  // to receive addr
       addr    = addr << 1'b1;
       addr[0] = sda;
       cntr_tb = cntr_tb + 4'b0001;
      end

      else if (cntr_tb == 4'b0111 ) begin           // to detect tx or rx
       rw     = sda;
       cntr_tb = cntr_tb + 1'b1;
      end

      else if(cntr_tb == 4'b1000)
       cntr_tb = 4'b0000;
     end

     else begin                                     // to receive data

      if (cntr_tb < 4'b1000)  begin
        tempdata_tb    = tempdata_tb << 1'b1;
        tempdata_tb[0] = sda;
        cntr_tb        = cntr_tb + 4'b0001;
      end

      else if (cntr_tb == 4'b1000 )
       cntr_tb = 4'b0000;

     end
    end
   end
  end
end

always @(negedge scl or posedge reset) begin        // to give ack and send data

 if (reset == 1'b1) begin
```

```
    sda_det  = 1'b0;
    fg_rw    = 1'b0;
    bit_cntr = 4'b0000;
    sda_high = 1'b0;
    byte_addr = 8'b0000_0000;
    rx_data  = 8'b0000_0000;
  end

  else begin



  if (incr_tb == 2'b01 && cntr_tb == 4'b1000)
     byte_addr = tempdata_tb;

  if (incr_tb == 2'b10 && cntr_tb == 4'b1000)  begin
     mem [byte_addr] = tempdata_tb;
     rx_data       =  mem [byte_addr];
  end

   if (rw == 1'b1 && en1 == 1'b0 && incr_tb == 2'b10) begin          // to tx data
     sda_det  = rx_data[7];
     rx_data  = rx_data << 1'b1;
     bit_cntr = bit_cntr + 4'b0001;

      if (bit_cntr == 4'b1001) begin
        bit_cntr = 4'b0000;
        sda_high = 1'b1;
      end

      if (cntr_tb == 4'b0000 && rw == 1'b1 && incr_tb == 2'b10)
        fg_rw  = 1'b1;
    end
  end
 end

always @ (posedge sda or posedge reset) begin                    // to issue stop

  if (reset == 1'b1) begin
     fg2_start_tb = 1'b0;
     fg_incr_tb   = 1'b0;
  end

  else begin

   if(scl == 1'b1 && reset == 1'b0) begin
```

```
    fg_incr_tb  = 1'b1;
    fg2_start_tb = 1'b1;
  end

  else if(fg2_start_tb == 1'b1)
    fg2_start_tb = 1'b0;

  else if(fg_incr_tb == 1'b1)
    fg_incr_tb = 1'b0;

  end
end

always @ (posedge clk or posedge reset) begin          // to make en1 zero after stop

if (reset == 1'b1) begin
   w1_en1    = 1'b0;
   incr_tb   = 2'b00;
end

else begin

 if (cntr_tb == 4'b1000 && (incr_tb == 2'b01 || incr_tb == 2'b10)) begin
    incr_tb = incr_tb + 2'b01;
    w1_en1    = 1'b1;
 end

 else if (w1_en1 == 1'b1)
   w1_en1 = 1'b0;

  else if (cntr_tb == 4'b1000 && incr_tb == 2'b00) begin

    if(addr == 7'b1010_111) begin                      // to give ack for addr
      w1_en1    = 1'b1;
      incr_tb = 2'b01;
    end
 end

   if(fg_incr_tb == 1'b1)
     incr_tb = 2'b00;

  end
end

always @ (negedge clk or posedge reset) begin
```

```
if (reset == 1'b1) begin
   w2_en1     = 1'b0;
 end
 else begin

 if (w1_en1 == 1'b1)
       w2_en1 = 1'b1;
 else if (w2_en1 == 1'b1)
    w2_en1 = 1'b0;
  end

 end

 always @ (posedge start or posedge reset) begin                    // internal start

  if (reset == 1'b1)

    fg1_start_tb = 1'b0;

  else
   fg1_start_tb = 1'b1;

  end

  endmodule
```

7.2 I2C WITH ACKNOWLEDGE:

7.2.1 MASTER:

```
module i2c1(clk,start,sda,scl,reset);

  inout sda;
  inout scl;

  input start;
  input clk;
  input reset;

  reg      t1;                  // internal register to assign for sda
  reg [7:0] data;               // temp reg  to shift addr and data
  reg      data1;               // to assign last bit of address to check for read or write
                                // counter to shift data
```

```
reg      en;                  // enable bit to assign clk pulse for scl
reg      en_bit;              // to make sda high after data transmission
reg      flag_stop;           // to issue stop internally
reg      flag_ack;            // to detect ack
reg [1:0] cntr_stop;          // to issue stop
reg      flag_rw;             // to detect rx or tx ,if==0 data is sent by tx else rx
reg [3:0] cntr_ack;           // to give acknowledge
reg      sda_ack;             // to pull sda low for acklg checking flag-rw
reg      fg1_stop_low;        // to make stop zero if flag_start is at posedge clk
reg [1:0] cntr_start;         // to detect start at posedge clk only
reg      flag1_start;         // to activate internal start
reg      flag2_start;         // to stop internal start
reg      flag3_start;         // to stop internal start
reg [7:0] data2;              // received data from rxr

wire  stop;                   // to issue stop condition
wire  flag_start;             // internal start

assign flag_start = ((flag2_start == 1'b1 && flag3_start == 1'b0) || flag1_start ==
                                                                    1'b0) ? 1'b0 : 1'b1;

assign sda      = (en_bit == 1'b0 && flag_start == 1'b1 && sda_ack == 1'b0) ? t1 :
                                                                    1'bz ;

assign scl      = (en == 1'b1 && stop == 1'b0) ? clk : 1'b1;

assign stop     = (flag_stop == 1'b1 && fg1_stop_low == 1'b0) ? 1'b1 : 1'b0;

//START CONDITION:

always @ (negedge sda or posedge reset) begin

  if (reset == 1'b1)
    en = 1'b0;

  else begin

    if (scl == 1'b1 && stop == 1'b0)                    // used to assign clock
        en = 1'b1;

   end

  end
```

```
always @ (negedge scl or posedge reset) begin                    // to send data

  if (reset == 1'b1) begin
    t1       = 1'b0;
    cntr     = 4'b0000;
    data     = 8'b1010_1110;                    // addr of rxr with write active low
    flag_rw  = 1'b0;
    data1    = data[0];
  end

  else begin

    if(flag_ack == 1'b1 && cntr_stop == 2'b11) begin
      data     = 8'b1010_1111;                        // addr of rxr with read = 1
      data1    = data[0];
end

  if(stop == 1'b0) begin

    if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b01)
       data  = 8'b0111_0110;                          // byte address to write

     if (flag_ack == 1'b1 && flag_rw == 1'b1 && cntr_stop == 2'b01)
       data  = 8'b0111_0110;                          // byte address to read

     if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b10)
       data  = 8'b0111_0110;                          // first data sent

     if (cntr < 4'b1000) begin                        // to transmit data to sda
       t1   =  data[7];
       data =  data << 1'b1;
       cntr =  cntr + 4'b0001;
     end

     else if (cntr == 4'b1000) begin
       flag_rw =  data1;
       cntr    = 4'b0000;
       t1      = 1'b0;
     end
   end
  end
end

 //data:

 always @ (posedge scl or posedge reset) begin                   // to receive data
```

```
if (reset == 1'b1) begin
 cntr_stop = 2'b00;
 flag_ack  = 1'b0;
 cntr_ack  = 4'b0000;
 flag_stop = 1'b0;
 data2     = 8'b0000_0000;
end

 else begin

   if (fg1_stop_low == 1'b1)                          // to end stop depending on either clk
    flag_stop = 1'b0;

      if (cntr == 4'b0000 && sda == 1'b0) begin      // to increment after each ack
      cntr_stop = cntr_stop + 2'b01;

       if (en_bit == 1'b1)
         flag_ack  = 1'b1;

        if (cntr_stop == 2'b11 && flag_rw == 1'b0)
           flag_stop   = 1'b1;

       end

      else if (cntr_stop == 2'b11)
       cntr_stop     = 2'b00;



      if (en_bit == 1'b0)
        flag_ack = 1'b0;

       if (flag_rw == 1'b1 && cntr_ack < 4'b1000 && cntr_stop == 2'b10 &&
                                                          flag_ack == 1'b0) begin

        data2    = data2 << 1'b1;
        data2[0] = sda;                                  // to get the data from rxr
        cntr_ack = cntr_ack + 4'b0001;
       end

       else if (cntr_ack == 4'b1000)
        cntr_ack = cntr_ack + 4'b0001;

   end
  end
```

```
if (reset == 1'b1) begin
 en_bit  = 1'b0;

end

else begin

    if (sda_ack == 1'b0 && flag_rw == 1'b1 && (cntr_stop == 2'b10 || cntr_stop
                                                == 2'b11) && cntr_ack != 4'b1000)

    sda_ack = 1'b1;

   if (cntr == 4'b1000) begin

    if (cntr_ack == 4'b0000)                    // to make sda high after addr txion
      en_bit = 1'b1;

   end

    if (flag_ack == 1'b1 && (cntr_stop == 2'b01 || cntr_stop == 2'b10))
      en_bit = 1'b0;

    if (flag1_start == 1'b0)                    // to give sda high imp after stop
      en_bit  = 1'b0;

  end
end

always @ (posedge sda or posedge reset) begin


  if (reset == 1'b1)
    flag2_start = 1'b0;

   else begin

    if (scl == 1'b1 && stop == 1'b1)
      flag2_start  = 1'b1;

    else if (flag2_start ==1'b1)
      flag2_start = 1'b0;

   end

  end
endmodule
```

7.2.2 SLAVE:

```
module i2c1_rx(start,reset,clk,sda,scl);

 inout sda;
 inout scl;

 input clk;
 input start;
 input reset;

 reg [6:0] addr;              // to verify address
 reg [3:0] cntr_tb;           // to shift data and rx
 reg       en1;               // to bring sda low for acknowledge
 reg       rw;                // to receive lsb of addr
 reg [1:0] incr_tb;           // to distinguish data, addr and byte addr
 reg [7:0] tempdata_tb;       // to store the data
 reg [7:0] rx_data;           // data for rx
 reg       sda_det;           // temporary reg to shift to sda
 reg       fg_rw;             // detect rw in neg edge
 reg [3:0] bit_cntr;          // to make sda high after data txion
 reg       sda_high;          // to make sda high after data txion
 reg       fg_incr_tb;        // to make incr_tb zero after stop
 reg       fg1_start_tb;      // to initiate the start condition
 reg       fg2_start_tb;      // to terminate start
 reg [7:0] byte_addr;         // to assign byte addr
 reg [7:0] mem[255:0];        // to create memory for eeprom

 wire      fg_start_tb;       // instead of start

 assign sda       = ((en1 == 1'b1 || fg_rw == 1'b1) && sda_high == 1'b0) ? sda_det :
                                                                          1'bz;

 assign fg_start_tb = (fg1_start_tb == 1'b1 && fg2_start_tb == 1'b0) ? 1'b1 : 1'b0;

 always @ (posedge scl or posedge reset) begin                  // to rx

  if (reset == 1'b1) begin
    cntr_tb    = 4'b0000;
    addr       = 7'b000_0000;
    rw         = 1'b0;
    tempdata_tb = 8'b0000_0000;
  end

  else begin
```

```
      if(scl == 1'b1 && reset == 1'b0) begin
        fg_incr_tb  = 1'b1;
        fg2_start_tb = 1'b1;
      end

      else if(fg2_start_tb == 1'b1)
        fg2_start_tb = 1'b0;

      else if(fg_incr_tb == 1'b1)
        fg_incr_tb = 1'b0;

      end
    end

    always @ (posedge clk or posedge reset) begin          // to make en1 zero after stop

    if (reset == 1'b1) begin
      en1     = 1'b0;
      incr_tb = 2'b00;
    end

   else begin

    if (cntr_tb == 4'b1000 && (incr_tb == 2'b01 || incr_tb == 2'b10)) begin
      incr_tb = incr_tb + 2'b01;
      en1     = 1'b1;
    end



    else if (cntr_tb == 4'b1000 && incr_tb == 2'b00) begin

     if(addr == 7'b1010_111) begin                                  // to give ack for addr
       en1     = 1'b1;
       incr_tb = 2'b01;
     end
   end

    if(fg_incr_tb == 1'b1)
      incr_tb = 2'b00;

   end
  end
always @ (negedge clk or posedge reset) begin
if (en1 == 1'b1)
```

```
end

always @ (posedge start or posedge reset) begin                    // internal start
if (reset == 1'b1)

  fg1_start_tb = 1'b0;

else
 fg1_start_tb = 1'b1;

end

endmodule
```

7.3 ARBITRATION FOR DEVICES HAVING SAME CLOCK RATE:

7.3.1 DEVICE 1:

```
module i2c1(clk,start,sda,scl,reset);

inout sda;
inout scl;

input start;
input clk;
input reset;

reg      t1;                          // internal register to assign for sda
reg [7:0] data;                       // temp reg  to shift addr and data
reg      data1;
reg [3:0] cntr;                       // counter to shift data
reg      en;                          // enable bit to assign clk pulse for scl
reg      en_bit;                      // to make sda high after data transmission
reg      flag_stop;                   // to issue stop internally
reg      flag_ack;                    // to detect ack
reg [1:0] cntr_stop;                  // to issue stop
reg      flag_rw;                     // to detect rx or tx
reg [3:0] cntr_ack;                   // to give acknowledge
reg      sda_ack;
reg      fg1_stop_low;
reg [1:0] cntr_start;                 // to detect start
```

```
 reg      flag1_start;                    // to activate internal start
 reg      flag2_start;                    // to stop internal start
 reg      flag3_start;                    // to stop internal start
 reg [7:0] data2;                         // received data from rxr
 reg      arbit1;                         // to arbitrate with same clock
 reg      sda_l2h;
 reg      set;

 wire   stop;
 wire   flag_start;                       // to issue stop condition
                                          // internal start

assign flag_start = ((flag2_start == 1'b1 && flag3_start == 1'b0) || flag1_start ==
                                                    1'b0) ?   1'b0 : 1'b1;

 assign sda      = (en_bit == 1'b0 && flag_start == 1'b1 && sda_ack == 1'b0 &&
                                  set ==  1'b1 && arbit1 == 1'b0) ? t1 : 1'bz ;

 assign scl      = (en == 1'b1 && stop == 1'b0 && set == 1'b1) ? clk : 1'b1;

 assign stop     = (flag_stop == 1'b1 && fg1_stop_low == 1'b0) ? 1'b1 : 1'b0;

 //START CONDITION:

 always @ (negedge sda or posedge reset) begin

  if (reset == 1'b1)
    en = 1'b0;

  else begin

   if (scl == 1'b1 && stop == 1'b0)                    // used to assign clock
       en = 1'b1;

  end

end


always @ (negedge scl or posedge reset) begin        // to send data

 if (reset == 1'b1) begin
  t1       = 1'b0;
  cntr     = 4'b0000;
  data     = 8'b1100_1110;
  flag_rw  = 1'b0;                                   // addr of rxr with write active low
```

```
    data1     = data[0];
   end

   else begin

    if(flag_ack == 1'b1 && cntr_stop == 2'b11) begin      // similar to second reset
     data     = 8'b1100_1111;                             // addr of rxr with read = 1
     data1    = data[0];
    end

   if(stop == 1'b0) begin

    if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b01)
       data  = 8'b0111_0110;                              // byte address to write

     if (flag_ack == 1'b1 && flag_rw == 1'b1 && cntr_stop == 2'b01)
       data  = 8'b0111_0110;                              // byte address to read

     if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b10)
       data  = 8'b0111_0111;                              // first data sent

     if (cntr < 4'b1000) begin                            // to transmit data to sda
       t1  =  data[7];
       data =  data << 1'b1;
       cntr =  cntr + 4'b0001;
     end

     else if (cntr == 4'b1000) begin     // to make flag_rw high after byte addr
       flag_rw =  data1;
       cntr   = 4'b0000;
       t1     = 1'b0;
     end
   end
  end
end

//data:

always @ (posedge scl or posedge reset) begin                    // to receive data

  if (reset == 1'b1) begin
   cntr_stop = 2'b00;
   flag_ack  = 1'b0;
   cntr_ack  = 4'b0000;
   flag_stop = 1'b0;
```

```
 arbit1    = 1'b0;
end

else begin

 if (t1 == 1'b1 && sda === 1'bx)
   arbit1 = 1'b1;

  if (fg1_stop_low == 1'b1)                    // to end stop depending on either clk
    flag_stop = 1'b0;

  if (flag_rw == 1'b0) begin

  if (cntr == 4'b0000) begin                   // to increment after each ack
    cntr_stop = cntr_stop + 2'b01;

    if (en_bit == 1'b1)
      flag_ack  = 1'b1;

    if (cntr_stop == 2'b11)
      flag_stop   = 1'b1;

   end


   else if (cntr_stop == 2'b11)               // to prevent further incrementation
     cntr_stop     = 2'b00;
  end

 else if (flag_rw == 1'b1) begin

    if (cntr == 4'b0000) begin                    // to increment after each ack
      cntr_stop = cntr_stop + 2'b01;

       if (en_bit == 1'b1)
         flag_ack  = 1'b1;

        if (cntr_stop == 2'b11 && sda == 1'b1)
          flag_stop   = 1'b1;

       end

       else if (cntr_stop == 2'b11)
        cntr_stop     = 2'b00;
       end
```

```
      if (en_bit == 1'b0)
        flag_ack = 1'b0;

      if (flag_rw == 1'b1 && cntr_ack < 4'b1000 && cntr_stop == 2'b10 &&
                                                        flag_ack  == 1'b0) begin
        data2   = data2 << 1'b1;
        data2[0] = sda;                          // to get the data from rxr
        cntr_ack = cntr_ack + 4'b0001;
      end

      else if (cntr_ack == 4'b1000)
        cntr_ack = cntr_ack + 4'b0001;

  end
end


always @ (posedge clk or posedge reset) begin

 if (reset == 1'b1) begin
    fg1_stop_low = 1'b0;
    cntr_start  = 2'b00;
    flag3_start  = 1'b0;
    flag1_start  = 1'b0;
    sda_l2h     = 1'b0;
  end
else begin

 if (flag_stop == 1'b1 && sda_ack == 1'b0 && flag_rw == 1'b1)
   sda_l2h = 1'b1;

 if (start == 1'b1)
    cntr_start = cntr_start + 2'b01;

 if (cntr_start == 2'b01) begin
    flag1_start = 1'b1;
    if (flag2_start == 1'b1)
        flag3_start = 1'b1;
 end

 if (flag2_start == 1'b0)
    flag3_start = 1'b0;

 if (start == 1'b0)                      // only if start is one cntr_start will be one
    cntr_start = 2'b00;
```

```
      if (flag2_start == 1'b1 && flag3_start == 1'b0)
        flag1_start = 1'b0;

      if (flag_stop == 1'b1) begin                    // to remove stop condition on start
       if (start == 1'b1 && cntr == 4'b0000)
         fg1_stop_low = 1'b1;
      end

      else if (fg1_stop_low == 1'b1 && flag_stop == 1'b0)
        fg1_stop_low = 1'b0;

   end
end

always @ (negedge clk or posedge reset) begin
 if (reset == 1'b1) begin
  en_bit  = 1'b0;
  sda_ack = 1'b0;
  set     = 1'b0;
end

else begin

  if(sda == 1'b1 && flag1_start == 1'b1) //arbitration process for different clocks
    set = 1'b1;

  if (cntr_stop == 2'b11 && flag_rw == 1'b1 && sda == 1'b1)
   sda_ack = 1'b0;                                    // to bring sda low for data ack
  if (sda_ack == 1'b0 && flag_rw == 1'b1 && (cntr_stop == 2'b10 || cntr_stop ==
   sda_ack = 1'b1;                                    2'b11) && cntr_ack != 4'b1001)

 if (cntr == 4'b1000) begin

  if (cntr_ack == 4'b0000)
   en_bit = 1'b1;                                   // to make sda high after addr txion

 end

if (flag_ack == 1'b1 && (cntr_stop == 2'b01 || cntr_stop == 2'b10))
 en_bit = 1'b0;

if (flag1_start == 1'b0)
 en_bit  = 1'b0;                                    // to give sda high i
```

```
    end
  end

  always @ (posedge sda or posedge reset) begin
   if (reset == 1'b1)
     flag2_start = 1'b0;

   else begin

    if (scl == 1'b1)
     flag2_start  = 1'b1;

    else if (flag2_start ==1'b1)
     flag2_start = 1'b0;

  end

 end
endmodule
```

7.3.2 DEVICE 2:

```
module i2c2(clk,start,sda,scl,reset);

 inout sda;
 inout scl;

 input start;
 input clk;
 input reset;

 reg      t1;                    // internal register to assign for sda
 reg [7:0] data;                  // temp reg  to shift addr and data
 reg      data1;
 reg [3:0] cntr;                  // counter to shift data
 reg      en;                     // enable bit to assign clk pulse for scl
 reg      en_bit;                 // to make sda high after data transmission
 reg      flag_stop;              // to issue stop internally
 reg      flag_ack;               // to detect ack
 reg [1:0] cntr_stop;             // to issue stop
 reg      flag_rw;                // to detect rx or tx
 reg [3:0] cntr_ack;              // to give acknowledge
 reg      sda_ack;
```

```
    reg     fg1_stop_low;
    reg [1:0] cntr_start;
    reg     flag1_start;                    // to detect start at posedge clk only
    reg     flag2_start;                    // to activate internal start
    reg     flag3_start;                    // to stop internal start
    reg [7:0] data2;                        // to stop internal start
    reg     arbit1;                         // received data from rxr
    reg     sda_l2h;                        // to arbitrate with same clock
    reg     set;

    wire  stop;
    wire  flag_start;                       // to issue stop condition
                                            // internal start

    assign flag_start = ((flag2_start == 1'b1 && flag3_start == 1'b0) || flag1_start == 1'b0)
  ?
                                                                    1'b0 : 1'b1;

    assign sda      = (en_bit == 1'b0 && flag_start == 1'b1 && sda_ack == 1'b0 &&
                                    set ==  1'b1 && arbit1 == 1'b0) ? t1 : 1'bz ;
    assign scl      = (en == 1'b1 && stop == 1'b0 && set == 1'b1) ? clk : 1'b1;
    assign stop     = (flag_stop == 1'b1 && fg1_stop_low == 1'b0) ? 1'b1 : 1'b0;

  //START CONDITION:

  always @ (negedge sda or posedge reset) begin

   if (reset == 1'b1)
     en = 1'b0;

   else begin

    if (scl == 1'b1 && stop == 1'b0)
        en = 1'b1;                          // used to assign clock

  end

end


always @ (negedge scl or posedge reset) begin       // to send data

 if (reset == 1'b1) begin
   t1       = 1'b0;
```

```
        cntr      = 4'b0000;
        data      = 8'b1011_1110;                    // addr of rxr with write active low
        flag_rw   = 1'b0;
        data1     = data[0];
       end

       else begin

        if(flag_ack == 1'b1 && cntr_stop == 2'b11) begin      // similar to second reset
         data     = 8'b1011_1111;                             // addr of rxr with read = 1
         data1    = data[0];
        end

      if(stop == 1'b0) begin

       if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b01)
          data   = 8'b0111_0110;                             // byte address to write

       if (flag_ack == 1'b1 && flag_rw == 1'b1 && cntr_stop == 2'b01)
         data   = 8'b0111_0110;                              // byte address to read

       if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b10)
         data   = 8'b0111_0111;                              // first data sent

       if (cntr < 4'b1000) begin                             // to transmit data to sda
         t1   =  data[7];
         data =  data << 1'b1;
         cntr =  cntr + 4'b0001;
       end

      else if (cntr == 4'b1000) begin     // to make flag_rw high after byte addr
        flag_rw =  data1;
        cntr    = 4'b0000;
        t1      = 1'b0;
      end
    end
  end
end

//data:

always @ (posedge scl or posedge reset) begin                // to receive data

 if (reset == 1'b1) begin
  cntr_stop = 2'b00;
  flag_ack  = 1'b0;
```

```
    cntr_ack  = 4'b0000;
    flag_stop = 1'b0;
    data2     = 8'b0000_0000;
    arbit1    = 1'b0;
  end

  else begin

  if (t1 == 1'b1 && sda === 1'bx)
    arbit1 = 1'b1;

   if (fg1_stop_low == 1'b1)                    // to end stop depending on either clk
    flag_stop = 1'b0;

   if (flag_rw == 1'b0) begin

   if (cntr == 4'b0000) begin                   // to increment after each ack
    cntr_stop = cntr_stop + 2'b01;

    if (en_bit == 1'b1)
      flag_ack  = 1'b1;

    if (cntr_stop == 2'b11)
      flag_stop   = 1'b1;

  end


  else if (cntr_stop == 2'b11)                  // to prevent further incrementation
    cntr_stop     = 2'b00;
end

else if (flag_rw == 1'b1) begin

   if (cntr == 4'b0000) begin                   // to increment after each ack
    cntr_stop = cntr_stop + 2'b01;

     if (en_bit == 1'b1)
       flag_ack  = 1'b1;

     if (cntr_stop == 2'b11 && sda == 1'b1)
       flag_stop   = 1'b1;

    end

   else if (cntr_stop == 2'b11)
```

```
        cntr_stop     = 2'b00;
       end



      if (en_bit == 1'b0)
        flag_ack = 1'b0;

      if (flag_rw == 1'b1 && cntr_ack < 4'b1000 && cntr_stop == 2'b10 &&
                                                          flag_ack  == 1'b0) begin
       data2    = data2 << 1'b1;
       data2[0] = sda;                              // to get the data from rxr
       cntr_ack = cntr_ack + 4'b0001;
      end

      else if (cntr_ack == 4'b1000)
       cntr_ack = cntr_ack + 4'b0001;

  end
end


always @ (posedge clk or posedge reset) begin

 if (reset == 1'b1) begin
   fg1_stop_low = 1'b0;
   cntr_start  = 2'b00;
   flag3_start  = 1'b0;
   flag1_start  = 1'b0;
   sda_l2h     = 1'b0;
  end
else begin

 if (flag_stop == 1'b1 && sda_ack == 1'b0 && flag_rw == 1'b1)
  sda_l2h = 1'b1;

 if (start == 1'b1)
  cntr_start = cntr_start + 2'b01;

 if (cntr_start == 2'b01) begin
   flag1_start = 1'b1;
   if (flag2_start == 1'b1)
      flag3_start = 1'b1;
end

if (flag2_start == 1'b0)
```

```
        flag3_start = 1'b0;

      if (start == 1'b0)
        cntr_start = 2'b00;                    // only if start is one cntr_start will be one

      if (flag2_start == 1'b1 && flag3_start == 1'b0)
        flag1_start = 1'b0;

      if (flag_stop == 1'b1) begin                    // to remove stop condition on start

       if (start == 1'b1 && cntr == 4'b0000)
         fg1_stop_low = 1'b1;
      end

     else if (fg1_stop_low == 1'b1 && flag_stop == 1'b0)
       fg1_stop_low = 1'b0;

  end
end

always @ (negedge clk or posedge reset) begin
 if (reset == 1'b1) begin
  en_bit  = 1'b0;
  sda_ack = 1'b0;
  set     = 1'b0;
end

else begin

  if(sda == 1'b1 && flag1_start == 1'b1) //arbitration process for different clocks
    set = 1'b1;

 if (cntr_stop == 2'b11 && flag_rw == 1'b1 && sda == 1'b1)
  sda_ack = 1'b0;                                  // to bring sda low for data ack
 if (sda_ack == 1'b0 && flag_rw == 1'b1 && (cntr_stop == 2'b10 || cntr_stop ==
 sda_ack = 1'b1;                                   2'b11) && cntr_ack != 4'b1001)

if (cntr == 4'b1000) begin

 if (cntr_ack == 4'b0000)
  en_bit = 1'b1;                          // to make sda high after addr txion

end
```

```
        if (flag_ack == 1'b1 && (cntr_stop == 2'b01 || cntr_stop == 2'b10))
         en_bit = 1'b0;

        if (flag1_start == 1'b0)
         en_bit  = 1'b0;                                  // to give sda high imp after stop

      end
     end

     always @ (posedge sda or posedge reset) begin          // to issue stop condition

      if (reset == 1'b1)
        flag2_start = 1'b0;

      else begin

       if (scl == 1'b1)
        flag2_start  = 1'b1;

       else if (flag2_start ==1'b1)
        flag2_start = 1'b0;

     end

   end
  endmodule
```

7.3.3 EEPROM:

```
module i2c1_rx(start,reset,clk3,sda,scl);

 inout sda;
 inout scl;

 input clk3;
 input start;
 input reset;

 reg [6:0] addr;              // to verify address
 reg [3:0] cntr_tb;           // to shift data and rx
 reg       rw;                // to receive lsb of addr
 reg [1:0] incr_tb;           // to distinguish data, addr and byte addr
 reg [7:0] tempdata_tb;       // to store the data
 reg [7:0] rx_data;           // data for rx
 reg       sda_det;           // temporary reg to shift to sda
 reg       fg_rw;             // detect rw in neg edge
```

```
 reg [3:0] bit_cntr;          // to make sda high after data txion
 reg       sda_high;          // to make sda high after data txion
 reg       fg_incr_tb;        // to make incr_tb zero after stop
 reg       fg1_start_tb;      // to initiate the start condition
 reg       fg2_start_tb;      // to terminate start
 reg [7:0] byte_addr;         // to assign byte addr
 reg       w1_en1;            // to make en1 high
 reg       w2_en1 ;           // to make en1 low


 reg [7:0] mem[255:0];        // to create memory for eeprom

 wire      fg_start_tb;       // instead of start
 wire      en1;               // to bring sda low for ack

 assign en1       = (w1_en1 == 1'b1 && w2_en1 == 1'b0) ? 1'b1 :1'b0;

 assign sda       = ((en1 == 1'b1 || fg_rw == 1'b1) && sda_high == 1'b0) ? sda_det :
1'bz;

 assign fg_start_tb = (fg1_start_tb == 1'b1 && fg2_start_tb == 1'b0) ? 1'b1 : 1'b0;

 always @ (posedge scl or posedge reset) begin                    // to rx

 if (reset == 1'b1) begin
   cntr_tb    = 4'b0000;
   addr       = 7'b000_0000;
   rw         = 1'b0;
   tempdata_tb = 8'b0000_0000;
 end

 else begin

 if (fg_rw == 1'b0) begin                                         // to detect  tx or rx

   if (fg_start_tb == 1'b1) begin                                 // only if start = 1 it receives data

     if (incr_tb == 1'b0) begin                                   // to detect addr or data

       if (cntr_tb < 4'b0111) begin                               // to receive addr
         addr    = addr << 1'b1;
         addr[0] = sda;
         cntr_tb = cntr_tb + 4'b0001;
       end
```

```
        else if (cntr_tb == 4'b0111 ) begin              // to detect tx or rx
          rw     = sda;
          cntr_tb = cntr_tb + 1'b1;
        end

        else if(cntr_tb == 4'b1000)
          cntr_tb = 4'b0000;
      end

      else begin                                         // to receive data

        if (cntr_tb < 4'b1000)  begin
          tempdata_tb    = tempdata_tb << 1'b1;
          tempdata_tb[0] = sda;
          cntr_tb        = cntr_tb + 4'b0001;
        end

        else if (cntr_tb == 4'b1000 )
          cntr_tb = 4'b0000;

      end
    end
  end
 end
end

always @(negedge scl or posedge reset) begin              // to give ack and send data

 if (reset == 1'b1) begin
   sda_det  = 1'b0;
   fg_rw    = 1'b0;
   bit_cntr = 4'b0000;
   sda_high = 1'b0;
   byte_addr = 8'b0000_0000;
   rx_data  = 8'b0000_0000;
end

else begin

if (incr_tb == 2'b01 && cntr_tb == 4'b1000)
   byte_addr = tempdata_tb;

if (incr_tb == 2'b10 && cntr_tb == 4'b1000)  begin
   mem [byte_addr] = tempdata_tb;
   rx_data         =  mem [byte_addr];
 end
```

```
    if (rw == 1'b1 && en1 == 1'b0 && incr_tb == 2'b10) begin
      sda_det  = rx_data[7];
      rx_data  = rx_data << 1'b1;
      bit_cntr = bit_cntr + 4'b0001;

       if (bit_cntr == 4'b1001) begin                    // to leave sda high after data txion
         bit_cntr = 4'b0000;
         sda_high = 1'b1;
       end

       if (cntr_tb == 4'b0000 && rw == 1'b1 && incr_tb == 2'b10)
         fg_rw  = 1'b1;
     end
    end
   end

  always @ (posedge sda or posedge reset) begin                    // to issue stop

   if (reset == 1'b1) begin
      fg2_start_tb = 1'b0;
      fg_incr_tb   = 1'b0;
   end

   else begin

    if(scl == 1'b1 && reset == 1'b0) begin
      fg_incr_tb  = 1'b1;
      fg2_start_tb = 1'b1;
    end

   else if(fg2_start_tb == 1'b1)
     fg2_start_tb = 1'b0;

   else if(fg_incr_tb == 1'b1)
     fg_incr_tb = 1'b0;

   end
  end

  always @ (posedge clk3 or posedge reset) begin          // to make en1 zero after stop

 if (reset == 1'b1) begin
   w1_en1    = 1'b0;
   incr_tb = 2'b00;
 end
```

```
      else begin

       if (cntr_tb == 4'b1000 && (incr_tb == 2'b01 || incr_tb == 2'b10)) begin
         incr_tb = incr_tb + 2'b01;
         w1_en1    = 1'b1;
       end

       else if (w1_en1 == 1'b1)
        w1_en1 = 1'b0;

       else if (cntr_tb == 4'b1000 && incr_tb == 2'b00) begin

        if(addr == 7'b1011_111) begin
          w1_en1    = 1'b1;                          // to give ack for addr
          incr_tb = 2'b01;
        end
      end

       if(fg_incr_tb == 1'b1)
        incr_tb = 2'b00;

     end
    end
  always @ (negedge clk3 or posedge reset) begin

  if (reset == 1'b1) begin
     w2_en1    = 1'b0;
   end
   else begin

  if (w1_en1 == 1'b1)
       w2_en1 = 1'b1;
  else if (w2_en1 == 1'b1)
    w2_en1 = 1'b0;
  end

end


always @ (posedge start or posedge reset) begin                    // internal start
if (reset == 1'b1)

 fg1_start_tb = 1'b0;
```

```
    else
     fg1_start_tb = 1'b1;

   end

   endmodule
```

7.4 ARBITRATION WITH DEVICES HAVING DIFFERENT CLOCKS:

7.4.1. DEVICE1:

```
module i2c1(clk,start,sda,scl,reset);

 inout sda;
 inout scl;

 input start;
 input clk;
 input reset;

 reg      t1;
 reg [7:0] data;                    // internal register to assign for sda
 reg      data1;                    // temp reg  to shift addr and data
 reg [3:0] cntr;
 reg      en;                       // counter to shift data
 reg      en_bit;                   // enable bit to assign clk pulse for scl
 reg      flag_stop;
 reg      flag_ack;                 // to issue stop internally
 reg [1:0] cntr_stop;               // to detect ack
 reg      flag_rw;                  // to issue stop
 reg [3:0] cntr_ack;
 reg      sda_ack;                  // to give acknowledge
 reg      fg1_stop_low;
 reg [1:0] cntr_start;
 reg      flag1_start;              // to detect start at posedge clk only
 reg      flag2_start;              // to activate internal start
 reg      flag3_start;              // to stop internal start
 reg [7:0] data2;                   // to stop internal start
 reg      set;                      // received data from rxr
 reg      sda_l2h;

wire  stop;
wire  flag_start;                   // to issue stop condition
                                    // internal start
```

```
assign flag_start = ((flag2_start == 1'b1 && flag3_start == 1'b0) || flag1_start ==
                                                                    1'b0) ? 1'b0 : 1'b1;
assign sda      = (sda_l2h == 1'b0 && en_bit == 1'b0 && flag_start == 1'b1 &&
                                    sda_ack == 1'b0 && set == 1'b1) ? t1 : 1'bz ;
assign scl      = (en == 1'b1 && stop == 1'b0 && set == 1'b1) ? clk : 1'b1;
assign stop     = (flag_stop == 1'b1 && fg1_stop_low == 1'b0) ? 1'b1 : 1'b0;

//START CONDITION:

always @ (negedge sda or posedge reset) begin

 if (reset == 1'b1)
   en = 1'b0;

 else begin

  if (scl == 1'b1 && stop == 1'b0)
     en = 1'b1;                                          // used to assign clock

 end

end


always @ (negedge scl or posedge reset) begin            // to send data

 if (reset == 1'b1) begin
  t1       = 1'b0;
  cntr     = 4'b0000;
  data     = 8'b0110_1110;
  flag_rw  = 1'b0;
  data1    = data[0];
end

else begin

 if(flag_ack == 1'b1 && cntr_stop == 2'b11) begin
  data     = 8'b1010_1111;
  data1    = data[0];                                  // addr of rxr with read = 1
 end

if(stop == 1'b0) begin
```

```
      if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b01)
         data  = 8'b0111_0110;                                  // byte address to write

       if (flag_ack == 1'b1 && flag_rw == 1'b1 && cntr_stop == 2'b01)
         data  = 8'b0111_0110;                                  // byte address to read

       if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b10)
         data  = 8'b0111_0111;                                  // first data sent

       if (cntr < 4'b1000) begin                              // to transmit data to sda
         t1   =  data[7];
         data =  data << 1'b1;
         cntr =  cntr + 4'b0001;
       end

      else if (cntr == 4'b1000) begin
         flag_rw =  data1;
         cntr    = 4'b0000;
         t1      = 1'b0;
      end
    end
  end
end

//data:

always @ (posedge scl or posedge reset) begin                  // to receive data

 if (reset == 1'b1) begin
   cntr_stop = 2'b00;
   flag_ack  = 1'b0;
   cntr_ack  = 4'b0000;
   flag_stop = 1'b0;
   data2     = 8'b0000_0000;
end

else begin

  if (fg1_stop_low == 1'b1)                                    // to end stop depending on either clk
    flag_stop = 1'b0;

 if (flag_rw == 1'b0) begin

  if (cntr == 4'b0000) begin                                   // to increment after each ack
    cntr_stop = cntr_stop + 2'b01;
```

```
        if (en_bit == 1'b1)
          flag_ack  = 1'b1;

        if (cntr_stop == 2'b11)
          flag_stop  = 1'b1;

      end

      else if (cntr_stop == 2'b11)                    // to prevent further incrementation
        cntr_stop     = 2'b00;
       end
    else if (flag_rw == 1'b1) begin

      if (cntr == 4'b0000) begin                      // to increment after each ack
       cntr_stop = cntr_stop + 2'b01;

        if (en_bit == 1'b1)
          flag_ack  = 1'b1;

         if (cntr_stop == 2'b11 && sda == 1'b1)
           flag_stop  = 1'b1;

       end

      else if (cntr_stop == 2'b11)
       cntr_stop     = 2'b00;
      end


      if (en_bit == 1'b0)
        flag_ack = 1'b0;

      if (flag_rw == 1'b1 && cntr_ack < 4'b1000 && cntr_stop == 2'b10 &&
                                                      flag_ack  == 1'b0) begin
       data2    = data2 << 1'b1;
       data2[0] = sda;                                // to get the data from rxr
       cntr_ack = cntr_ack + 4'b0001;
      end

      else if (cntr_ack == 4'b1000)
       cntr_ack = cntr_ack + 4'b0001;

  end
end
```

```
always @ (posedge clk or posedge reset) begin
 if (reset == 1'b1) begin
   fg1_stop_low = 1'b0;
   cntr_start  = 2'b00;
   flag3_start  = 1'b0;
   flag1_start  = 1'b0;
   sda_l2h = 1'b0;
 end

 else begin

if (flag_stop == 1'b1 && sda_ack == 1'b0 && flag_rw == 1'b1)
  sda_l2h = 1'b1;

 if (start == 1'b1)
  cntr_start = cntr_start + 2'b01;

 if (cntr_start == 2'b01) begin
  flag1_start = 1'b1;

  if (flag2_start == 1'b1)
     flag3_start = 1'b1;
 end

 if (flag2_start == 1'b0)
  flag3_start = 1'b0;

 if (start == 1'b0)
  cntr_start = 2'b00;

 if (flag2_start == 1'b1 && flag3_start == 1'b0)
  flag1_start = 1'b0;

 if (flag_stop == 1'b1) begin

  if (start == 1'b1 && cntr == 4'b0000)
   fg1_stop_low = 1'b1;
 end

 else if (fg1_stop_low == 1'b1 && flag_stop == 1'b0)
  fg1_stop_low = 1'b0;

 end
end
```

```
always @ (negedge clk or posedge reset) begin

 if (reset == 1'b1) begin
  en_bit  = 1'b0;
  sda_ack = 1'b0;
  set     = 1'b0;
 end

 else begin

  if(sda == 1'b1 && flag1_start == 1'b1)
    set = 1'b1;                                    // to check whether sda is in Z

  if (sda_ack == 1'b0 && flag_rw == 1'b1 && (cntr_stop == 2'b10 || cntr_stop ==
   sda_ack = 1'b1;                                                2'b11) && cntr_ack != 4'b1001)

 if (cntr == 4'b1000) begin

  if (cntr_ack == 4'b0000)
   en_bit = 1'b1;                                  // to make sda high after addr txion

 end

 if (flag_ack == 1'b1 && (cntr_stop == 2'b01 || cntr_stop == 2'b10))
  en_bit = 1'b0;

 if (flag1_start == 1'b0)
  en_bit  = 1'b0;                                  // to give sda high imp after stop

 end
end

always @ (posedge sda or posedge reset) begin

 if (reset == 1'b1)
   flag2_start = 1'b0;

 else begin

  if (scl == 1'b1)
   flag2_start  = 1'b1;

 else if (flag2_start ==1'b1)
  flag2_start = 1'b0;
```

```
    end

  end
endmodule
```

7.4.2. DEVICE2:

```
module i2c1(clk2,start,sda,scl,reset);

 inout sda;
 inout scl;

 input start;
 input clk2;
 input reset;

 reg      t1;                          // internal register to assign for sda
 reg [7:0] data;                       // temp reg  to shift addr and data
 reg      data1;
 reg [3:0] cntr;                       // counter to shift data
 reg      en;                          // enable bit to assign clk pulse for scl
 reg      en_bit;
 reg      flag_stop;                   // to issue stop internally
 reg      flag_ack;                    // to detect ack
 reg [1:0] cntr_stop;                  // to issue stop
 reg      flag_rw;
 reg [3:0] cntr_ack;                   // to give acknowledge
 reg      sda_ack;
 reg      fg1_stop_low;
 reg [1:0] cntr_start;                 // to detect start at posedge clk only
 reg      flag1_start;                 // to activate internal start
 reg      flag2_start;                 // to stop internal start
 reg      flag3_start;                 // to stop internal start
 reg [7:0] data2;                      // received data from rxr
 reg      set2;
 reg      sda_l2h;

wire   stop;                           // to issue stop condition
wire   flag_start;                     // internal start

assign flag_start = ((flag2_start == 1'b1 && flag3_start == 1'b0) || flag1_start ==
                                                          1'b0) ? 1'b0 : 1'b1;

assign sda      = (sda_l2h == 1'b0 && en_bit == 1'b0 && flag_start == 1'b1 &&
```

```
                                    sda_ack == 1'b0 && set2 == 1'b1) ? t1 : 1'bz ;
  assign scl      = (en == 1'b1 && stop == 1'b0 && set == 1'b1) ? clk2 : 1'b1;
  assign stop     = (flag_stop == 1'b1 && fg1_stop_low == 1'b0) ? 1'b1 : 1'b0;

  //START CONDITION:

  always @ (negedge sda or posedge reset) begin

   if (reset == 1'b1)
     en = 1'b0;

   else begin

    if (scl == 1'b1 && stop == 1'b0)
       en = 1'b1;                                    // used to assign clock

  end

end


always @ (negedge scl or posedge reset) begin          // to send data

 if (reset == 1'b1) begin
   t1       = 1'b0;
   cntr     = 4'b0000;
   data     = 8'b0110_1110;
   flag_rw  = 1'b0;
   data1    = data[0];
end

else begin

 if(flag_ack == 1'b1 && cntr_stop == 2'b11) begin
   data     = 8'b1010_1111;
   data1    = data[0];                            // addr of rxr with read = 1
 end

if(stop == 1'b0) begin

 if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b01)
    data  = 8'b0111_0110;                          // byte address to write
```

```
        if (flag_ack == 1'b1 && flag_rw == 1'b1 && cntr_stop == 2'b01)
          data  = 8'b0111_0110;                          // byte address to read

        if (flag_ack == 1'b1 && flag_rw == 1'b0 && cntr_stop == 2'b10)
          data  = 8'b0111_0111;                          // first data sent

        if (cntr < 4'b1000) begin                        // to transmit data to sda
          t1  = data[7];
          data = data << 1'b1;
          cntr = cntr + 4'b0001;
        end

        else if (cntr == 4'b1000) begin
          flag_rw = data1;
          cntr   = 4'b0000;
          t1     = 1'b0;
        end
      end
    end
  end

  //data:

  always @ (posedge scl or posedge reset) begin          // to receive data
    if (reset == 1'b1) begin
      cntr_stop = 2'b00;
      flag_ack  = 1'b0;
      cntr_ack  = 4'b0000;
      flag_stop = 1'b0;
      data2     = 8'b0000_0000;
    end

    else begin

      if (fg1_stop_low == 1'b1)                          // to end stop depending on either
clk
        flag_stop = 1'b0;

      if (flag_rw == 1'b0) begin

      if (cntr == 4'b0000) begin                         // to increment after each ack
        cntr_stop = cntr_stop + 2'b01;

      if (en_bit == 1'b1)
        flag_ack  = 1'b1;
```

```
        if (cntr_stop == 2'b11)
          flag_stop   = 1'b1;

      end

      else if (cntr_stop == 2'b11)                    // to prevent further incrementation
        cntr_stop     = 2'b00;
       end
     else if (flag_rw == 1'b1) begin

      if (cntr == 4'b0000) begin                      // to increment after each ack
       cntr_stop = cntr_stop + 2'b01;

        if (en_bit == 1'b1)
          flag_ack  = 1'b1;

         if (cntr_stop == 2'b11 && sda == 1'b1)
           flag_stop   = 1'b1;

        end

       else if (cntr_stop == 2'b11)
        cntr_stop     = 2'b00;
      end


      if (en_bit == 1'b0)
        flag_ack = 1'b0;

      if (flag_rw == 1'b1 && cntr_ack < 4'b1000 && cntr_stop == 2'b10 && flag_ack
                                                                    == 1'b0) begin
       data2    = data2 << 1'b1;
       data2[0] = sda;                                // to get the data from rxr
       cntr_ack = cntr_ack + 4'b0001;
      end

      else if (cntr_ack == 4'b1000)
       cntr_ack = cntr_ack + 4'b0001;

  end
end


always @ (posedge clk2 or posedge reset) begin
```

```
      if (reset == 1'b1) begin
        fg1_stop_low = 1'b0;
        cntr_start  = 2'b00;
        flag3_start  = 1'b0;
        flag1_start  = 1'b0;
        sda_l2h = 1'b0;
      end

      else begin

      if (flag_stop == 1'b1 && sda_ack == 1'b0 && flag_rw == 1'b1)
        sda_l2h = 1'b1;

       if (start == 1'b1)
        cntr_start = cntr_start + 2'b01;

       if (cntr_start == 2'b01) begin
        flag1_start = 1'b1;

        if (flag2_start == 1'b1)
            flag3_start = 1'b1;
      end

      if (flag2_start == 1'b0)
        flag3_start = 1'b0;

     if (start == 1'b0)
       cntr_start = 2'b00;

     if (flag2_start == 1'b1 && flag3_start == 1'b0)
       flag1_start = 1'b0;

     if (flag_stop == 1'b1) begin

      if (start == 1'b1 && cntr == 4'b0000)
        fg1_stop_low = 1'b1;
    end

    else if (fg1_stop_low == 1'b1 && flag_stop == 1'b0)
      fg1_stop_low = 1'b0;

  end
end

always @ (negedge clk2 or posedge reset) begin
```

```
        if (reset == 1'b1) begin
         en_bit  = 1'b0;
         sda_ack = 1'b0;
         set 2   = 1'b0;
        end

        else begin

         if(sda == 1'b1 && flag1_start == 1'b1)
           set2 = 1'b1;                                    // to check whether sda is in Z
         if (sda_ack == 1'b0 && flag_rw == 1'b1 && (cntr_stop == 2'b10 || cntr_stop ==
         sda_ack = 1'b1;                                          2'b11) && cntr_ack != 4'b1001)

        if (cntr == 4'b1000) begin

         if (cntr_ack == 4'b0000)
          en_bit = 1'b1;                                 // to make sda high after addr txion
         end

       if (flag_ack == 1'b1 && (cntr_stop == 2'b01 || cntr_stop == 2'b10))
        en_bit = 1'b0;

       if (flag1_start == 1'b0)
        en_bit  = 1'b0;                                   // to give sda high imp after stop

     end
    end

    always @ (posedge sda or posedge reset) begin

     if (reset == 1'b1)
       flag2_start = 1'b0;

    else begin

     if (scl == 1'b1)
      flag2_start  = 1'b1;

    else if (flag2_start ==1'b1)
     flag2_start = 1'b0;

  end

end
```

```
endmodule
```

7.4.3. **EEPROM:**

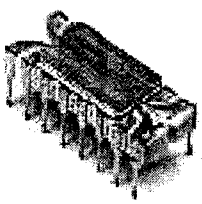
```
module i2c1_rx(start,reset,clk3,sda,scl);

 inout sda;
 inout scl;

 input clk3;
 input start;
 input reset;

 reg [6:0] addr;              // to verify address
 reg [3:0] cntr_tb;           // to shift data and rx
 reg       rw;                // to receive lsb of addr
 reg [1:0] incr_tb;           // to distinguish data, addr and byte addr
 reg [7:0] tempdata_tb;       // to store the data
 reg [7:0] rx_data;           // data for rx
 reg       sda_det;           // temporary reg to shift to sda
 reg       fg_rw;             // detect rw in neg edge
 reg [3:0] bit_cntr;          // to make sda high after data txion
 reg       sda_high;          // to make sda high after data txion
 reg       fg_incr_tb;        // to make incr_tb zero after stop
 reg       fg1_start_tb;      // to initiate the start condition
 reg       fg2_start_tb;      // to terminate start
 reg [7:0] byte_addr;         // to assign byte addr
 reg       w1_en1;            // to make en1 high
 reg       w2_en1 ;           // to make en1 low


 reg [7:0] mem[255:0];        // to create memory for eeprom

 wire      fg_start_tb;       // instead of start
 wire      en1;

 assign en1      = (w1_en1 == 1'b1 && w2_en1 == 1'b0) ? 1'b1 :1'b0;

 assign sda      = ((en1 == 1'b1 || fg_rw == 1'b1) && sda_high == 1'b0) ? sda_det :
                                                                          1'bz;
 assign fg_start_tb = (fg1_start_tb == 1'b1 && fg2_start_tb == 1'b0) ? 1'b1 : 1'b0;
```

```
always @ (posedge scl or posedge reset) begin                    // to rx

 if (reset == 1'b1) begin
   cntr_tb    = 4'b0000;
   addr       = 7'b000_0000;
   rw         = 1'b0;
   tempdata_tb = 8'b0000_0000;
 end

 else begin

  if (fg_rw == 1'b0) begin                                       // to detect  tx or rx

    if (fg_start_tb == 1'b1) begin                               // only if start = 1 it receives data

     if (incr_tb == 1'b0) begin                                  // to detect addr or data

      if (cntr_tb < 4'b0111) begin                               // to receive addr
        addr    = addr << 1'b1;
        addr[0] = sda;
        cntr_tb = cntr_tb + 4'b0001;
      end

      else if (cntr_tb == 4'b0111 ) begin                        // to detect tx or rx
        rw     = sda;
        cntr_tb = cntr_tb + 1'b1;
      end

      else if(cntr_tb == 4'b1000)
        cntr_tb = 4'b0000;
    end

    else begin                                                   // to receive data

     if (cntr_tb < 4'b1000)  begin
       tempdata_tb    = tempdata_tb << 1'b1;
       tempdata_tb[0] = sda;
       cntr_tb        = cntr_tb + 4'b0001;
     end

     else if (cntr_tb == 4'b1000 )
      cntr_tb = 4'b0000;

    end
  end
```

```
      end
    end
  end

  always @(negedge scl or posedge reset) begin                    // to give ack and send
data

  if (reset == 1'b1) begin
    sda_det  = 1'b0;
    fg_rw    = 1'b0;
    bit_cntr = 4'b0000;
    sda_high = 1'b0;
    byte_addr = 8'b0000_0000;
    rx_data  = 8'b0000_0000;
  end

  else begin

  if (incr_tb == 2'b01 && cntr_tb == 4'b1000)
    byte_addr = tempdata_tb;

  if (incr_tb == 2'b10 && cntr_tb == 4'b1000)  begin
    mem [byte_addr] = tempdata_tb;
    rx_data       =  mem [byte_addr];
  end

  if (rw == 1'b1 && en1 == 1'b0 && incr_tb == 2'b10) begin          // to tx data
    sda_det  = rx_data[7];
    rx_data  = rx_data << 1'b1;
    bit_cntr = bit_cntr + 4'b0001;

    if (bit_cntr == 4'b1001) begin
      bit_cntr = 4'b0000;
      sda_high = 1'b1;
    end

    if (cntr_tb == 4'b0000 && rw == 1'b1 && incr_tb == 2'b10)
      fg_rw  = 1'b1;
  end
 end
end

always @ (posedge sda or posedge reset) begin                     // to issue stop

 if (reset == 1'b1) begin
   fg2_start_tb = 1'b0;
```

```
      fg_incr_tb   = 1'b0;
    end

    else begin

    if(scl == 1'b1 && reset == 1'b0) begin
      fg_incr_tb  = 1'b1;
      fg2_start_tb = 1'b1;
    end

    else if(fg2_start_tb == 1'b1)
     fg2_start_tb = 1'b0;

    else if(fg_incr_tb == 1'b1)
      fg_incr_tb = 1'b0;

   end
  end

  always @ (posedge clk3 or posedge reset) begin

  if (reset == 1'b1) begin
    w1_en1    = 1'b0;
    incr_tb = 2'b00;
  end

 else begin

  if (cntr_tb == 4'b1000 && (incr_tb == 2'b01 || incr_tb == 2'b10)) begin
    incr_tb = incr_tb + 2'b01;
    w1_en1    = 1'b1;
  end

  else if (w1_en1 == 1'b1)
   w1_en1 = 1'b0;

  else if (cntr_tb == 4'b1000 && incr_tb == 2'b00) begin

   if(addr == 7'b1010_101) begin                          // to give ack for addr
     w1_en1    = 1'b1;
     incr_tb = 2'b01;
   end
 end

 if(fg_incr_tb == 1'b1)
  incr_tb = 2'b00;
```

```
    end
  end

  always @ (negedge clk3 or posedge reset) begin

  if (reset == 1'b1) begin
     w2_en1    = 1'b0;
   end
   else begin

  if (w1_en1 == 1'b1)
       w2_en1 = 1'b1;
  else if (w2_en1 == 1'b1)
    w2_en1 = 1'b0;
   end

 end


  always @ (posedge start or posedge reset) begin                    // internal start

  if (reset == 1'b1)

   fg1_start_tb = 1'b0;

 else
  fg1_start_tb = 1'b1;

end

endmodule
```



SIMULATION OUTPUT

CHAPTER-8

8. SIMULATED OUTPUT:

8.1 REPORT MESSAGES:

8.1.1. I2C WITH NOT ACKNOWLEDGE:

```
Beginning Compile
Beginning Phase I
Compiling source file: F:\I^2bus\finalcoding\i2c_not_ack\clock_i2c.v
Compiling source file: F:\I^2bus\finalcoding\i2c_not_ack\I2c1_tb.v
Compiling source file: F:\I^2bus\finalcoding\i2c_not_ack\I2c1.v
Compiling source file: F:\I^2bus\finalcoding\i2c_not_ack\i2c_rx_syn.v
Compiling source file: F:\I^2bus\finalcoding\i2c_not_ack\I2c1_top.v
Finished Phase I
Entering Phase II...
Finished Phase II
Entering Phase III...
Finished Phase III
Highest level modules:  i2c1_top
Finding handle to i2c1_top.w_clk
Finding handle to i2c1_top.w_start
Finding handle to i2c1_top.w_sda
Finding handle to i2c1_top.w_scl
Finding handle to i2c1_top.w_reset
Finding handle to i2c1_top.w_clk2
Warning: Could not find handle to net i2c1_top.w_clk2 for addition to watched list
Finding handle to i2c1_top.w_clk3
Warning: Could not find handle to net i2c1_top.w_clk3 for addition to watched list
Compile Complete
.
Running...
        715start=0,reset=0
Exiting TestBencher Pro at simulation time 1515000
0 Errors, 0 Warnings
Compile time = 0.00000, Load time = 0.00000, Execution time = 0.00000

Normal exit
```

8.1.2. I2C WITH ACKNOWLEDGE:

```
Beginning Compile
Beginning Phase I
Compiling source file: F:\I^2bus\finalcoding\i2c_no&with_ack\i2c_not_ack\clock_i2c.v
Compiling source file: F:\I^2bus\finalcoding\i2c_no&with_ack\i2c_not_ack\I2c1_tb.v
Compiling source file: F:\I^2bus\finalcoding\i2c_no&with_ack\i2c_not_ack\I2c1_rx.v
Compiling source file: F:\I^2bus\finalcoding\i2c_no&with_ack\i2c_not_ack\I2c1.v
Compiling source file: F:\I^2bus\finalcoding\i2c_no&with_ack\i2c_not_ack\I2c1_top.v
Finished Phase I
Entering Phase II...
Finished Phase II
Entering Phase III...
Finished Phase III
Highest level modules:  i2c1_top
Finding handle to i2c1_top.w_clk
Finding handle to i2c1_top.w_start
Finding handle to i2c1_top.w_sda
Finding handle to i2c1_top.w_scl
Finding handle to i2c1_top.w_reset
Finding handle to i2c1_top.w_clk2

Compile Complete
.
Running...
        720start=0,reset=0
Exiting TestBencher Pro at simulation time 1520000
0 Errors, 0 Warnings
Compile time = 0.00000, Load time = 0.05000, Execution time = 0.00000

Normal exit
```

8.1.3. ARBITRATION FOR DEVICES HAVING SAME CLOCK RATE:

```
Beginning Compile
Beginning Phase I
Compiling source file: F:\I^2bus\finalcoding\arbit with same
clk\arbit_i2c_syn\clock_i2c.v
Compiling source file: F:\I^2bus\finalcoding\arbit with same
clk\arbit_i2c_syn\I2c1_top.v
Compiling source file: F:\I^2bus\finalcoding\arbit with same
clk\arbit_i2c_syn\I2c1_tb.v
Compiling source file: F:\I^2bus\finalcoding\arbit with same
clk\arbit_i2c_syn\I2c1_rx_syn.v
Compiling source file: F:\I^2bus\finalcoding\arbit with same clk\arbit_i2c_syn\I2c1.v
```

Compiling source file: F:\I^2bus\finalcoding\arbit with same clk\arbit_i2c_syn\clock2.v
Compiling source file: F:\I^2bus\finalcoding\arbit with same clk\arbit_i2c_syn\clock_rx.v
Compiling source file: F:\I^2bus\finalcoding\arbit with same clk\arbit_i2c_syn\i2c2.v
Finished Phase I
Entering Phase II...
Finished Phase II
Entering Phase III...
Finished Phase III
Highest level modules: i2c1_top
Finding handle to i2c1_top.w_clk
Finding handle to i2c1_top.w_start
Finding handle to i2c1_top.w_sda
Finding handle to i2c1_top.w_scl
Finding handle to i2c1_top.w_reset
Finding handle to i2c1_top.w_clk2
Finding handle to i2c1_top.w_clk3
Compile Complete
.
Running...
720start=0,reset=0
Exiting TestBencher Pro at simulation time 1520000
0 Errors, 0 Warnings
Compile time = 0.05000, Load time = 0.06000, Execution time = 0.00000

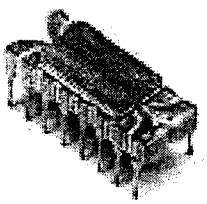
Normal exit

8.1.4. ARBITRATION FOR DEVICES HAVING DIFFERENT CLOCKS:

Beginning Compile
Beginning Phase I
Compiling source file: C:\arbit_i2c_diff_clk\clock_i2c.v
Compiling source file: C:\arbit_i2c_diff_clk\I2c1_top.v
Compiling source file: C:\arbit_i2c_diff_clk\I2c1_tb.v
Compiling source file: C:\arbit_i2c_diff_clk\I2c1_rx.v
Compiling source file: C:\arbit_i2c_diff_clk\I2c1.v
Compiling source file: C:\arbit_i2c_diff_clk\clock2.v
Compiling source file: C:\arbit_i2c_diff_clk\clock_rx.v
Compiling source file: C:\arbit_i2c_diff_clk\i2c2.v
Finished Phase I
Entering Phase II...
Finished Phase II
Entering Phase III...
Finished Phase III
Highest level modules: i2c1_top

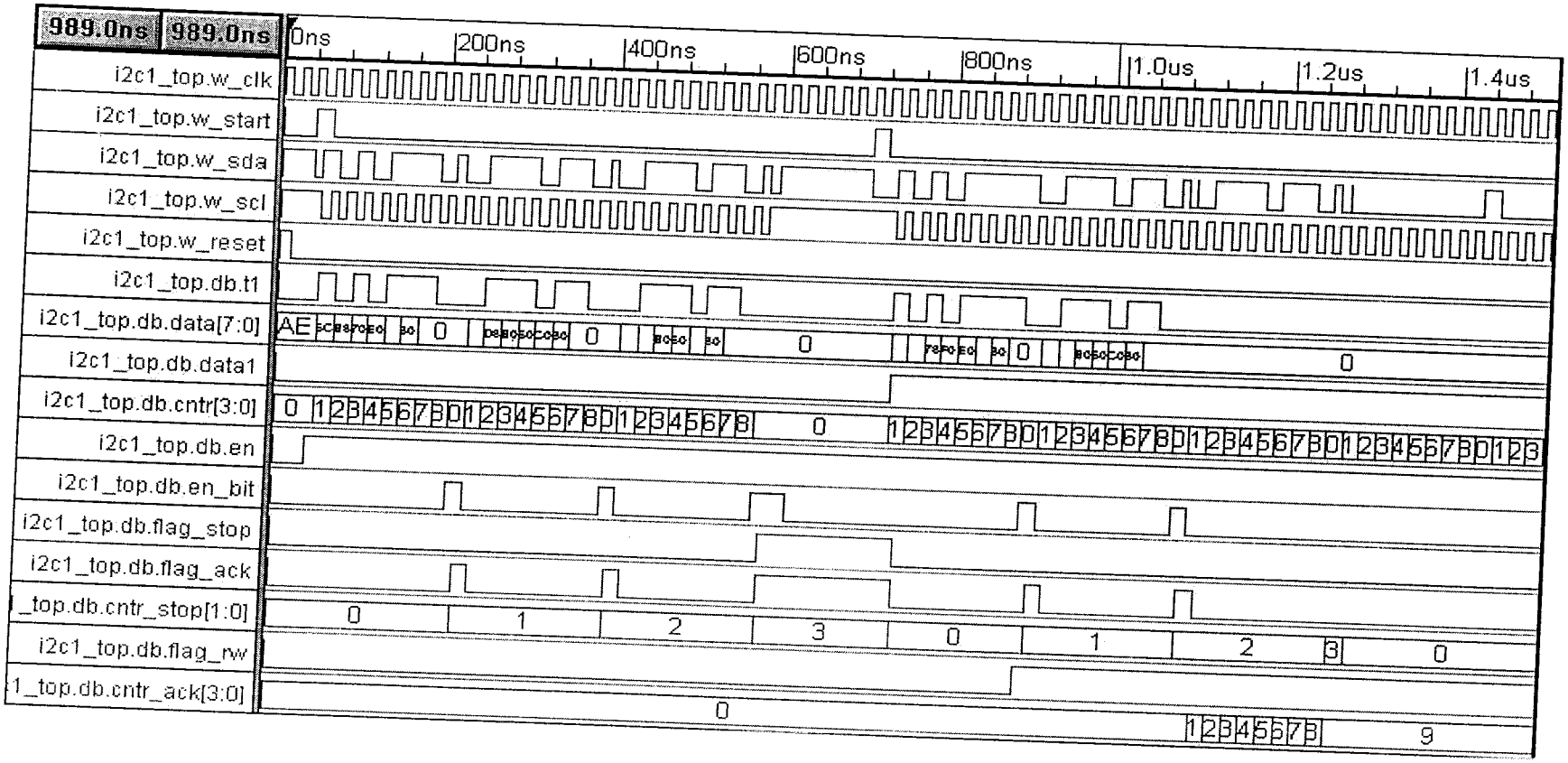
```
Finding handle to i2c1_top.w_clk
Finding handle to i2c1_top.w_start
Finding handle to i2c1_top.w_sda
Finding handle to i2c1_top.w_scl
Finding handle to i2c1_top.w_reset
Finding handle to i2c1_top.w_clk2
Finding handle to i2c1_top.w_clk3
Compile Complete
.
Running...
          720start=0,reset=0
Exiting TestBencher Pro at simulation time 1520000
0 Errors, 0 Warnings
Compile time = 0.11000, Load time = 0.22000, Execution time = 0.05000

Normal exit
```

CHAPTER-9

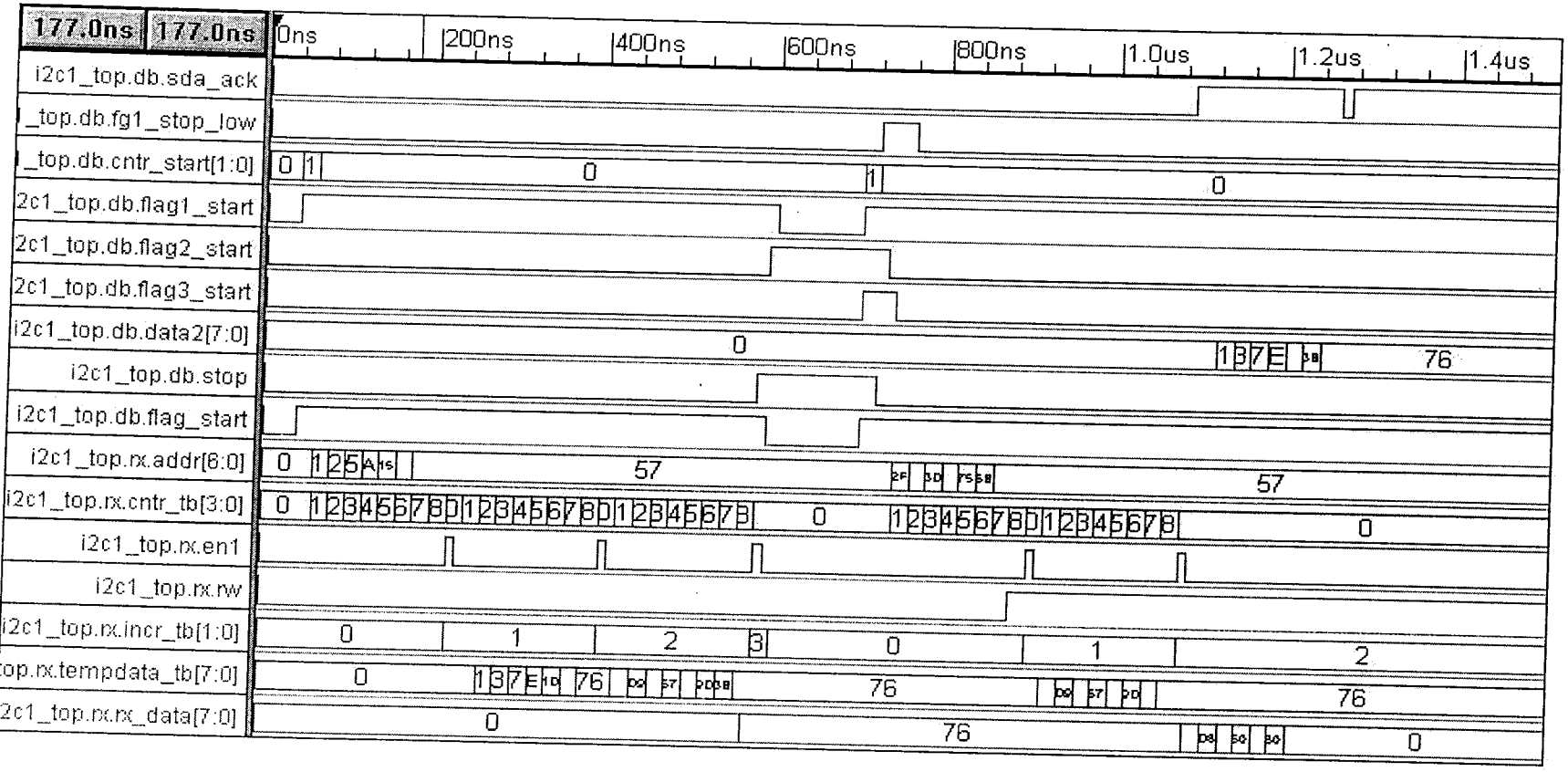


SIMULATION WAVEFORM

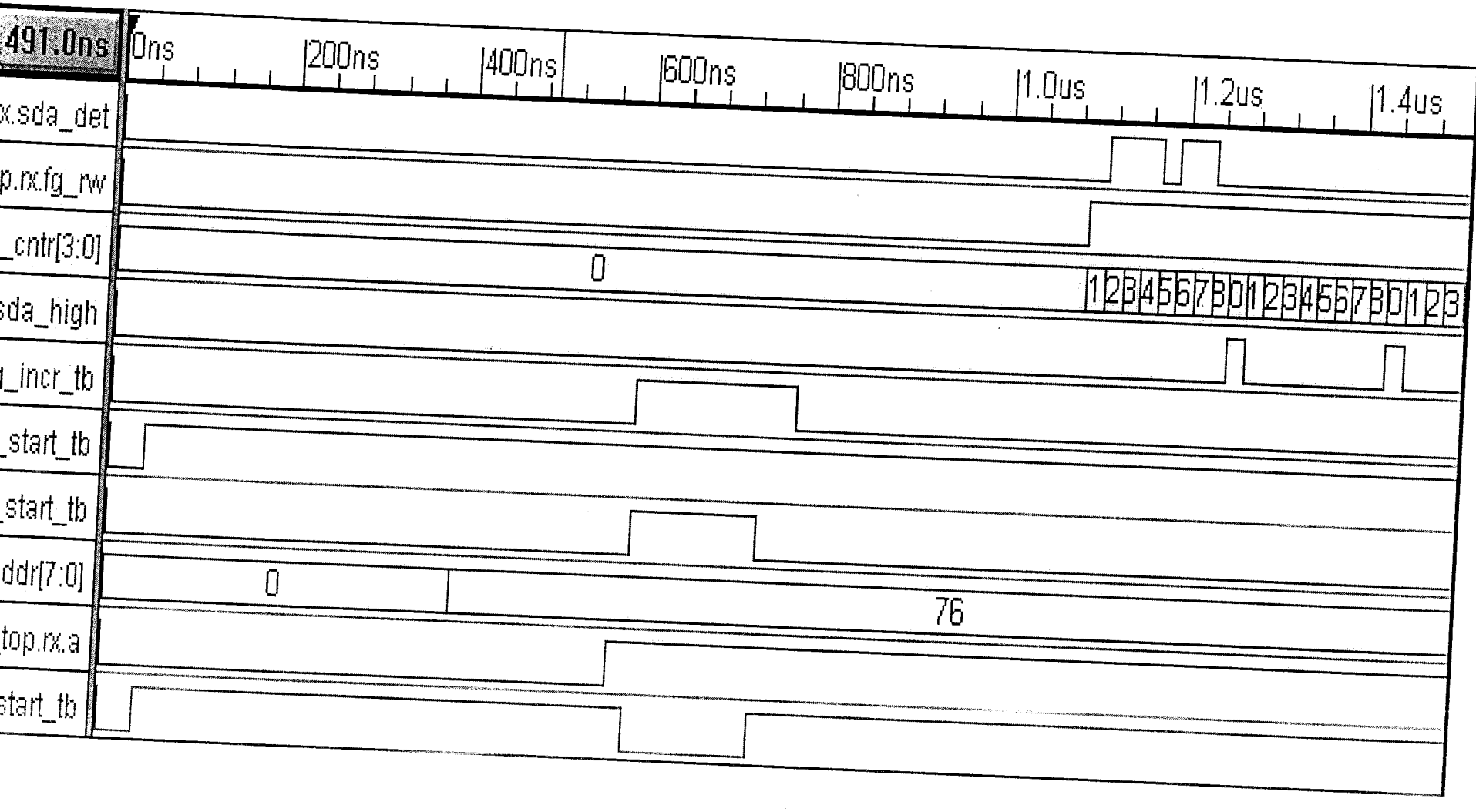
I2C WITH NO ACKNOWLEDGE



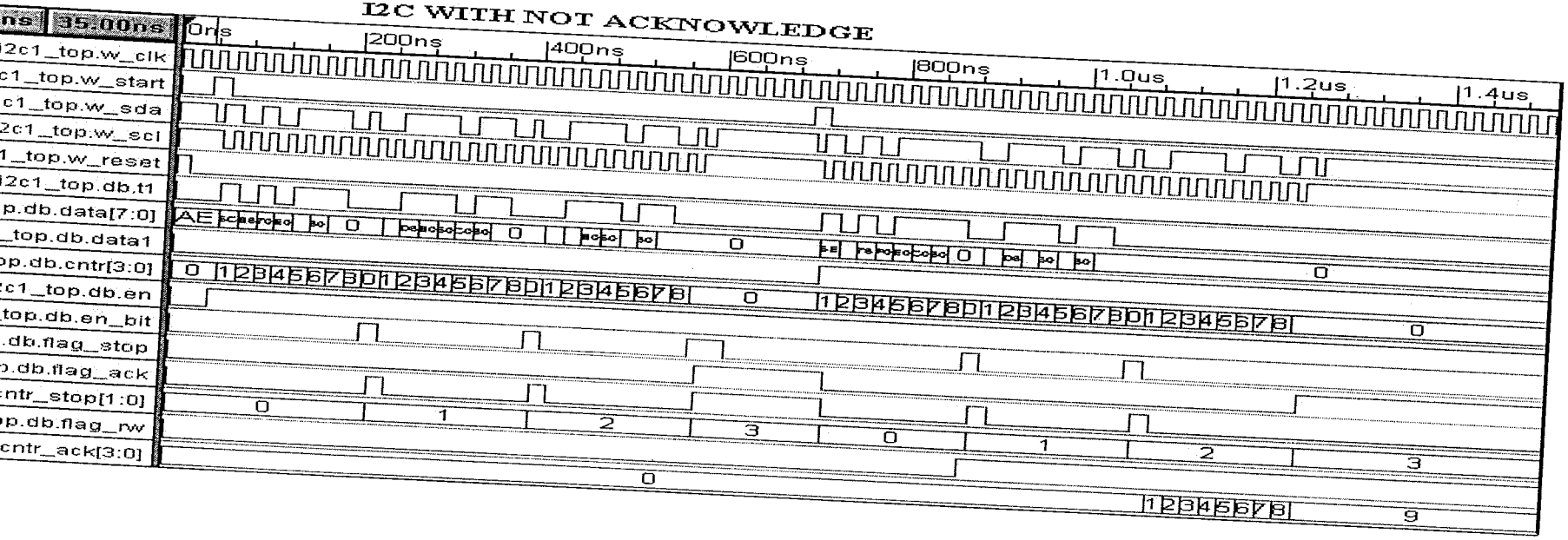
I2C WITH NO ACKNOWLEDGE



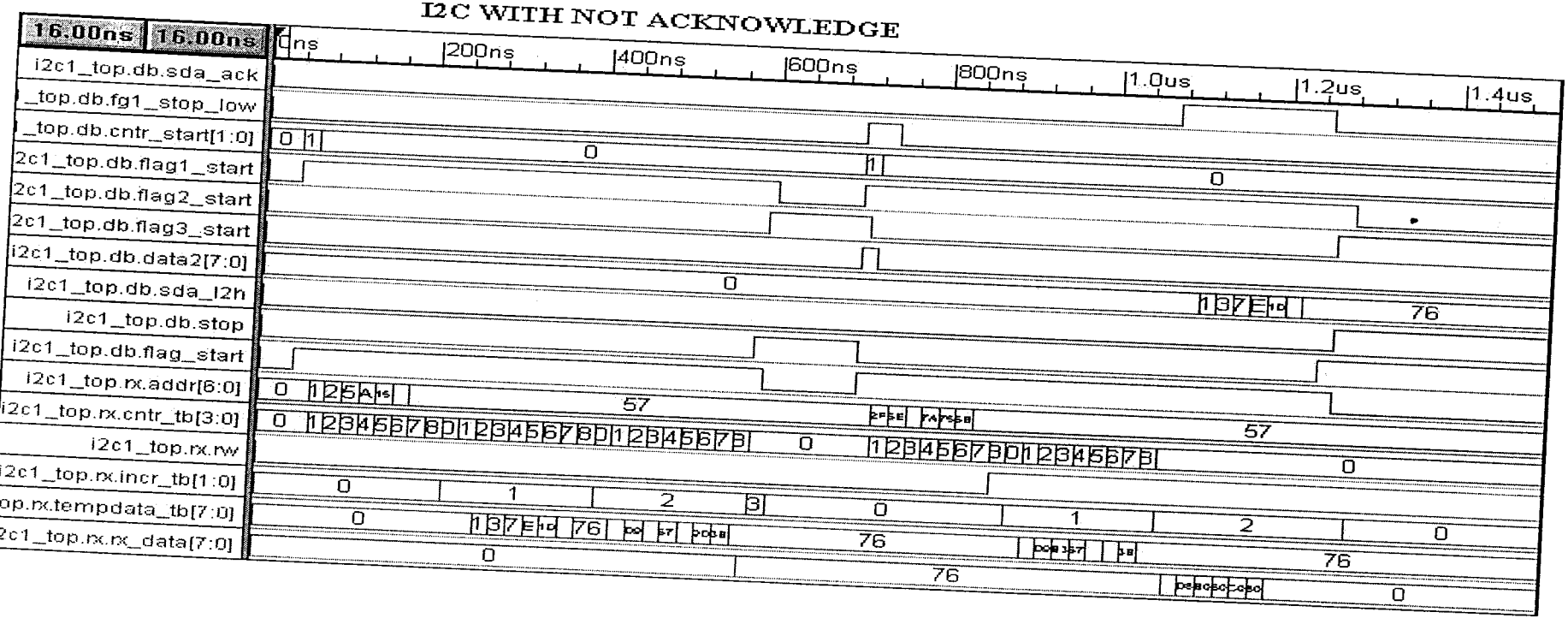
I2C WITH NO ACKNOWLEDGE



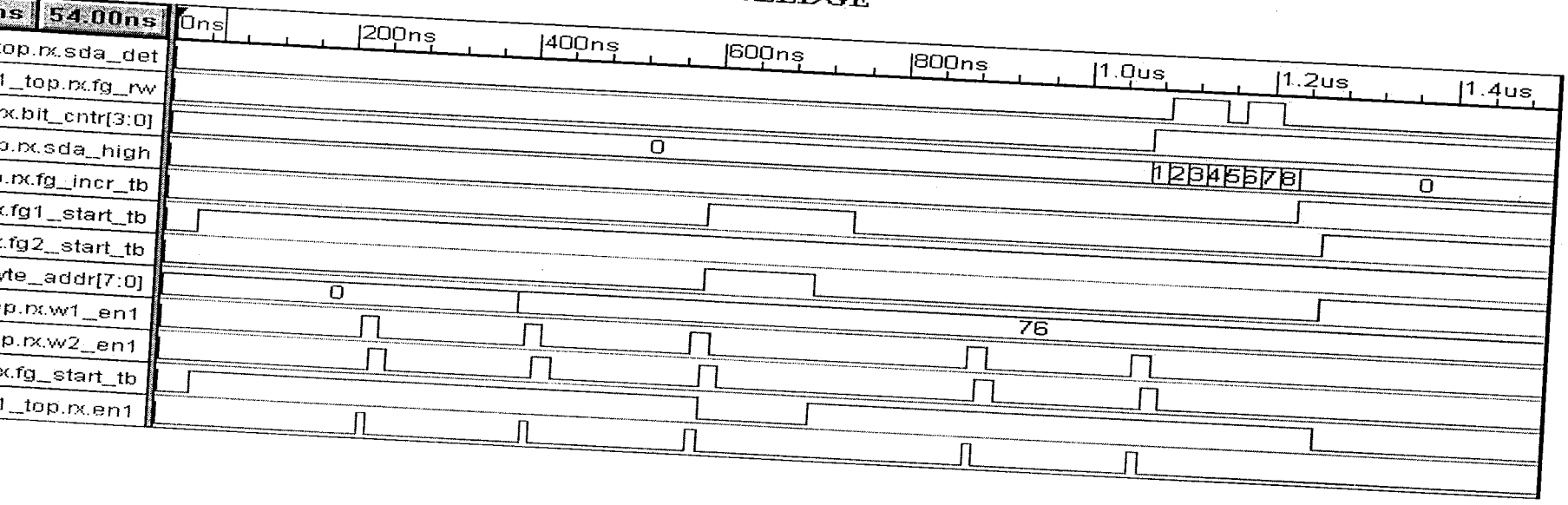
I2C WITH NOT ACKNOWLEDGE



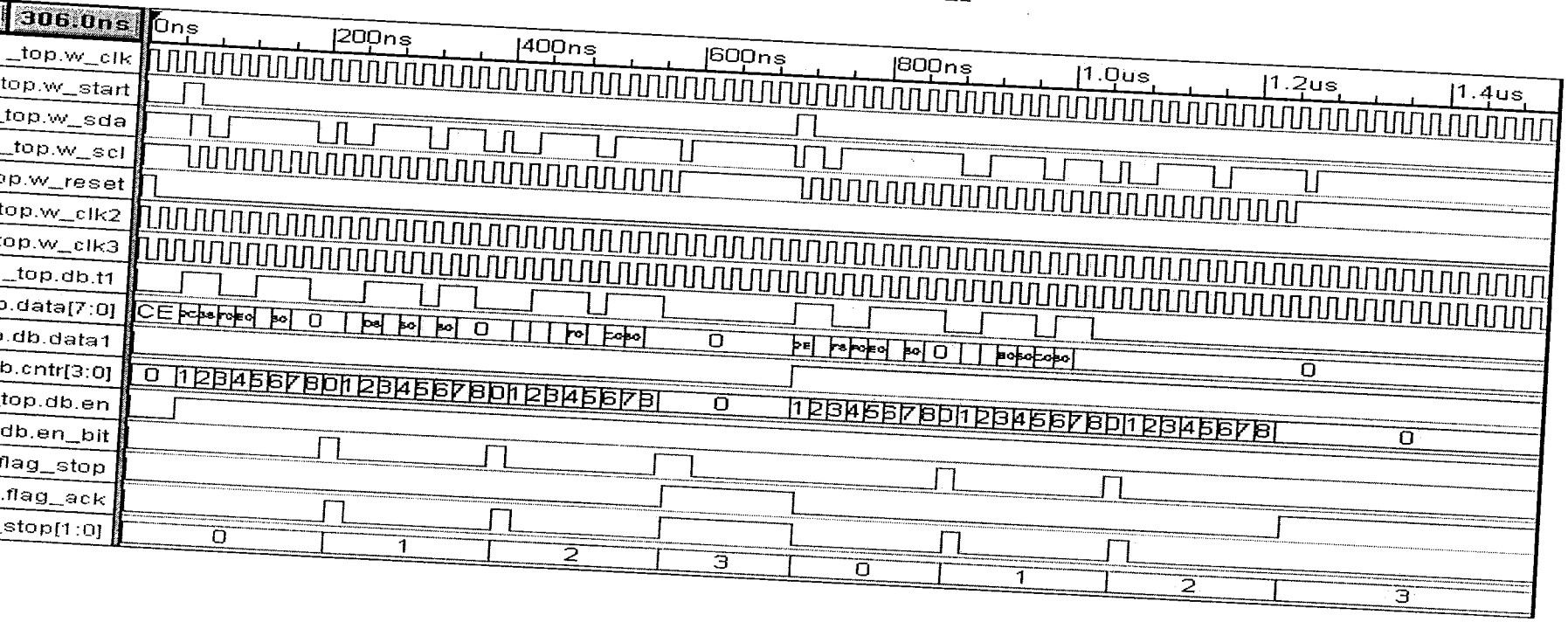
I2C WITH NOT ACKNOWLEDGE



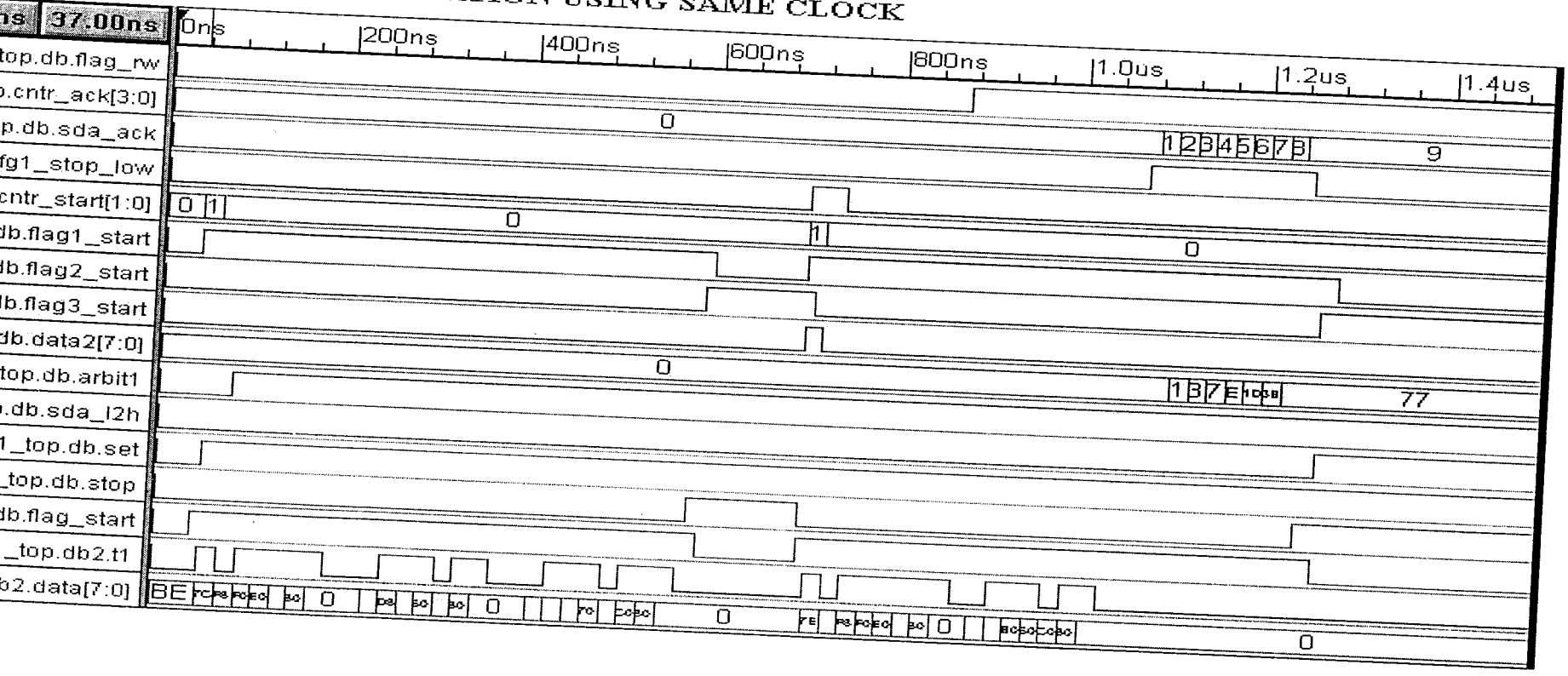
I2C WITH NOT ACKNOWLEDGE



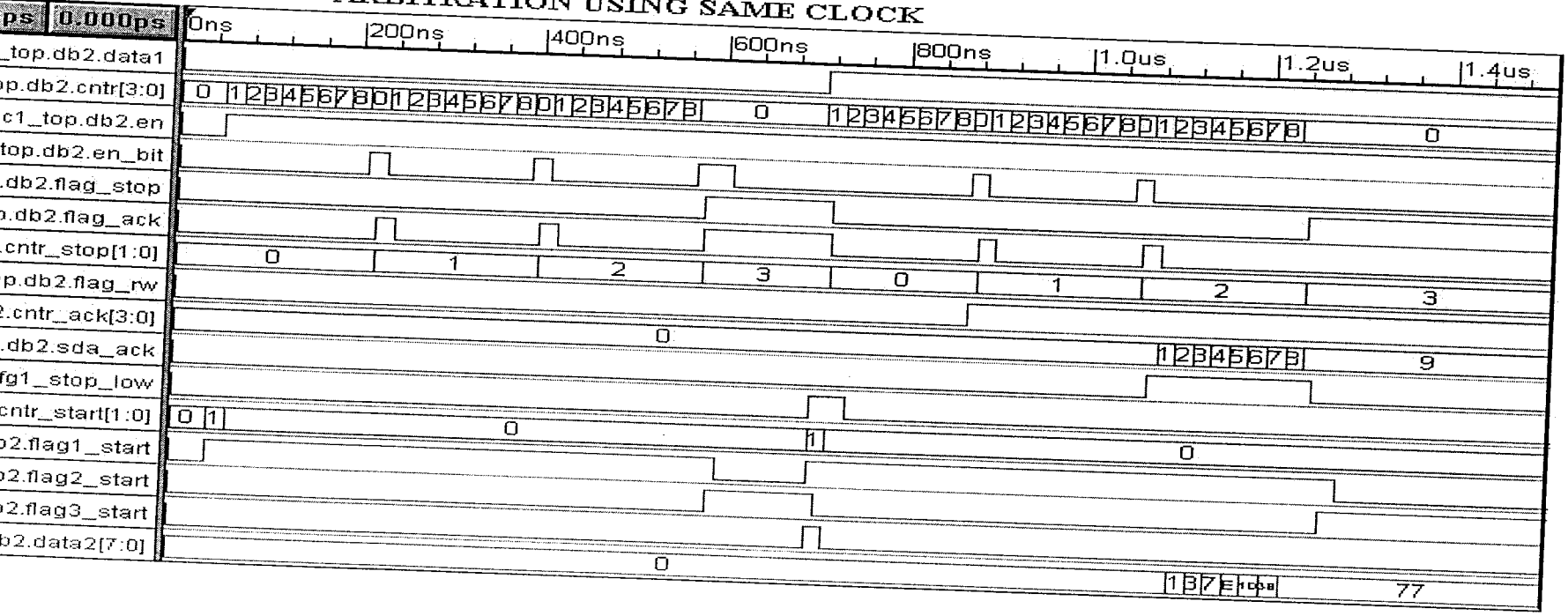
ARBITRATION USING SAME CLOCK



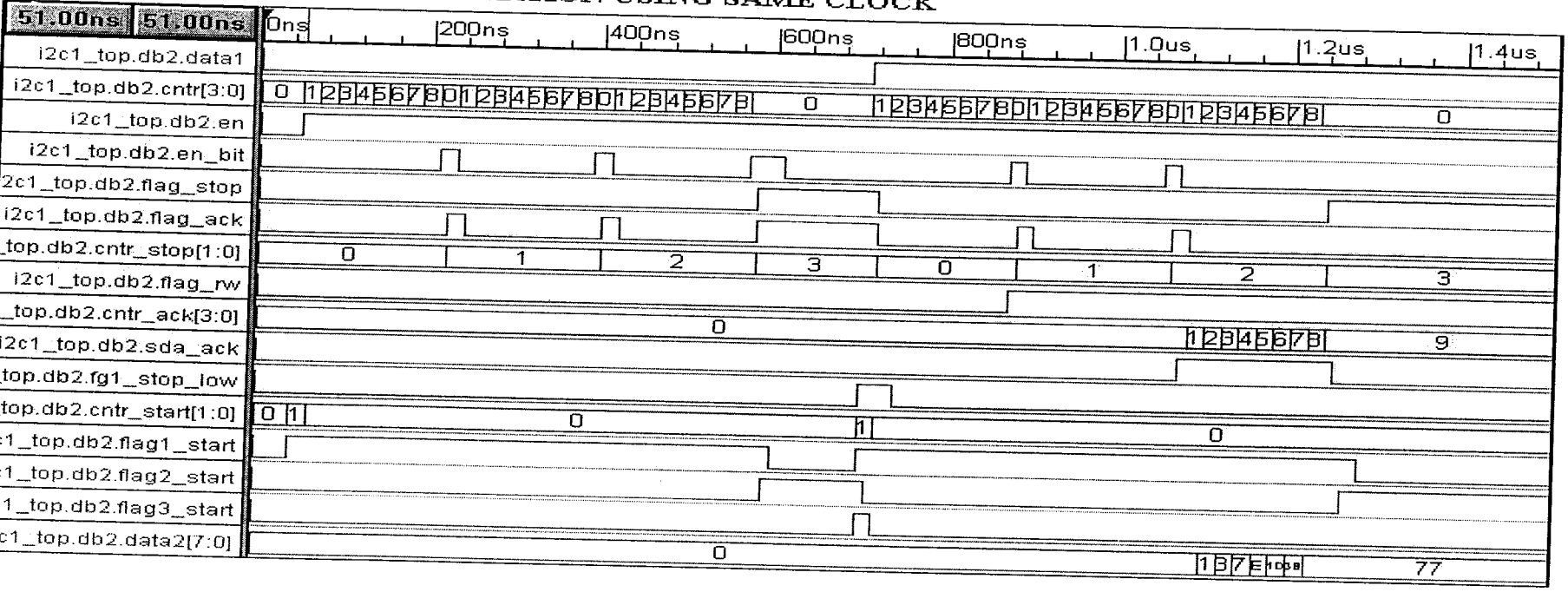
ARBITRATION USING SAME CLOCK



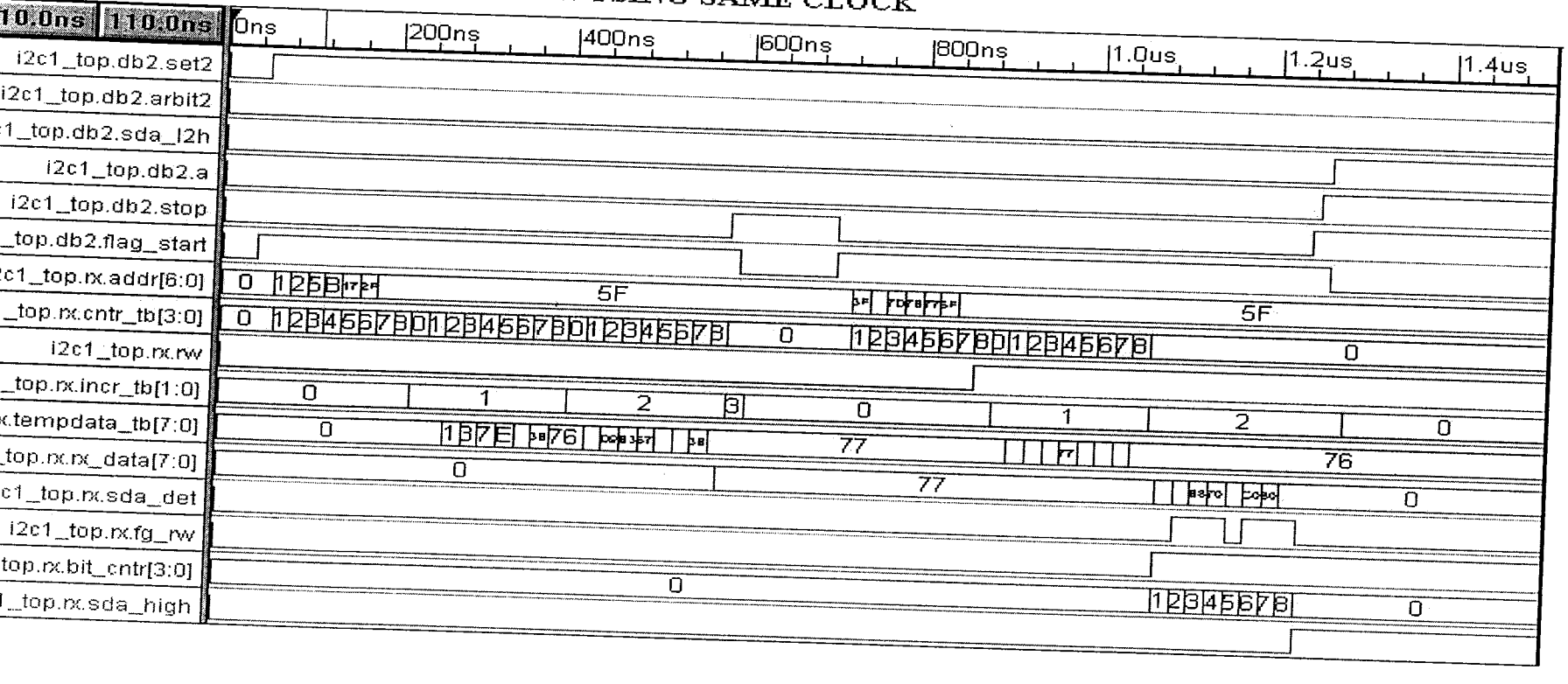
ARBITRATION USING SAME CLOCK



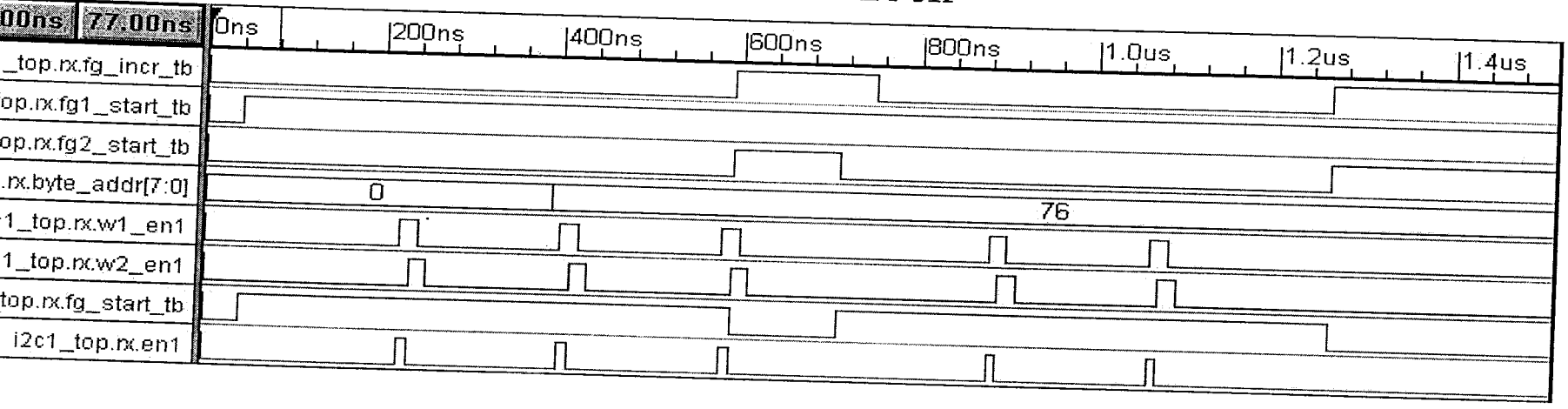
ARBITRATION USING SAME CLOCK



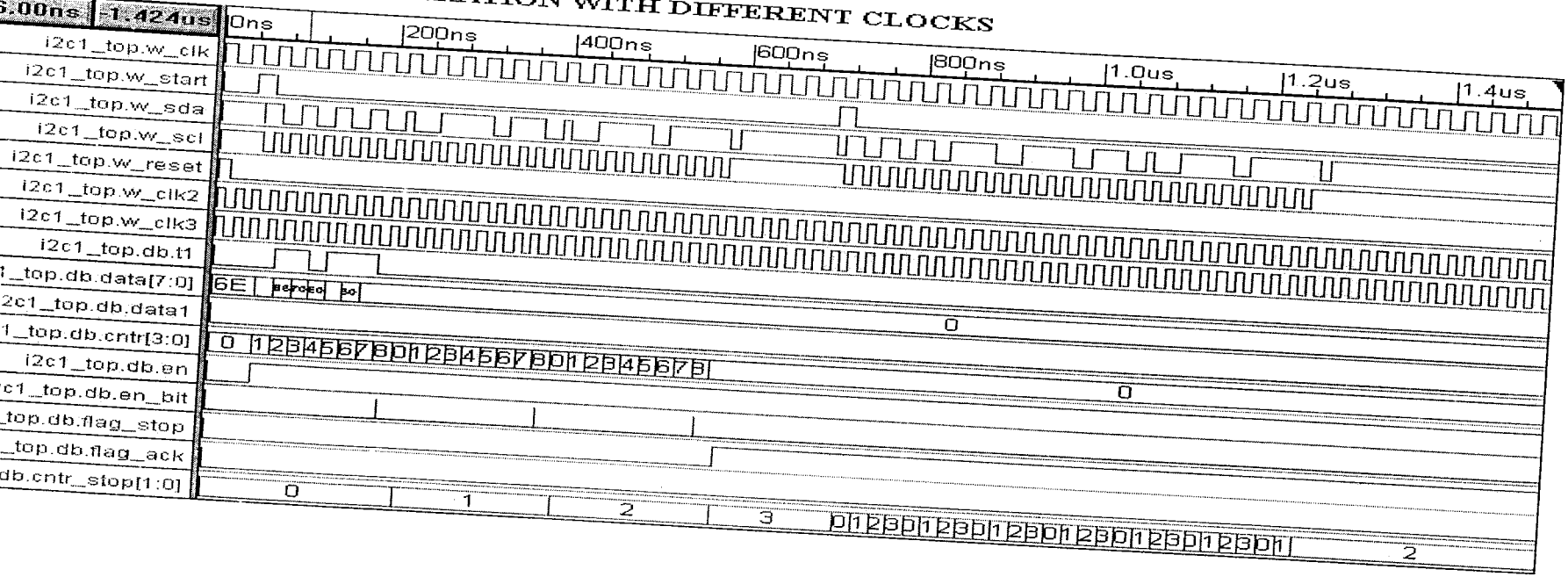
ARBITRATION USING SAME CLOCK



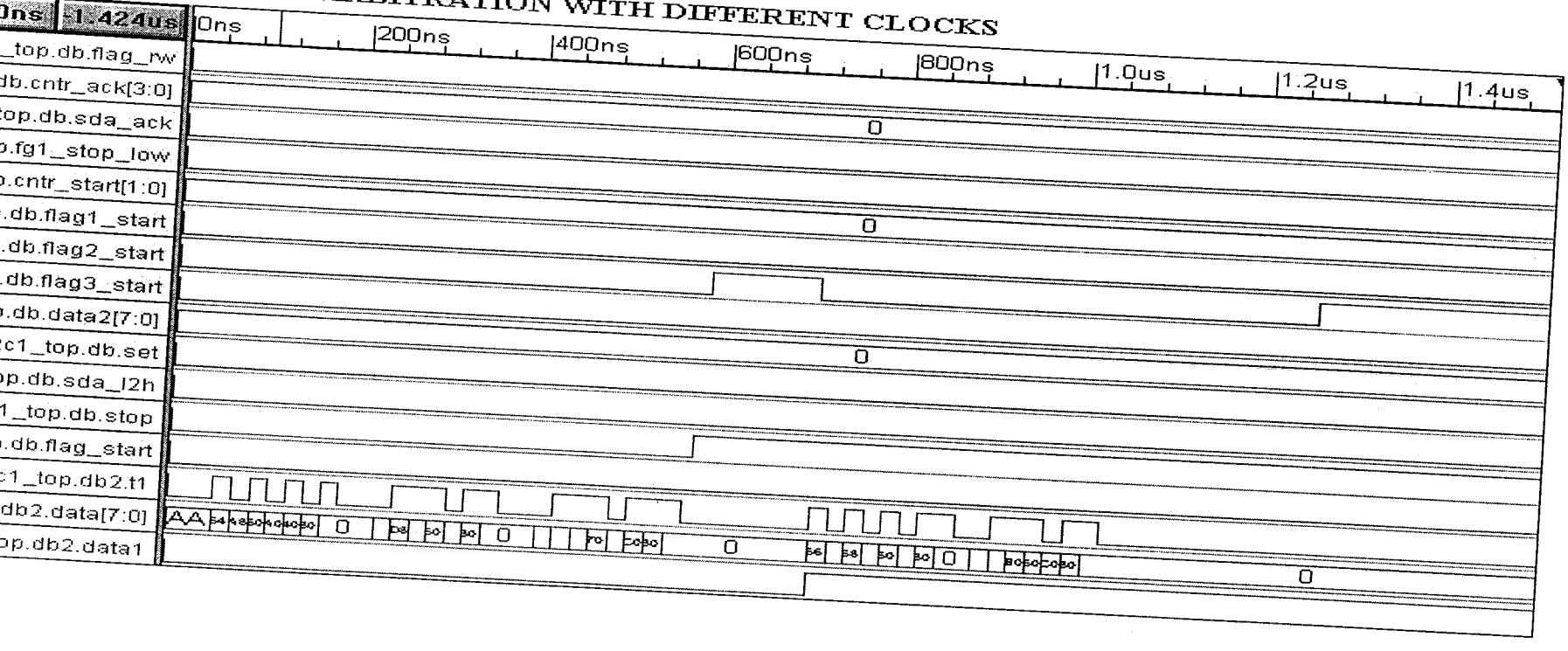
ARBITRATION USING SAME CLOCK



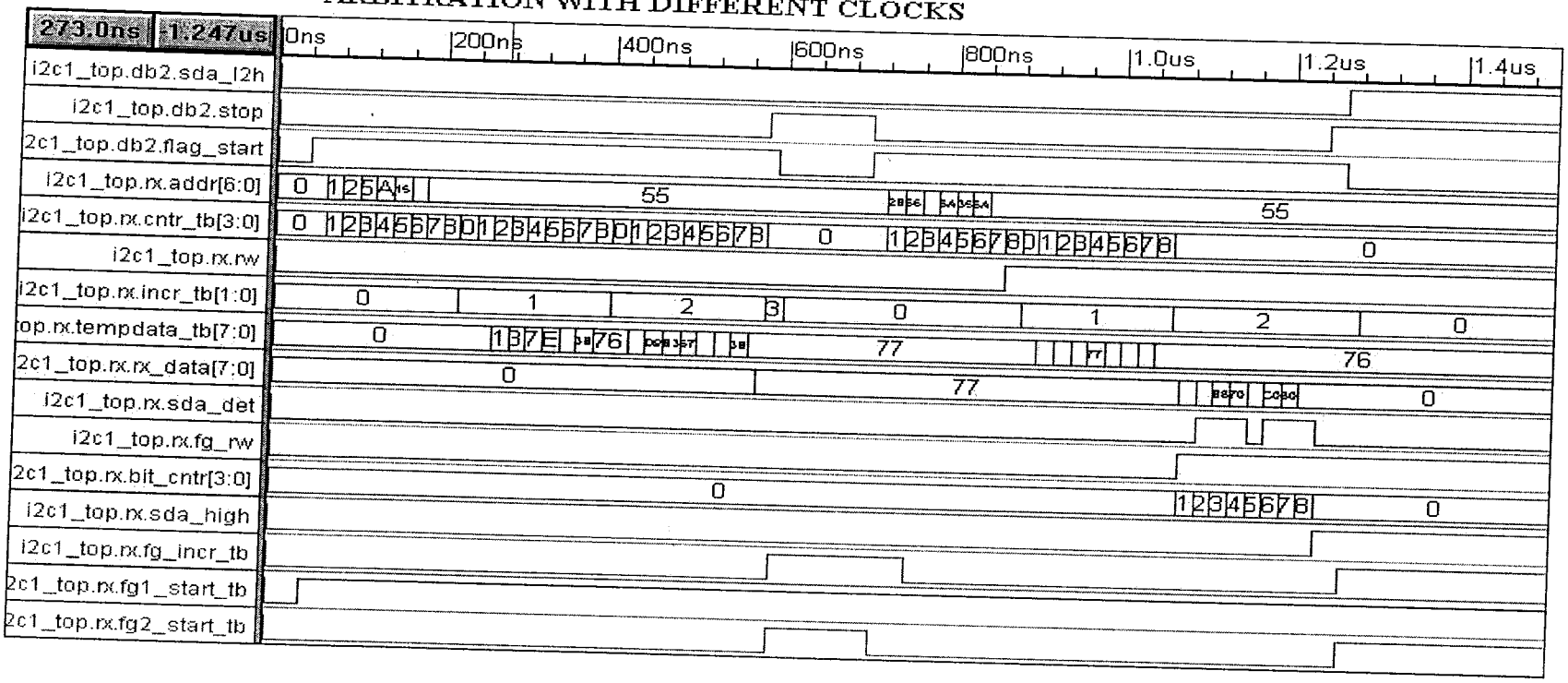
ARBITRATION WITH DIFFERENT CLOCKS



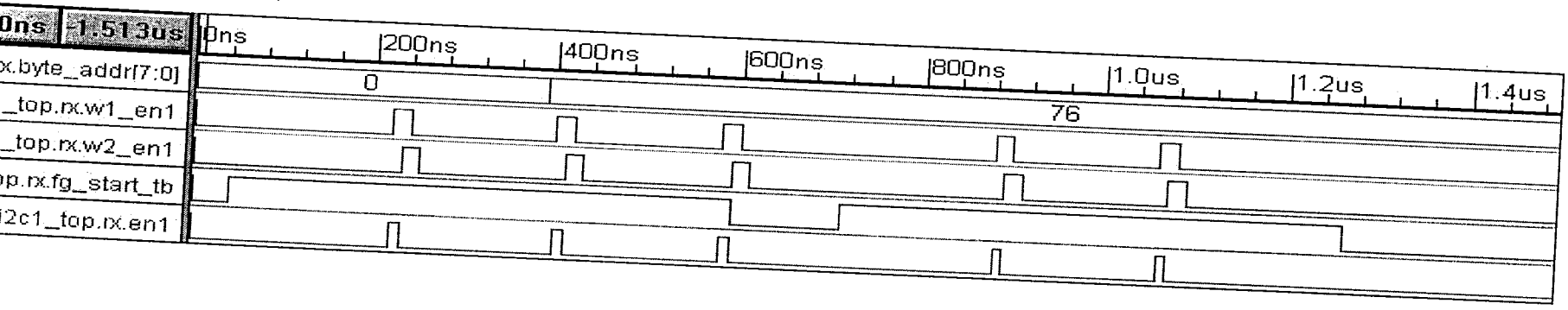
ARBITRATION WITH DIFFERENT CLOCKS

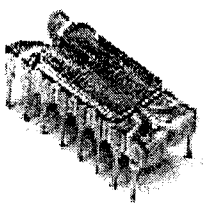


ARBITRATION WITH DIFFERENT CLOCKS



ARBITRATION WITH DIFFERENT CLOCKS





SYNTHESIS OUTPUT

CHAPTER-10

10. SYNTHESIS OUTPUT:

```
*********************************************************

Cell: i2c1_top   View: INTERFACE   Library: work

*********************************************************
Total accumulated area :

Number of ports                  : 5
Number of nets                   : 205
Number of instances              : 204
Number of references to this view : 0

        Cell      Library  References    Total Area

       AND2     PRIMITIVES   58 x 1     58 AND2
     DFFERS     PRIMITIVES   28 x 1     28 DFFERS
      FALSE     PRIMITIVES    1 x 1      1 FALSE
        INV     PRIMITIVES   46 x 1     46 INV
        MUX     PRIMITIVES   22 x 1     22 MUX
        OR2     PRIMITIVES   39 x 1     39 OR2
        TRI     PRIMITIVES    1 x 1      1 TRI
       TRUE     PRIMITIVES    1 x 1      1 TRUE
       XOR2     PRIMITIVES    7 x 1      7 XOR2
i2c1_rx_notri   work          1 x 7      7 XOR2
                                  1   1 DFFRS
                               2097   2097 DFFERS
                               4127   4127 MUX
                                857   857 OR2
                                 44   44 AND2
                                313   313 INV
                                  1   1 FALSE
                                  1   1 TRUE

*********************************************************

Cell: i2c1_rx_notri   View: INTERFACE   Library: work

*********************************************************
Total accumulated area :

Number of ports
```

Number of nets : 7453
Number of instances : 7448
Number of references to this view : 1

Cell	Library	References	Total Area
AND2	PRIMITIVES	44 x 1	44 AND2
DFFERS	PRIMITIVES	2097 x 1	2097 DFFERS
DFFRS	PRIMITIVES	1 x 1	1 DFFRS
FALSE	PRIMITIVES	1 x 1	1 FALSE
INV	PRIMITIVES	313 x 1	313 INV
MUX	PRIMITIVES	4127 x 1	4127 MUX
OR2	PRIMITIVES	857 x 1	857 OR2
TRUE	PRIMITIVES	1 x 1	1 TRUE
XOR2	PRIMITIVES	7 x 1	7 XOR2

Clock Frequency Report

Clock	: Frequency
scl	: Infinity

Slack Table at End Points

End points	Slack	Arrival		Required	
	rise	fall	rise	fall	
scl/	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix23/in	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix41/in	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix49/in	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix51/in[0]	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix51/in[1]	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix76/in[0]	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix76/in[1]	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix98/in	: 50.00	0.00	0.00	50.00	50.00
db_notri_ix100/in[0]	: 50.00	0.00	0.00	50.00	50.00

Critical Path Report

Critical path #1, (path slack = 50.0):

NAME	GATE	ARRIVAL	LOAD

```
---------------------------------------------------------------------------------
sda/
rx_notri/reg_fg2_start_tb/clk                0.00  0.00 up        3.10
data arrival time                            DFFERS    0.00  0.00 up        0.00
data required time  (default specified - setup time)          0.00
                                                             50.00
---------------------------------------------------------------------------------
data required time                                           50.00
data arrival time                                             0.00
                                                          ---------
slack                                                        50.00
---------------------------------------------------------------------------------
```

Critical path #2, (path slack = 50.0):

```
NAME                                         GATE          ARRIVAL          LOAD
---------------------------------------------------------------------------------
sda/
rx_notri/reg_fg_incr_tb/clk                  0.00  0.00 up        3.10
data arrival time                            DFFERS    0.00  0.00 up        0.00
data required time  (default specified - setup time)          0.00
                                                             50.00
---------------------------------------------------------------------------------
data required time                                           50.00
data arrival time                                             0.00
                                                          ---------
slack                                                        50.00
---------------------------------------------------------------------------------
```

Critical path #3, (path slack = 50.0):

```
NAME                                         GATE          ARRIVAL          LOAD
---------------------------------------------------------------------------------
sda/
rx_notri/reg_addr(0)/in                      0.00  0.00 up        3.10
data arrival time                            DFFERS    0.00  0.00 up        0.00
data required time  (default specified - setup time)          0.00
                                                             50.00
---------------------------------------------------------------------------------
data required time                                           50.00
data arrival time                                             0.00
                                                          ---------
slack                                                        50.00
---------------------------------------------------------------------------------
```

Critical path #4, (path slack = 50.0):

```
NAME                                         GATE          ARRIVAL          LOAD
```

```
---------------------------------------------------------------------------
sda/
rx_notri/reg_rw/in                    0.00  0.00 up          3.10
data arrival time                     DFFERS     0.00  0.00 up          0.00
data required time  (default specified - setup time)
                                                0.00
                                               50.00
---------------------------------------------------------------------------
data required time                             50.00
data arrival time                               0.00
                                              ---------
slack                                          50.00
---------------------------------------------------------------------------
```

Critical path #5, (path slack = 50.0):

```
NAME                                  GATE          ARRIVAL          LOAD
---------------------------------------------------------------------------
sda/
db_notri_reg_flag2_start/clk          0.00  0.00 up          3.10
data arrival time                     DFFERS     0.00  0.00 up          0.00
data required time  (default specified - setup time)
                                                0.00
                                               50.00
---------------------------------------------------------------------------
data required time                             50.00
data arrival time                               0.00
                                              ---------
slack                                          50.00
---------------------------------------------------------------------------
```

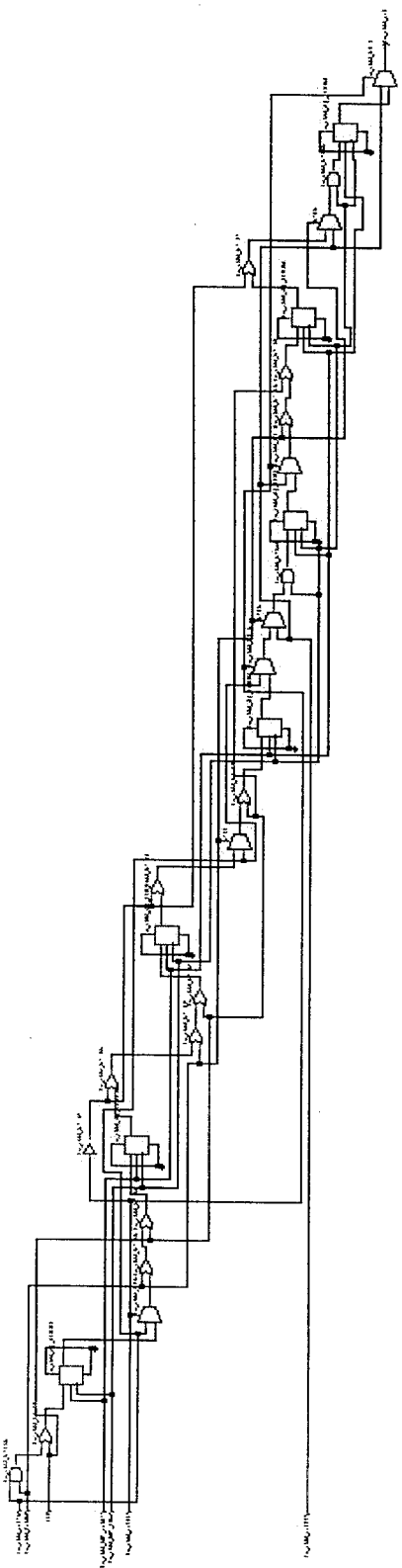
Critical path #6, (path slack = 50.0):

```
NAME                                  GATE          ARRIVAL          LOAD
---------------------------------------------------------------------------
sda/
db_notri_ix906/in[1]                  0.00  0.00 up          3.10
data arrival time                     AND2       0.00  0.00 up          0.00
data required time  (default specified - setup time)
                                                0.00
                                               50.00
---------------------------------------------------------------------------
data required time                             50.00
data arrival time                               0.00
                                              ---------
slack                                          50.00
---------------------------------------------------------------------------
```

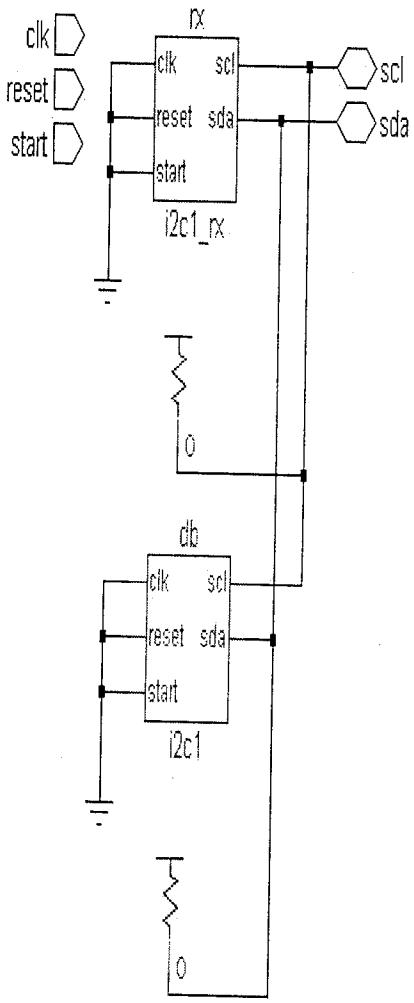
Critical path #7, (path slack = 50.0):

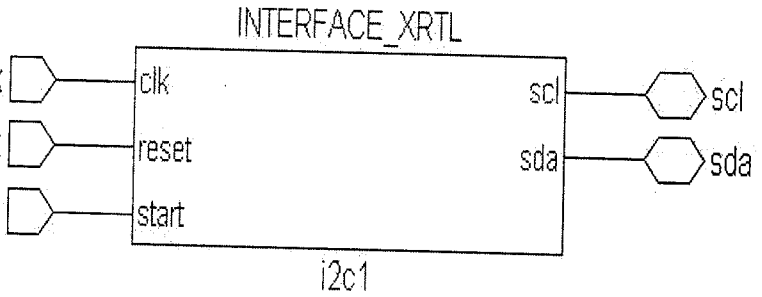
```
NAME                                  GATE          ARRIVAL          LOAD
---------------------------------------------------------------------------
```

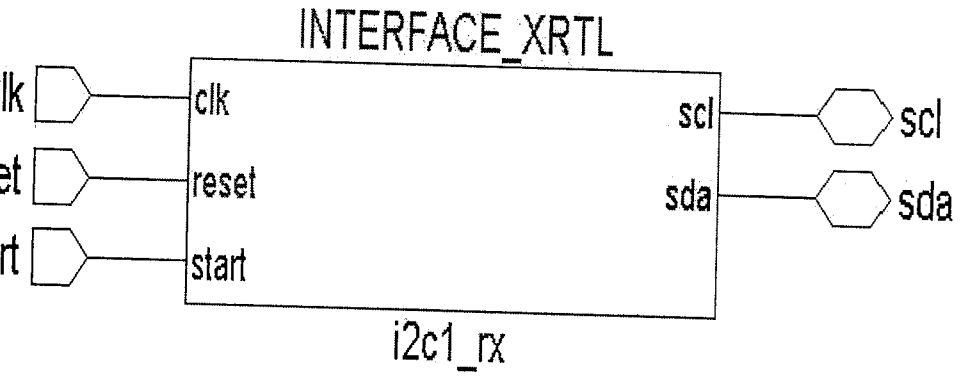
```
sda/
db_notri_ix533/in[1]                    0.00  0.00 up          3.10
data arrival time                       AND2       0.00  0.00 up          0.00
                                                   0.00
data required time  (default specified - setup time)                50.00
-------------------------------------------------------------------------------

data required time                                 50.00
data arrival time                                  0.00
                                                 ---------
slack                                              50.00
-------------------------------------------------------------------------------
```



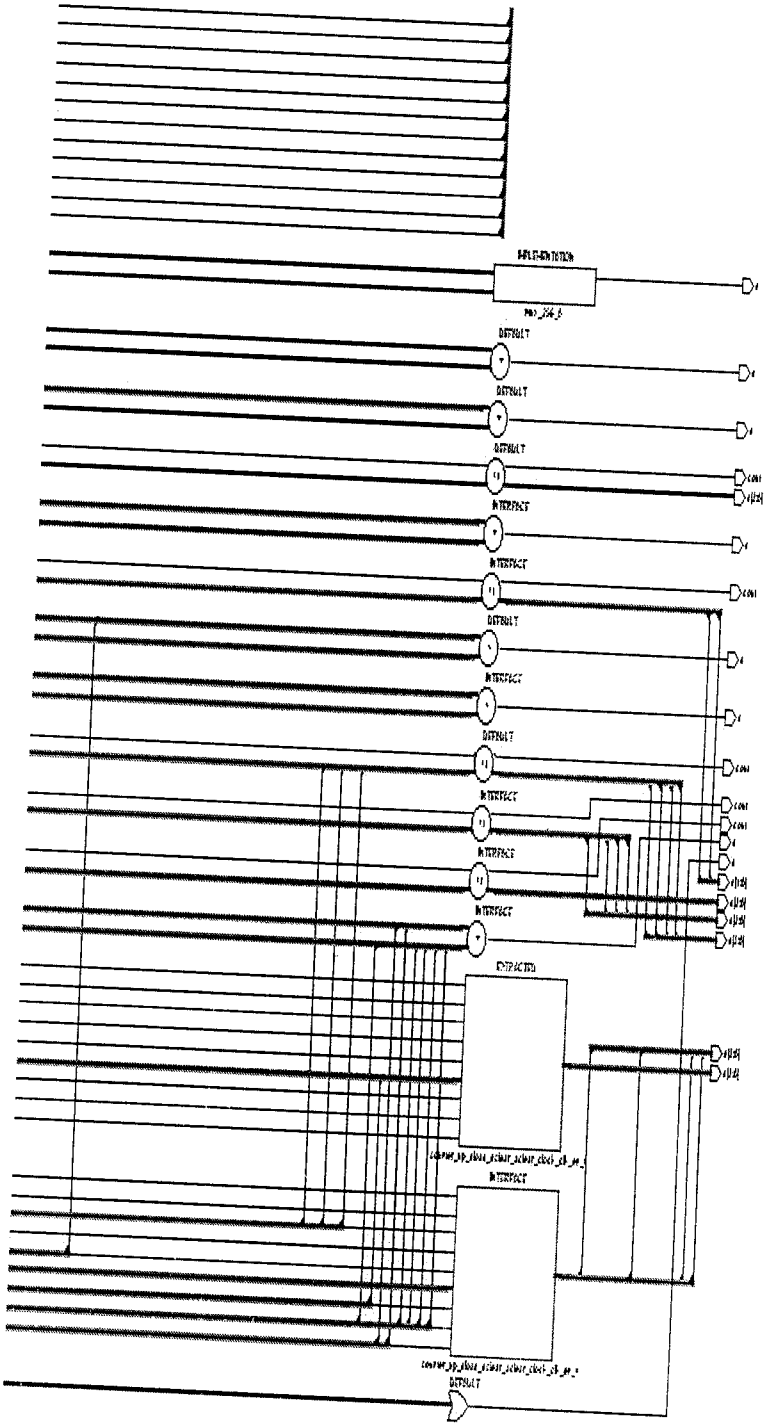
RTL SCHEMATIC



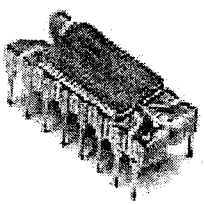

INTERFACE_XRTL
clk
reset
start
scl
sda
scl
sda
i2c1


INTERFACE_XRTL
clk
reset
start
scl
sda
scl
sda
i2c1_rx

OPERATOR SCHEMATIC-OUTPUT STAGE



CHAPTER-11



APPENDIX

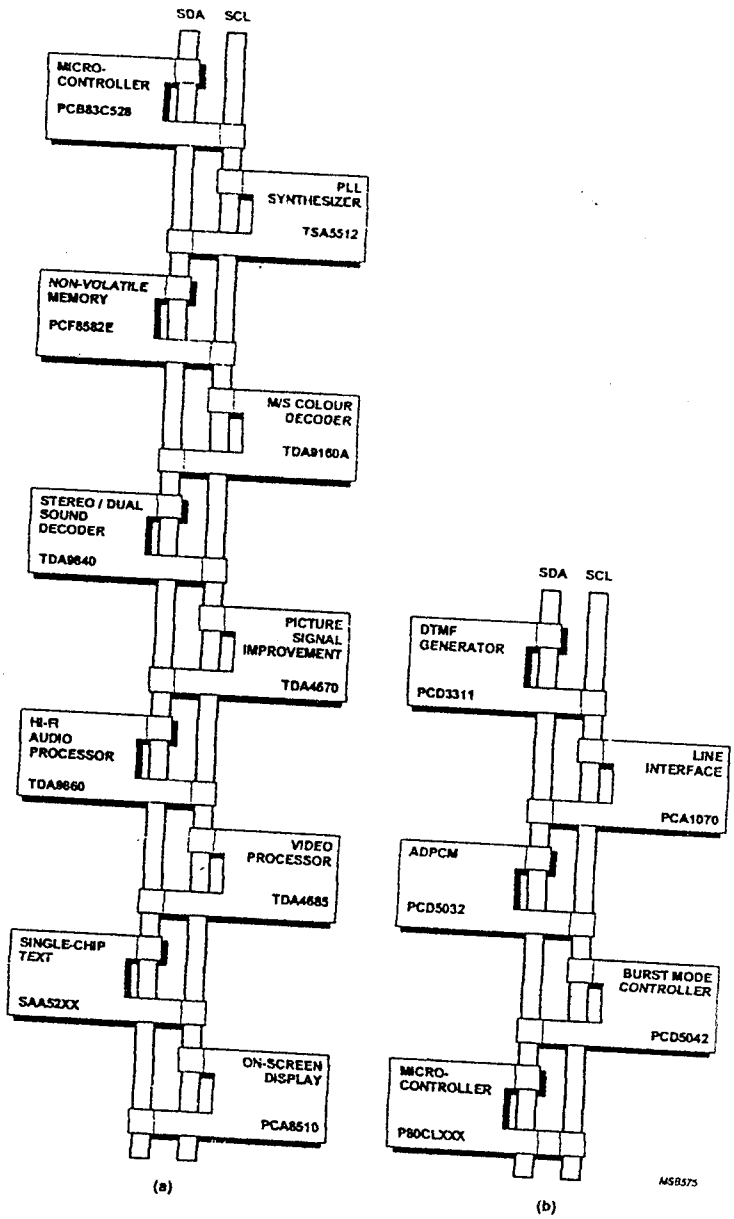


Fig.1 Two examples of I²C-bus applications: (a) a high performance highly-integrated TV set (b) DECT cordless phone base-station.

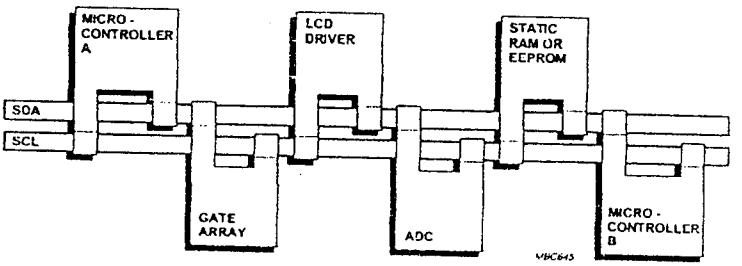


Fig.2 Example of an I²C-bus configuration using two microcontrollers.

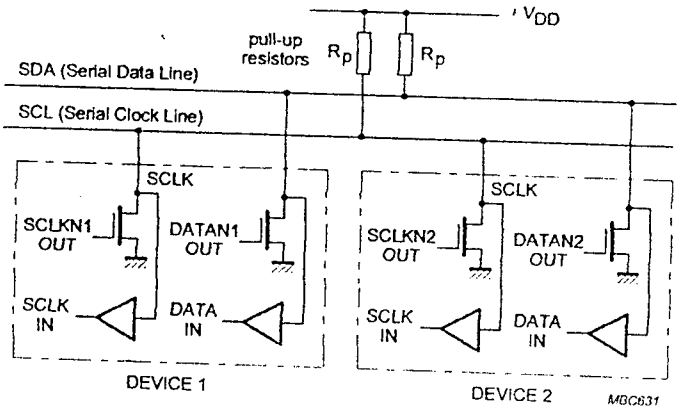


Fig.3 Connection of Standard- and Fast-mode devices to the I²C-bus.

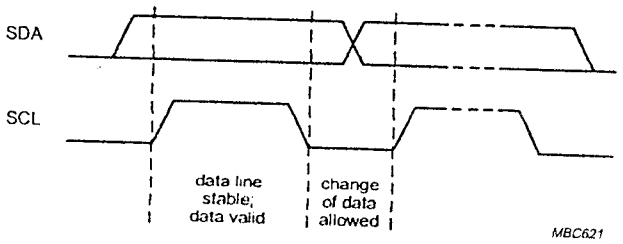


Fig.4 Bit transfer on the I²C-bus.

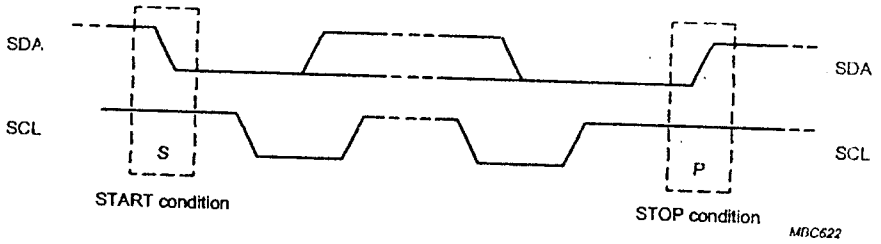


Fig.5 START and STOP conditions.

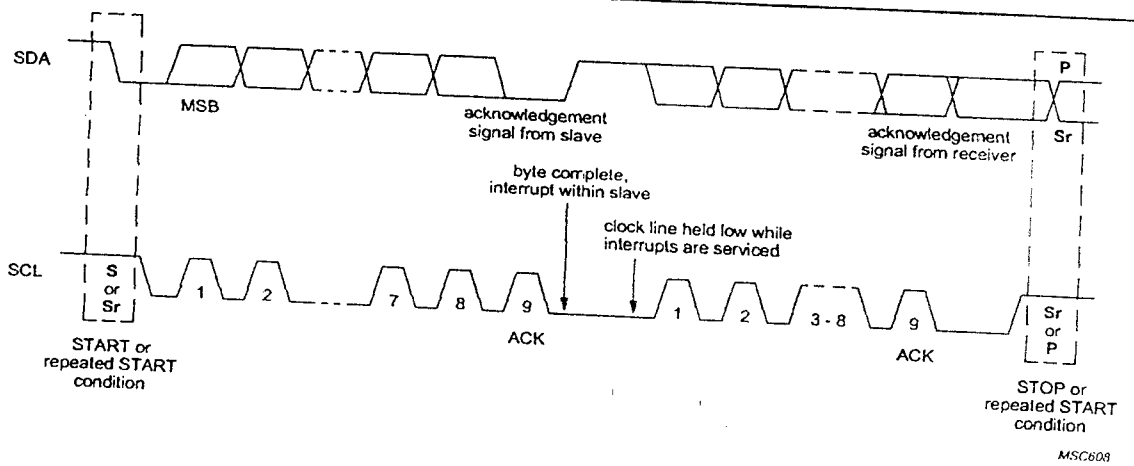


Fig.6 Data transfer on the I²C-bus.

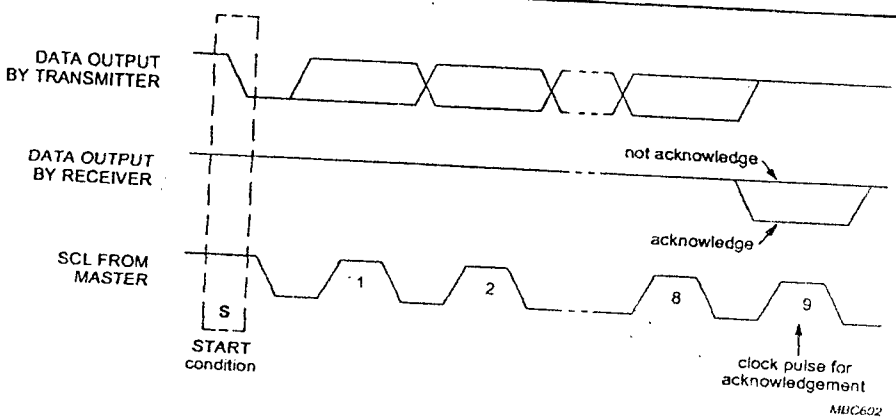


Fig.7 Acknowledge on the I²C-bus.

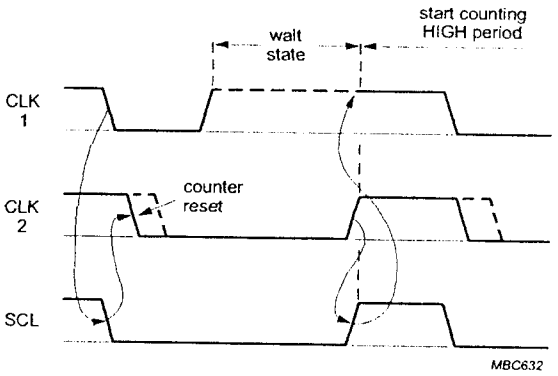


Fig.8 Clock synchronization during the arbitration procedure.

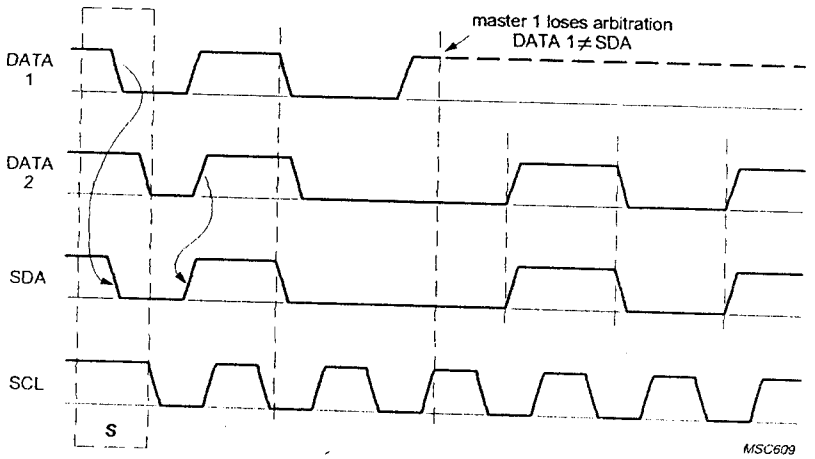
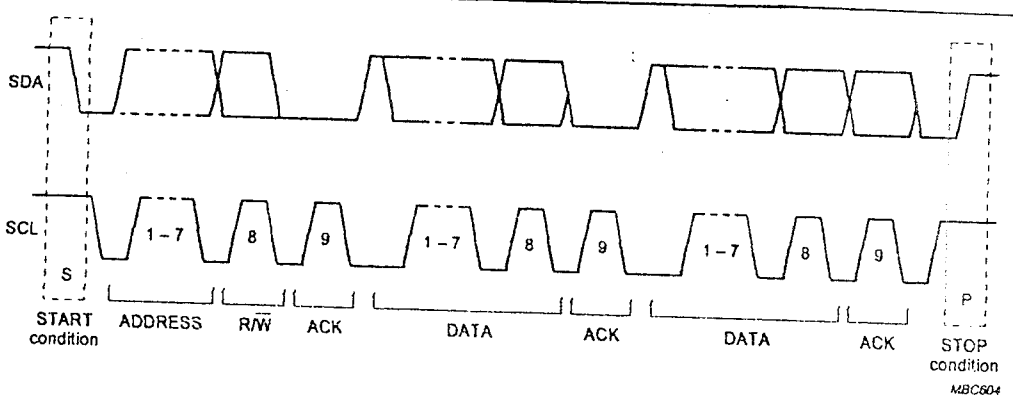


Fig.9 Arbitration procedure of two masters.



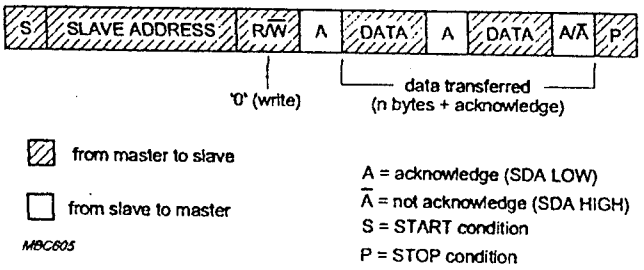


Fig.11 A master-transmitter addressing a slave receiver with a 7-bit address. The transfer direction is not changed.

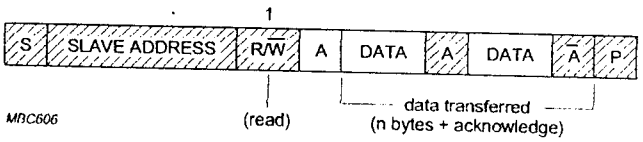


Fig.12 A master reads a slave immediately after the first byte.

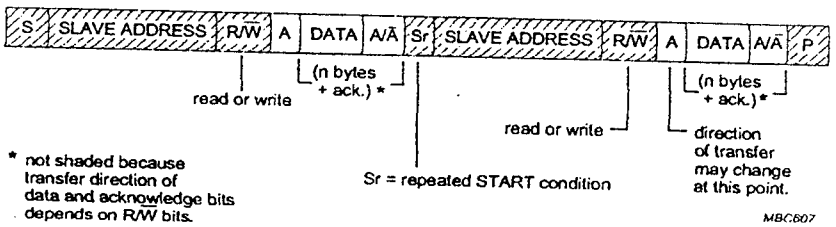


Fig.13 Combined format.

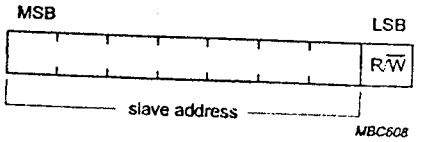


Fig.14 The first byte after the START procedure.

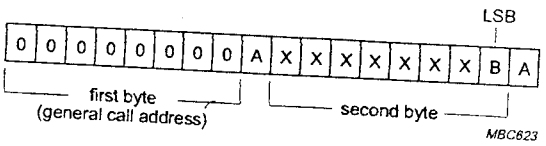


Fig.15 General call address format.

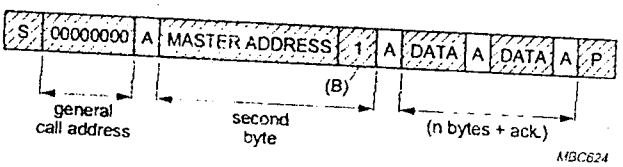


Fig.16 Data transfer from a hardware master-transmitter.

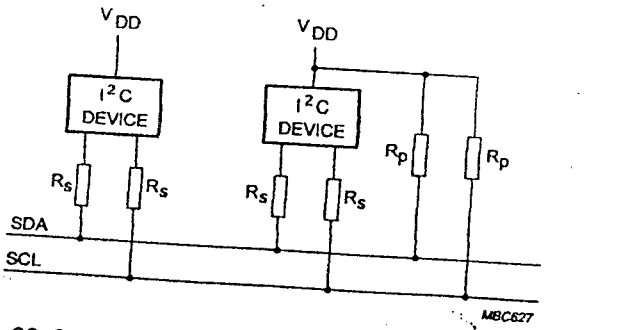


Fig.36 Series resistors (R_s) for protection against high-voltage spikes.

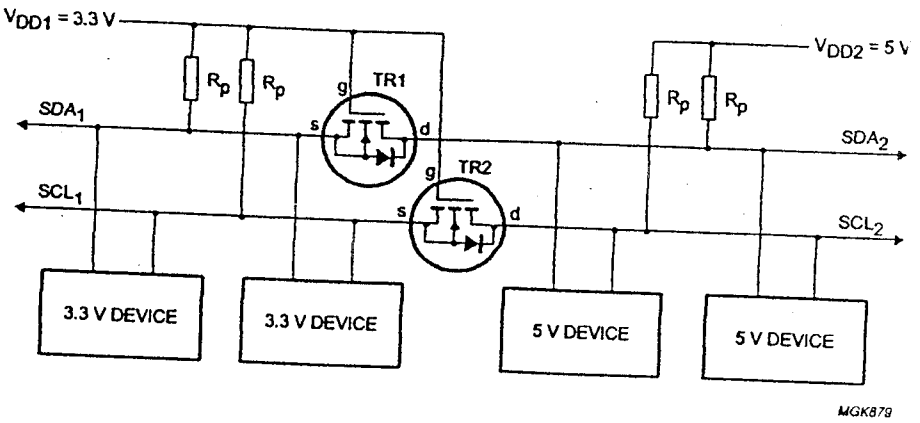
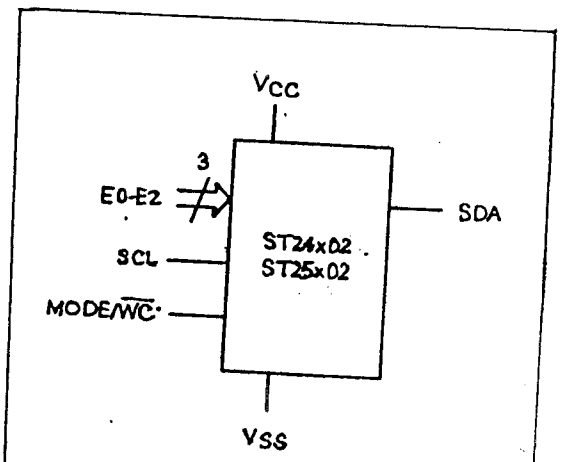


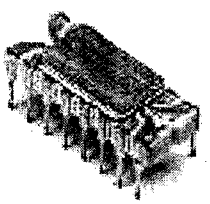
Fig.45 Bi-directional level shifter circuit connecting two different voltage sections in an I²C-bus system.

Table 2 Definition of bits in the first byte

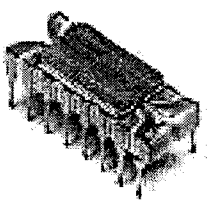
SLAVE ADDRESS	R/$\overline{W}$ BIT	DESCRIPTION
0000 000	0	General call address
0000 000	1	START byte[1]
0000 001	X	CBUS address[2]
0000 010	X	Reserved for different bus format[3]
0000 011	X	Reserved for future purposes
0000 1XX	X	Hs-mode master code
1111 1XX	X	Reserved for future purposes
1111 0XX	X	10-bit slave addressing

EEPROM LOGIC DIAGRAM (FIG 2.1)





CONCLUSION



BIBLIOGRAPHY

13. BIBLIOGRAPHY

1. INTRODUCTION TO VERILOG HDL

   - SAMIR PALNITKAR

2. VERILOG HDL PRIMER

   - BHASKAR

3. VERILOG HDL SYNTHESIS

   - BHASKAR

4. DIGITAL DESIGN

   - MORRIS MANO

5. FUNDAMENTALS OF LOGIC DESIGN

   - CHARLES .H. ROTH

6. DESIGN USING PIC MICRO CONTROLLER

   - JOHN PEATMAN