

**DIGITAL CIRCUIT SIMULATION SOFTWARE: DESIGN AND
IMPLEMENTATION
A DISSERTATION**

*Submitted in partial fulfillment of the
requirements for the award of the degree*

of

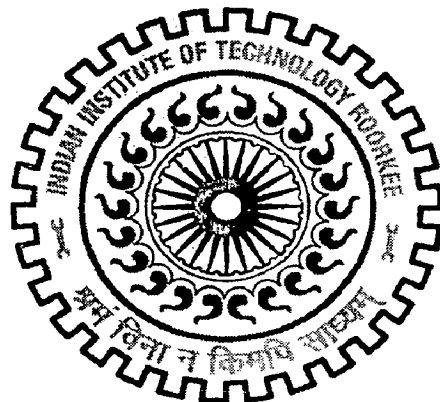
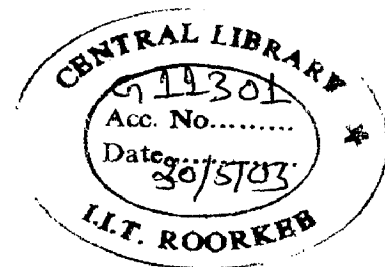
MASTER OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

By

MUKESH KUMAR



**IIT Roorkee – ER&DCI, Noida
C-56/1, “Anusandhan Bhawan”
Sector 62, Noida - 201 307**

FEBRUARY, 2003

ENROLLMENT NO. : 019025

621380285
MUK

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this dissertation titled “**DIGITAL CIRCUIT SIMULATION SOFTWARE: DESIGN AND IMPLEMENTATION**”, in partial fulfillment of the requirements for the award of the degree of **Master of Technology in Information Technology**, submitted in **IIT, Roorkee – ER&DCI Campus, Noida**, is an authentic record of my own work carried out during the period from August 2002 to February, 2003 under the guidance of **Dr. Poonam Rani Gupta**, Reader, Electronics Research and Development Centre of India, Noida.

The matter embodied in this dissertation has not been submitted by me for award of any other degree of diploma.

Date: 24th Feb 03

Place: Noida


(Mukesh Kumar)

CERTIFICATE

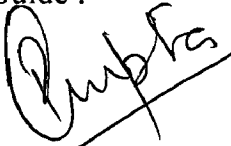
This is to certify that the above statement made by the candidate is correct to the best of my knowledge and belief.

Date: 24th Feb 03

Place: Noida

Guide :




(Dr. Poonam Rani Gupta)
Reader,
ER&DCI, Noida

M 101-7

CEBLIEICVLE

5/1/90

1/30/90


ACKNOWLEDGEMENT

The work presented in this report would not have been completed without the guidance and support of many people. In first place, I would like to thank **Prof. Prem Vrat**, Director, IIT Roorkee and **Sh. R.K.Verma**, Executive Director, ER&DCI, Noida.

I would also like to thank Prof. **A.K.Awasthi**, Dean PGS&R, IIT Roorkee. I would also like to thank Prof. **R.P.Agrawal**, Course Coordinator, M.Tech.(IT), IIT Roorkee and **Mr. V.N.Shukla**, Course Coordinator, M.Tech.(IT), ER&DCI, Noida.

I would like to thank my guide, **Dr. Poonam Rani Gupta**, for her constant support, incredible enthusiasm and encouragement. I am also grateful to Mr. Munish Kumar, Project Engineer, ER&DCI, Noida for the cooperation extended by him in the successful completion of this report.

Finally, I would like to extend my gratitude to all those persons who directly or indirectly helped me in the process and contributed towards this work.



(Mukesh Kumar)

Enroll. No. : 019025

CONTENTS

CANDIDATE'S DECLARATION	(i)
ACKNOWLEDGEMENT	(ii)
ABSTRACT	1
1. INTRODUCTION	3
1.1 Overview	3
1.2 Objective	4
1.3 Scope	4
1.4 Organization of Dissertation	5
2. LITERATURE SURVEY ON OBJECT ORIENTED MODELING AND SIMULATION	7
2.1 Object Oriented Approach	7
2.2 Object Oriented Methodology	9
2.2.1 Analysis	9
2.2.2 System Design	10
2.2.3 Object Design	11
2.2.4 Implementation	13
2.2.5 More about C++	13
2.3 Logic Gates	18
2.4 Digital Circuits	18
2.4.1 Asynchronous Logic	18
2.4.2 Combinational Circuits	20
2.4.3 Sequential Circuits	20
2.4.4 Time Dependence	21
2.5 Timing Diagrams	21
3. DESIGN AND IMPLEMENTATION	23
3.1 Logic Gates	23

3.2 Input/Output	23
3.3 Sequential Elements	23
3.3.1 Flip- Flop	23
3.3.2 Counter-4	26
3.3.3 Shift Register	26
3.4 Combinational Elements	27
3.4.1 4-bit full Adder	27
3.4.2 4-bit Comparator	28
3.4.3 8-input line Multiplexer	28
3.4.4 8- output line Demultiplexer	29
3.4.5 3-8 line Decoder	29
3.4.6 Octal to Binary Encoder	30
3.4.7 Data Selector	32
3.5 Macros	32
3.6 Source Code Description and Data Dictionary	33
3.7 Generated Files	44
3.7.1 *.CRC Files	44
3.7.2 *.MAC Files	44
3.8 Timing Diagrams	44
3.8.1 Generation of Timing Diagrams	45
4. RESULT AND DISCUSSION	47
REFERENCES	57
Appendix A	

ABSTRACT

Decoding the logic and speculating the output of any logical circuit has always been a task of great importance for the students learning digital circuits and circuit designers. This Project deals with the design and implementation of circuit simulation software that has an ability to simulate different types of combinational as well as sequential logical circuits. This software provides a tool for the students studying Electronics especially digital electronics to clearly understand the logical behaviour of any digital circuit. It also has a special feature of defining any circuit as a macro and adding it to the library, which is already defined. Now, this macro can be used several times within any complex digital circuit, thus reducing a huge amount of extra work and time. Moreover, This software also generates Timing waveforms for the different blocks in the circuit. With the help of those timing diagrams, one can better visualize the behaviour of any element within that circuit. Overall, It has a lot of features that makes the life easier for circuit design engineers and students studying digital electronics.

INTRODUCTION

1.1 Overview

This Software is designed to efficiently run digital circuit simulations. It is not a drafting program, i.e. it can't be used to draw nice circuit diagrams for digital circuits. On the other hand, it is a design program, which helps in designing own digital circuits and trying them out on the computer first.

This doesn't only cut down the time required for building a logical circuit; it also makes the testing process easy and straightforward. Because, if after testing the circuit on software and realizing it as hardware, if a bug occurred and the hardware circuit didn't work. Then the error is somewhere in the connections or in one of the ICs, but not in the logic, since the logic of the circuit is already been tested on the computer, it also provides facility to observe the Timing Diagrams (wave forms) of any part in the circuit. It also provides the feature of creating Macros. A Macro is simply a new IC, which the user builds from scratch. Macro could be added to library by simply drawing the circuit and assigning to it input pins and output pins. Usually large digital circuit projects are designed by the Divide and Conquer technique where they are broken down into smaller modules. These modules could be converted in to macros, and finally collected them in a single circuit.

Project shows the power of using OOP [1] in programming a modeling and simulation project. OOP stands for Object Oriented Programming. There is no one single clear definition of OOP, but the following definitions may help clarify conception of OOP. OOP is a new revolutionary technique of modeling software based on real-world objects, i.e. it is a programming style that mimics the way all of us get things done. OOP in itself is not a programming language; rather, it is a programming style that could be used with any of the programming languages present today, to simplify the software life cycle of any project.

Loosely speaking, OOP refers to a new way of organizing programs into objects that encapsulates data with a set of well-defined operations and that share

code with other objects in a predefined hierarchy by inheritance. Although OOP techniques can be implemented in any programming language (C being a good example), it is easier when the language has the features necessary to support objects. OOP is a direct response to the complexity of modern applications, complexity that has often made many programmers throw up their hands in despair. Inheritance and encapsulation are extremely effective means for managing complexity. Far more than structured programming, object orientation imposes a rational order on software structures that, like a taxonomy chart, imposes order without imposing limits. Add to that the promise of the extendibility and reusability of existing code, and the whole thing begins to sound almost too good to be true.

1.2 Objective

The Problem includes the design and Implementation of circuit simulation software capable of simulating different type of combinational and sequential circuits.

1.3 Scope

This Software is a very easy to use, but still very powerful, Digital Circuit Simulation software. The students studying electronics, especially those studying digital circuits, can utilize it.

Some of the major features of this software are:

- ✓ An easy to use interactive menu driven User Interface.
- ✓ An easy to draw method, with an intelligent drawing mechanism.
- ✓ A predefined library of 37 digital elements.
- ✓ The ability of the user to create new digital elements known as Macros.
- ✓ Combinational, as well as Sequential, logical circuits support.
- ✓ Feedback Simulation and compensation.
- ✓ More than one circuit could draw and tested on the same board.

- ✓ Change parameters of elements while running.
- ✓ Create Timing Diagrams.
- ✓ Run with or without Timing Diagrams.
- ✓ The capability to open more than one circuit simultaneously.
- ✓ On-Line circuit save and load
- ✓ Detection of trivial circuit errors.

1.5 Organization of Dissertation

Chapter 1 gives the Overview, Objective and the scope of the project. Chapter 2 gives the literature survey, which discusses the basic fundamental terms and approach used. Chapter 3 briefly presents the design and implementation part. Chapter 4 gives the results and discussion.

LITERATURE SURVEY ON OBJECT ORIENTED MODELING AND SIMULATION

Object Oriented modeling and design is a new way of thinking about problems using models organized around real world concepts. The Fundamental construct is the object, which combines both data and behavior in a single entity. First an analysis model is built to abstract essential aspects of the application domain without regard for eventual implementation. This model contains objects found in application domain, including a description of properties of the objects and their behavior. Then design decisions are made and details are added to the model to describe and optimize the implementation. The application domain objects form the framework of the design model, but they are implemented in a programming language, database or hardware.

2.1 Object Oriented Approach

The term “ Object Oriented [1] ” means that software is organized as a collection of discrete objects that incorporate both data structure and behavior. This is in contrast to conventional programming in which data structure and behavior are only loosely connected. Object Oriented generally include four aspects: Identity, Classification, Polymorphism and Inheritance.

Identity means that data is quantized into discrete, distinguishable entities called objects. A paragraph in a document, a window on my workstation, and the white queen in a chess game are examples of objects. Objects can be concrete, such as file in a file system, or conceptual, such as scheduling policy in a multiprocessing operating system. Each object has its own inherent identity. In other words, two objects are distinct even if all there attribute values (such as name and size) are identical. In a real world, an object simple exists, but within a programming language each object has a unique handle by which t can be uniquely referenced. The handle may be implemented in various ways, such as an address, array index,

or unique value of an attribute. Object references are uniform and independent of the contents of the objects, permitting mixed collections of objects to be created, such as file system directory that contains both files and sub directories.

Classification means that objects with the same data structure (attributes) and behavior (operations) are grouped into a class. A Class is an abstraction that describes properties important to an application and ignores the rest. Any choice of classes is arbitrary and depends on application. Each class describes a possibly infinite set of individual objects. Each object is said to be an instance of its class. Each instance of the class has its own value for each attribute but shares the attribute names and operations with other instance objects. An object contains an implicit reference to its own class; it “knows what kind of thing it is.”

Polymorphism means that the same operation may behave differently on different classes. The move operation, for example, may behave differently on the window and chess piece classes. An operation is an action or transformation that an object performs or is subject to. A specific implementation of an operation by a certain class is called a method. Since an object-oriented operator is polymorphic, it may have more than one method implementing it. In the real world an operation is simply an abstraction of an analogous behavior across different kind of objects. Each object “ knows how” to perform its own operations. In an object oriented programming language, however, the language automatically selects the correct method to implement an operation based on the name of the operation and the class of the object being operated upon. The user of an operation needs not to be aware of how many methods exist to implement a given polymorphic operation. New classes can be added with out changing existing code, provided methods are provided for each applicable operation on the new classes.

Inheritance is sharing of attributes and operations among classes based on the hierarchy relationship. A class can be defined broadly and then refined into successively finer subclasses. Each subclass incorporates, or inherits, all of the properties of its super class and its own unique properties. The properties of the super class need not to be repeated in each subclass .for example; scrolling window and fixed window are subclasses of window. Both subclasses inherit the

properties of class window, such as a visible region on the screen. Scrolling window adds a scroll bar and an offset. The ability to factor out common properties of several classes into a common super class and to inherit the properties from the super class can greatly reduce repetition within designs and programs and is one of the main advantages of an object-oriented system.

2.2 Object Oriented Methodology

The Methodology [1] consists of building a model of an application domain and then adding implementation details to it during the design of the system. The methodology has the following stages:

2.2.1 Analysis

Starting from a statement of the problem, the model of the real-world situation showing its important properties is build. The analysis model is concise, precise abstraction of what the desired system must do, not how it will be done. The objects in the model should be application domain concepts and not the computer implementation such as data structures. The analysis model should not contain any implementation decisions. The goal of analysis is to develop a model of what the system will do. The model is expressed in terms of objects and relationships, dynamic control flow, and functional transformations. The process of capturing requirements and consulting with the requestor should continue throughout analysis.

1. Write or obtain initial description of the problem.
2. Build an Object Model:
 - Identify object classes
 - Begin a data dictionary containing descriptions of classes, attributes, and associations
 - Add associations between classes.
 - Add attributes for objects and links.
 - Organize and simplify object classes using inheritance.

- Test access paths using scenarios and iterate the above steps as necessary
- Group classes into modules, based on close coupling and related function.

⇒ Object Model = object model diagram + data dictionary.

3. Construct a Functional Model:

- Identify input and output values.
- Use data flow diagrams as needed to show functional dependencies.
- Describe what each function does.
- Identify constraints.
- Specify optimization criteria.

⇒ Functional Model = data flow diagrams + constraints.

4. Verify, iterate, and refine the models:

- Add key operations that were discovered during preparation of the functional model to the object model. Do not show all operations during analysis as this would clutter the objects model; just show the most important operations.
- Verify that classes, associations, attributes, and operations are consistent and complete at the chosen level of abstraction.
- Develop more detailed scenarios (including error conditions) as variations on the basic scenarios
- Iterate the above steps as needed to complete the analysis.

2.2.2 System Design

System design is the first basic stage in which the basic approach to solving the problem is selected. During system design, the target system is organized into sub systems based on both the analysis structure and the proposed architecture. The architecture provides the context in which more detailed decisions are made

in later design stages. Figure 2.2 shows the Functional model of circuit simulation software

Following decisions are to be made after system design:

- ✓ Organize the system into subsystems.
- ✓ Identify concurrency inherent in the problem.
- ✓ Allocate subsystems to processors and tasks.
- ✓ Choose an approach for management of data stores.
- ✓ Handle access to global resources.
- ✓ Choose the implementation of control in software.
- ✓ Handle boundary conditions.
- ✓ Set trade off priorities.

2.2.3 Object Design

Object design phase determines the full definition of the classes and associations used in the implementation, as well as the interfaces and algorithms to implement operations. Figure 2.1 shows the Object Model of the circuit simulation. The object design phase adds internal objects for implementation and optimizes data structures and algorithms. The objects discovered during analysis serve as a skeleton of the design. In particular, the operations identified during analysis must be expressed as algorithms, with complex operations decomposed into simpler internal operations. The classes, attributes and associations from analysis must be implemented as specific data structures. New objects classes must be introduced to store intermediate results during program execution and to avoid the need for recomputation.

During object design, following steps are to be carried out:

- ✓ Combine the models to obtain operations on classes.
 - Find an operation for each process in the functional model.

- Define an operation for each event in the dynamic model, depending on the implementation of control.
- ✓ Design algorithms to implement operations
 - Choose algorithms that minimize the cost of implementing operations.
 - Select data structures appropriate to the algorithms.
 - Define new internal classes and operations as necessary.
 - Assign responsibility for operations that are not clearly associated with a single class.
- ✓ Optimize access paths to data
 - Add redundant associations to minimize access cost and maximize convenience.
 - Rearrange the computation for greater efficiency.
 - Save derived values to avoid recomputation of complicated expressions.
- ✓ Implement software control for external interactions.
- ✓ Adjust class structure to increase inheritance.
 - Rearrange and adjust classes and operations to increase inheritance.
 - Abstract common behavior out of groups of classes.
 - Use delegation to share behavior where inheritance is semantically invalid.
- ✓ Design implementation of associations
 - Analyze the traversal of associations.
 - Implement each association as a distinct object or by adding object-valued attributes to one or both classes in the associations.
- ✓ Determine the exact representation of Object attributes.
- ✓ Package classes and associations into modules

2.2.4 Implementation

The Object classes and relationship developed during object design are finally translated into a particular language, database, or hardware implementation. For the translation, language C++ is chosen because of its inherent property to support Objects Oriented Programming. During implementation, Good software engineering practice is followed to ensure that trace ability to the design is straightforward and so that implemented system remains flexible and extensible.

Implementations details and System Requirements :

Platform : Windows 98/2000

Language : C++

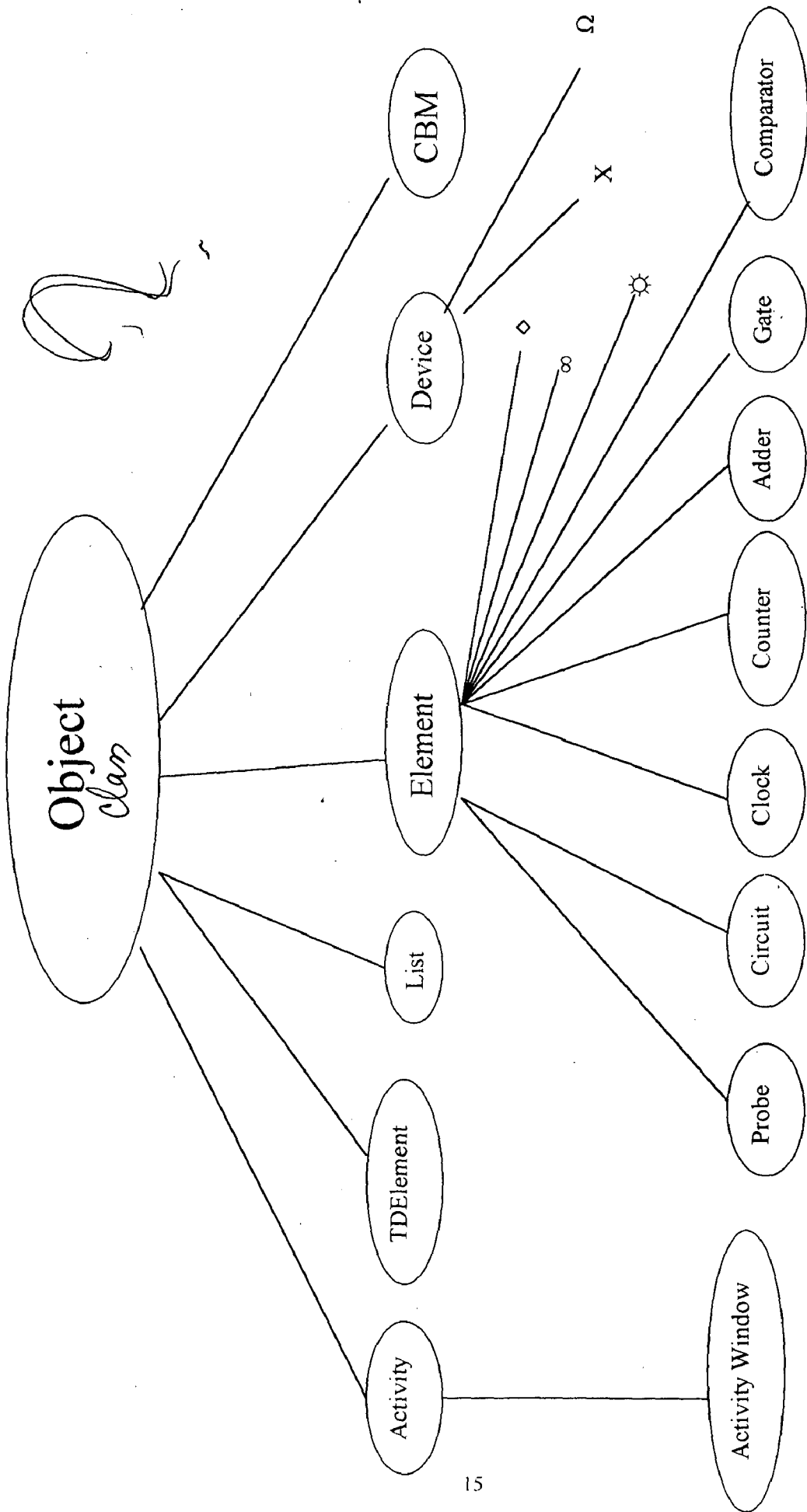
The necessary system requirements are:

- IBM PC, XT, AT, or PS/2 (all models) or compatible.
- EGA, VGA, or MCGA display card or compatible,
- One 360K or higher diskette drive, or a fixed disk drive.

2.2.5 More about C++

C++ is a strongly typed language developed by Bjarne Stroustrup at AT&T Bell laboratories. It was originally implemented as preprocessor that translates C++ into standard C. As a preprocessor, C++ introduced problems for symbolic debuggers, but direct compilers are now available, and symbolic debuggers that support objects with inheritance and dynamic binding are now available. C++ is a hybrid language, in which some entities are objects and some are not. C++ is an extension of the C language, implemented not only to add object-oriented capabilities but also to redress some of the weakness of the C language. Many added features are orthogonal to object oriented programming, such as inline expansion of subroutines, overloading of functions, and function prototypes. It is also designed to support OOP through its class data type, the concept of overloading functions, operators and virtual functions. The class type with overloaded functions and operators support Data Abstraction and Inheritance, the virtual functions provide the mechanism for implementing Polymorphism. C++ is one of the best languages (at the present time) that support OOP to its full extent.

C++ offers the programmer the base for realizing all his dreams through one of the best-structured languages ever devised. The project itself is a Digital Circuit Simulator that simplifies the design and testing of any small scale to medium scale hardware project. Thus it is very suitable for emphasizing the powers of OOP, where an AND-gate is realized as AND-object within the conceptual supreme data abstraction programming level of OOP.



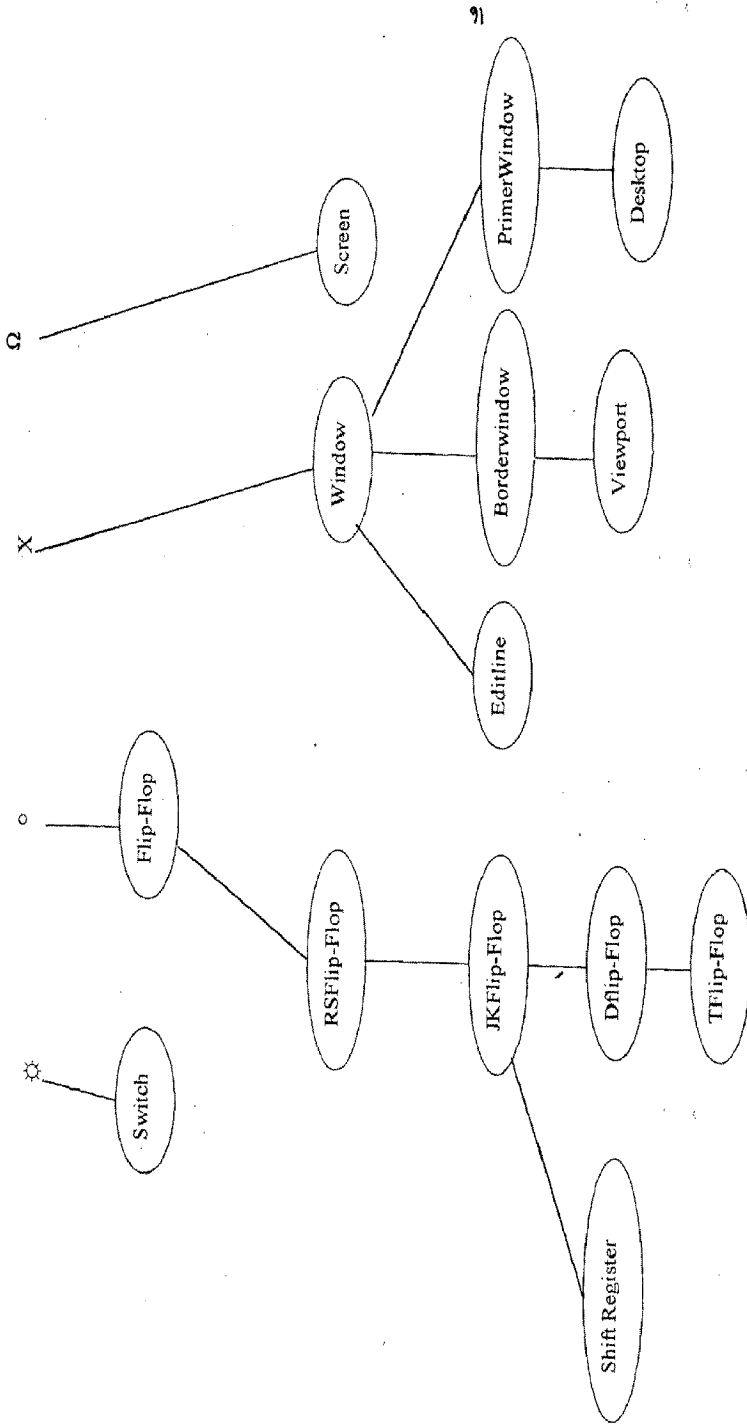


Figure 2.1 : Object Model Of Circuit Simulation Software

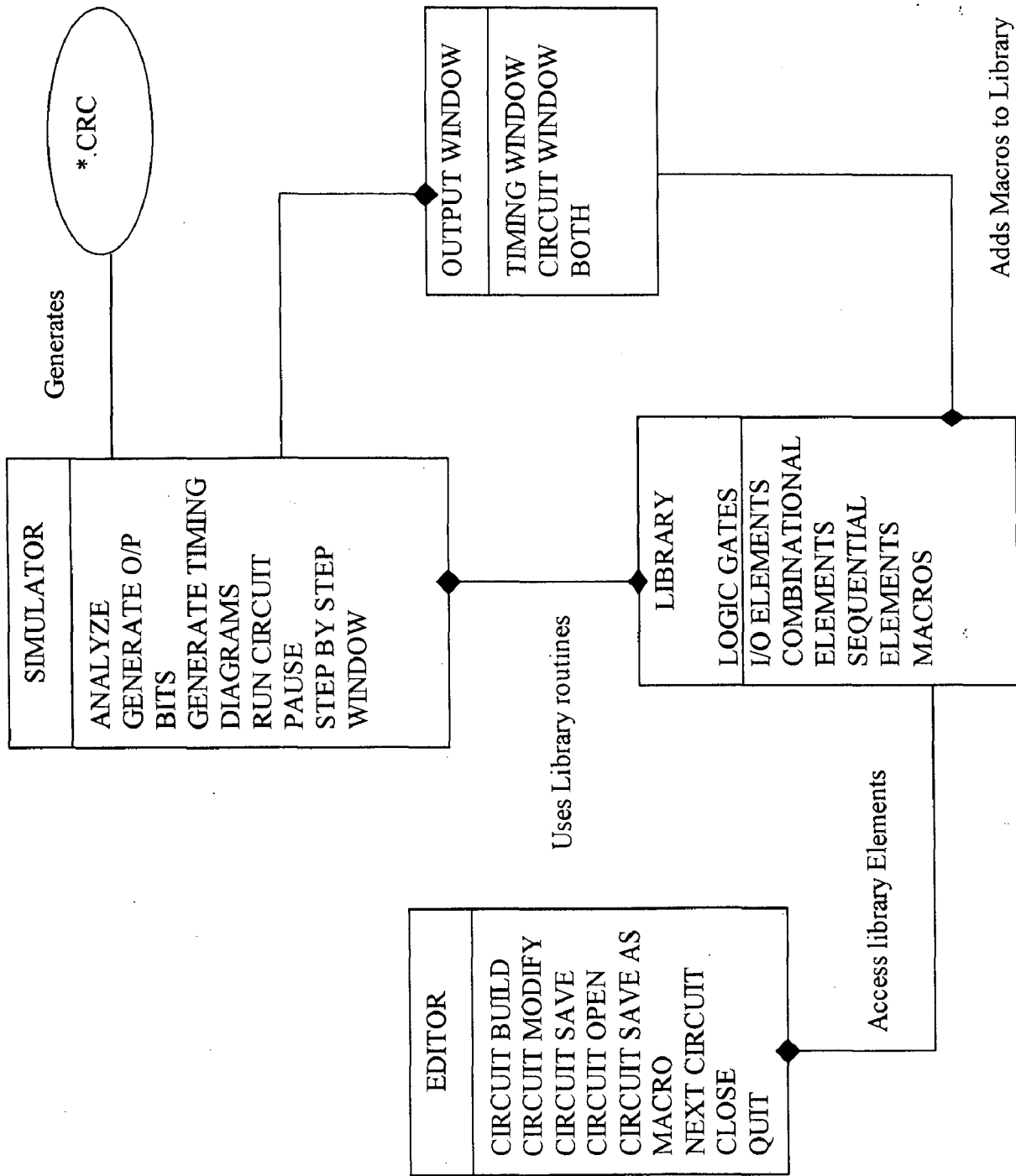


Figure 2.2.: Functional Model of circuit simulation software

2.3 Logic Gates

Logic gates [2] are block of hardware that produces a logic-1 or logic-0 output signal if input logic requirements are satisfied. There are 8 basic logic gates. Figure2.3 shows all the different types of logic gates with their truth tables.

2.4 Digital circuits

Digital circuits [2] are collections of devices that perform logical operations on two logical states, represented by voltage levels. Standard operations such as AND, OR, INVERT, EQUIVALENT, etc. are performed by devices known as gates. Groups of compatible gates can be combined to make yes/no decisions based on the states of the inputs. For example, a simple warning light circuit might check several switch settings and produce a single yes/no output. More complicated circuits can be used to manipulate information in the form of decimal digits, alphanumeric characters, or groups of yes/no inputs.

2.4.1 Asynchronous Logic

Suppose There is a statement which can be true or false, perhaps representing the Presence or absence of a particle, a light signal on or off, a voltage present or absent, or any other binary possibility. For now, the physical meaning of the statement can be ignored and it can be asked how one would decide the logical truth or falsehood of combinations of such statements, a subject called combinational logic. If higher truth-value of a statement or a variable is denoted by 1 and lower value by 0, it is called positive logic. But, if higher value is denoted by 0 and lower value by 1, it is called negative logic. These basic combinations, or similar ones, have been implemented in electronic circuitry, where truth-values can be represented by different voltage levels. By combining the basic operations, other complicated logical functions could be constructed.

There are two types of digital circuits, which are explained in the later portion.

1. Combinational Digital Circuit

2. Sequential digital Circuit

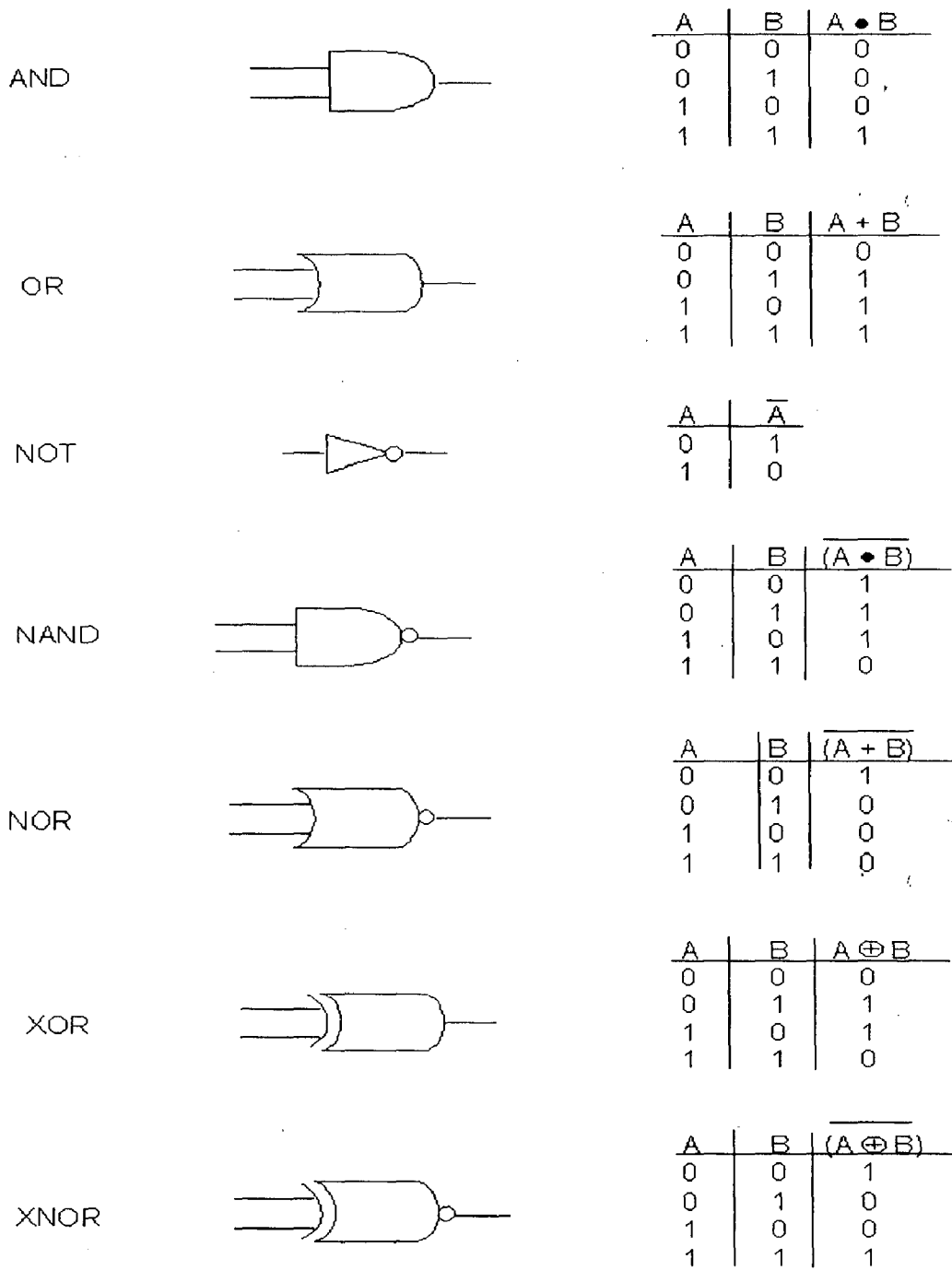


Figure2.3: Logic gates with their Truth Tables

2.4.2 Combinational Circuits

A Combinational circuit consists of input variables, logic gates and output variables. The Logic gate accepts signals from the inputs and generates signals to the outputs. This process transforms binary information from the given input data to the required output data. Obviously, both input and output data are represented by binary signals, i.e., they exist in two possible values, logic-0 and logic-1. Output is solely dependent over the type of input present at that time. There is no feedback present in these types of the circuit. Figure 2.4 shows its block diagram.

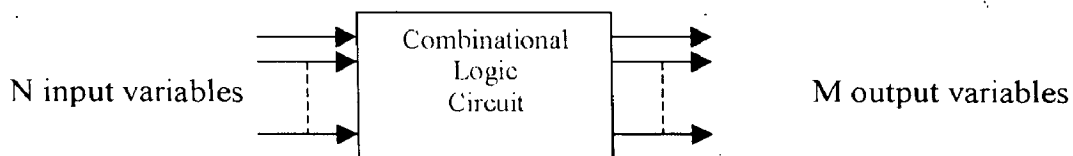


Figure 2.4: Block diagram of a combinational circuit.

2.4.3 Sequential Circuits

They consist of a combinational circuit to which memory elements are connected to form a feedback path. The Memory Elements are the devices capable of storing binary information within them. The binary information stored in the memory elements at any given time defines the state of the sequential circuit. The sequential circuit receives binary information from external inputs. These inputs together with the present state of the memory elements, determine the binary value at the output terminals. They also determine the condition for changing the state in the memory elements. The block diagram in Figure 2.5 demonstrates that the external output in the sequential circuit is a function of external inputs as well as present state of the memory element. The next state of the memory elements is also a function of external inputs and present state.

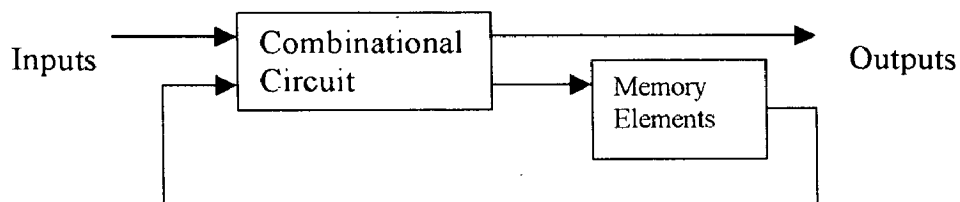


Figure 2.5: Block Diagram of a sequential Circuit.

2.4.4 Time dependence

In certain systems timing may become critical. Gates require a finite amount of time to change their output in response to a change in the input signals (gate delay). In a complicated circuit it may happen that the inputs to a particular gate have been processed through different numbers of preceding stages, and may not arrive at the same time. This will cause the last gate to produce an electrically correct but logically wrong output [3], at least transiently. In situations where this causes problems it can be cured either by accurate matching of the signal delays, or by clocking. The matching approach is used where the logic must handle events in "real time", as required in a particle-counting experiment. The method is to add delay as needed to insure that all possible signals require the same amount of time to propagate through each stage in the system. Timed logic circuits are sometimes called "combinational" or "asynchronous" logic, since they produce an output as quickly as possible after a change in input. Such circuits are obviously very difficult to adjust if they are at all complex. The alternative to synchronous logic is "synchronous" or "clocked" logic. In this scheme an additional input, the clock, is provided at each logical stage. The circuits are designed to accept input on, say, a low to high transition of the clock signal and to change output state on the following high to low transition. This scheme always leads to valid inputs at each successive stage as long as the clock period is longer than the longest propagation delay in the system. The timing problem is then reduced to distributing the clock signal synchronously to all stages, at the cost of a slower response to the inputs.

2.5 Timing Diagram

A Timing diagram [3] is the logical waveform of the output of one of the elements in the circuit, i.e. the timing diagram is a plot of the variation of the value of the output of one of the elements in the circuit with respect to time. Since this value could be either "Logic One Level" or "Logic Zero Level", therefore the timing diagram has a rectangular waveform nature.

DESIGN AND IMPLEMENTATION

The Library of this simulation software gives an access to 37 predefined logical design elements. It also lets one add Macros to the circuit, (Macros are elements that can be designed and added to the predefined library of elements). Brief Description of some of them is given below:

3.1 Gates

This group contains all the elementary logical gates, like AND, OR, XOR, NOR, NAND, NOT, BUFFER. Most of these gates are present in forms that can take more than one input. Their symbols and truth tables are given in Figure1.

3.2 Input /Output (I/O)

This group contains Input and Output elements like Switch, Clock, Probe, LED, Input-Pin and Output-Pin.

3.3 Sequential Elements

This group contains elements that are related to sequential circuits like RS-Flip-Flop, JK-Flip-Flop, T-Flip-Flop, D-Flip-Flop, Counter-4, and Shift-Register-4. There description is as follow;

3.3.1 Flip-Flops

A Flip-Flop [4] is a digital circuit that can maintain a binary state indefinitely (as long as power is delivered to the circuit) until directed by an input signal to switch states. The major differences among various types of flip-flops are in the number of inputs they possess and in the manner in which the inputs affect the binary state.

3.3.1.1 RS-Flip-flop

RS-Flip-Flop is an asynchronous sequential circuits with two outputs Q and \bar{Q} and two inputs set (S) and reset (R). Figure3.1 shows the logic diagram and Truth Table of RS-Flip-flop.

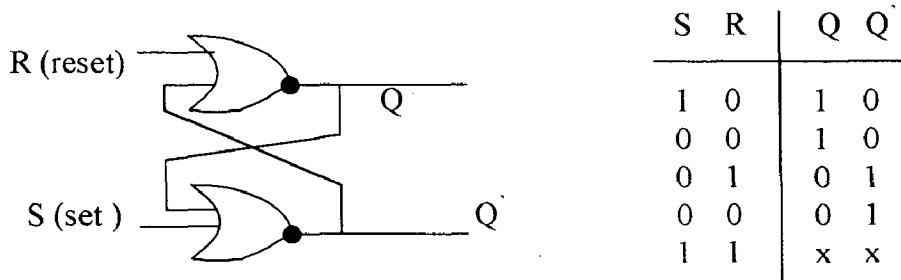


Figure3.1: Logic Diagram & truth table of R-S Flip Flop

3.3.1.2 JK-Flip-flop

JK Flip-Flop is a refinement of RS Flip-Flop in that the indeterminate state of the RS Flip-flop is defined in the JK Flip-Flop. Inputs J and K behave like inputs S and R to set and clear the Flip-Flop (note that in JK Flip-Flop, the letter J is for set and letter K is for Reset) Figure3.2 shows the logic diagram and Truth Table of JK-Flip-flop.

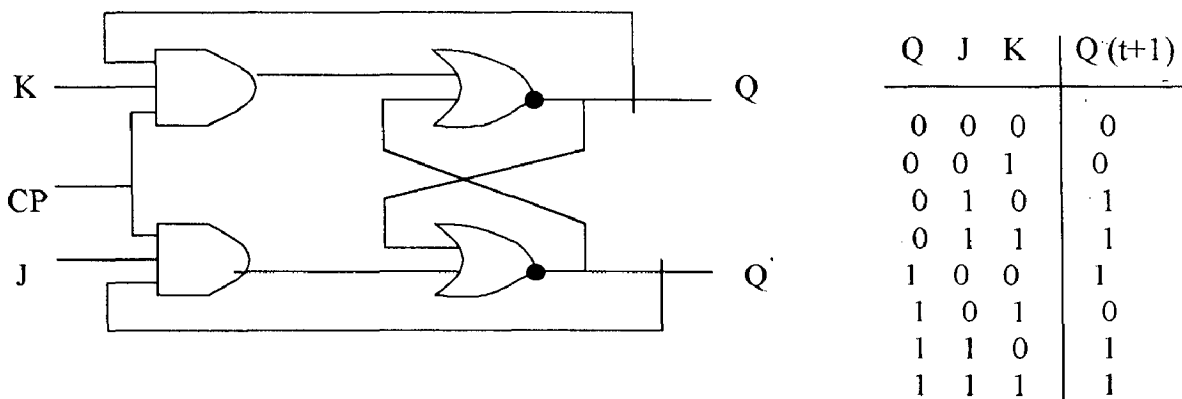


Figure 3.2: Logic diagram & Truth Table of JK-flip Flop

3.3.1.3 T-Flip-flop

The T Flip-Flop is a single version of the JK Flip-Flop. The T Flip-Flop is obtained from JK Flip-Flop if both the inputs are tied together. The designation T comes from the ability of the Flip-Flop to “toggle”, or changing state. Regardless of the present state of the Flip-flop, it assumes the complement state when the clock pulse occurs while input T is logic-1. Figure 3.3 shows the logic diagram and Truth Table of T-Flip-flop.

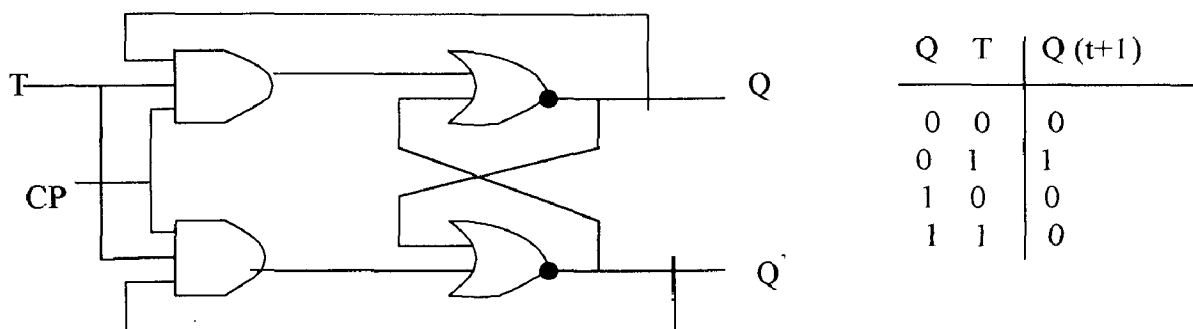


Figure 3.3: Logic diagram & Truth Table of T-Flip-Flop

3.3.1.4 D-Flip-flop

The D Flip-Flop is a modification of the RS Flip-flop. The D input goes directly to the S input, and its complement is applied to R input. The input is sampled during the occurrence of the clock pulse. The D Flip-Flop receives the designation from its ability to transfer “data” into a flip-flop. Figure 3.4 shows the logic diagram and Truth Table of D-Flip-flop.

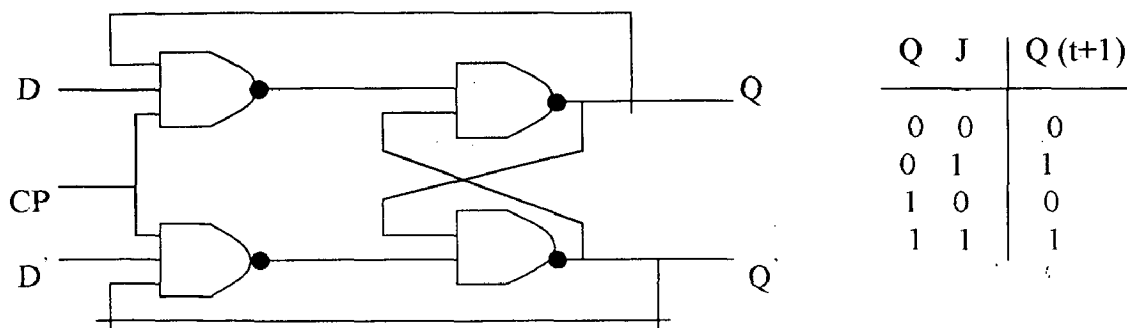


Figure 3.4: Logic diagram & Truth Table of D-flip flop

3.3.2 Counter-4

Counter-4 is sequential circuit that counts up to 16 clock pulses supplied to it externally. Figure 3.5 shows the block diagram of counter. 'n' denotes the number of bits used to represent the count.

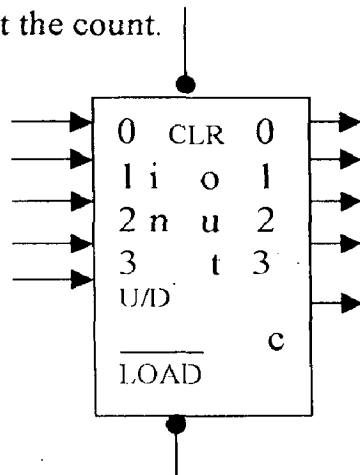


Figure 3.5: Block Diagram of Counter-4

3.3.3 Shift Register

A Register [4] is a group of binary storage cells suitable of holding binary information. A group of Flip-flops constitute a register, since each flip-flop is a binary cell capable of holding one bit of information. An n-bit register has a group of n Flip-flops and is capable of storing any binary information containing n bits.

A Shift Register is a Register capable of shifting its binary information either to left or to the right. The Logical configuration of a shift register consists of a chain of flip-flops connected in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses, which causes shift from one stage to next. Figure 3.6 shows its logical circuit.

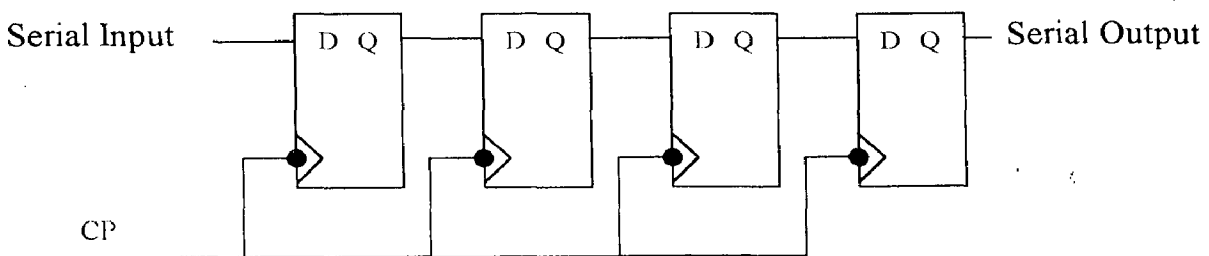


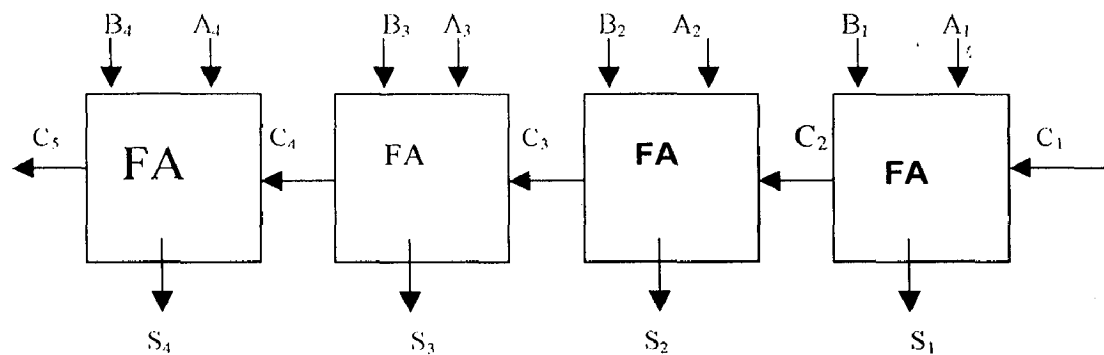
Figure 3.6: Logical Diagram of Shift Register

3.4 Combinational Elements

This group contains elements that are related to combinational circuits like 4-bit full Adder, 4-bit Comparator, 8-input line Multiplexer, 8 output line Demultiplexer, Octal to binary Encoder, 3 to 8 line Decoder, and Data Selector. There short description is as follows:

3.4.1 4 - bit full Adder

4-bit Binary parallel Adder is a Digital Function that adds up the two 4-bit binary numbers in parallel. It's logic diagram and block diagram is shown in Figure3.7.



Logical Diagram of 4-bit Full Adder

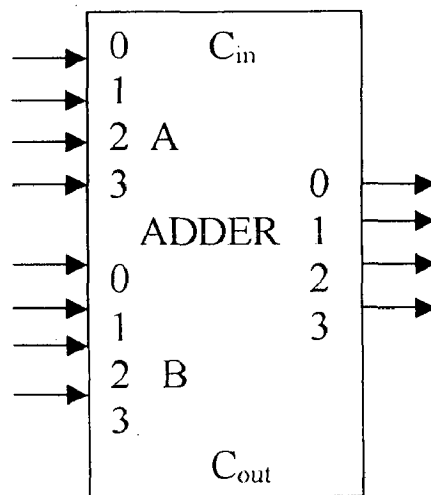


Figure 3.7: Block Diagram of 4-bit full Adder

3.4.2 4-bit Comparator

A Magnitude Comparator is a combinational circuit that compares two numbers, A and B, and determines their relative magnitudes. The Outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, $A < B$. Figure 3.8 shows block diagram of 4 bit comparator.

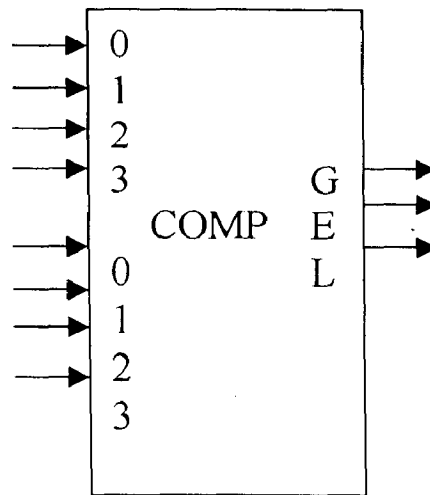


Figure 3.8: Block Diagram of 4-bit Comparator

3.4.3 8 input line Multiplexer

Multiplexing [5] means transmitting a large number of information units over a smaller number of channels or lines. A Digital Multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The Selection of a particular input line is controlled by a set of selection lines. Figure 3.9 shows the block diagram of 8 bit multiplexer.

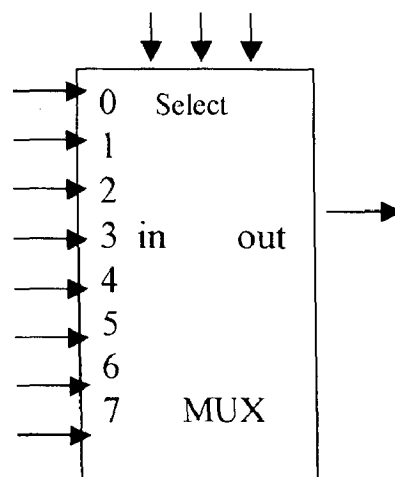


Figure 3.9: Block Diagram of 8 Bit Multiplexer

3.4.4 8 – output line Demultiplexer

A Demultiplexer is a logic circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The Selection of specific output line is controlled by the bit values of n selection lines. A Decoder with an enable input is referred to as Demultiplexer. Figure 3.10 shows the block diagram of Demultiplexer with 8 output lines.

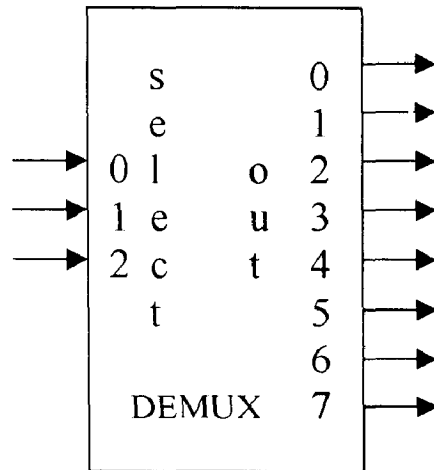


Figure 3.10: Demultiplexer with 8 output lines

3.4.5 3 to 8 line Decoder

A Decoder is a combinational circuit that converts binary information from n input lines to a maximum of 2^n unique output lines. If the n -bit decoded information has unused or don't care combinations, the decoder output will have less than 2^n outputs. Figure 3.11 shows the Truth table of 3 –to- 8-line decoder. A 3-to-8 line Decoder can be used for decoding any 3-bit code to provide eight outputs, one for each element of the code.

Inputs			Outputs							
X	Y	Z	D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Truth Table of 3-to-8-line decoder

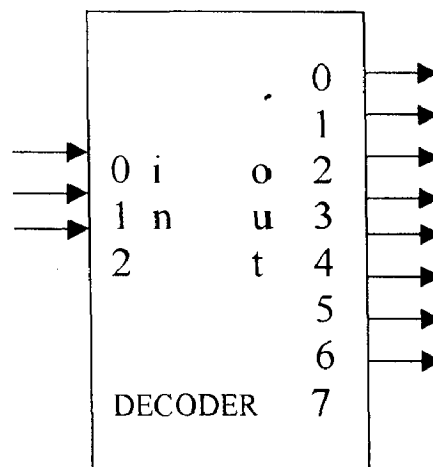


Figure 3.11: Block Diagram of Decoder

3.4.6 Octal to Binary Encoder

An Encoder [5] is a digital function that produces a reverse operation from that of a decoder. An encoder has 2^n (or less) input lines and n output lines. The output lines generate the binary code for the 2^n input variables. The Encoder assumes that only one input line can be equal to 1 at any time; otherwise the circuit has no

meaning. Figure 3.12 shows Truth Table and block Diagram of octal to binary Encoder.

Inputs								Outputs		
D ₀	D ₁	D ₂	D ₃	D ₄	D ₅	D ₆	D ₇	X	Y	Z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Truth Table of Octal to binary Encoder

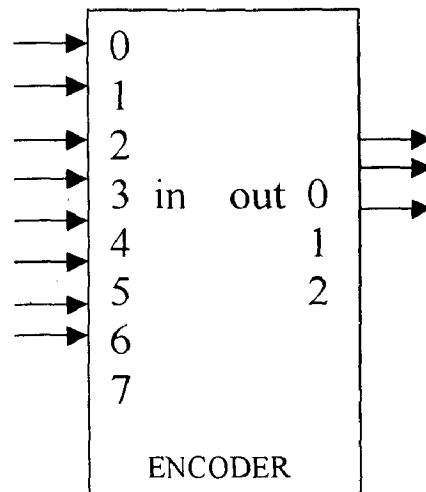


Figure 3.12: Block Diagram of Encoder

3.4.7 Data selector

A Multiplexer is also known as a Data Selector, since it selects one of the many information and steers to the output. This logic Circuit performs its functions with the help of select lines available. Figure 3.13 shows its Block diagram.

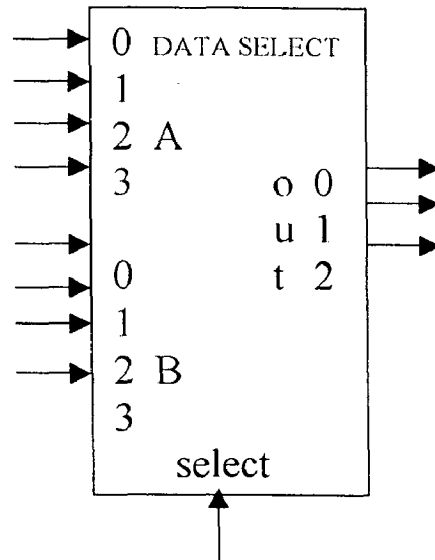


Figure 3.13: Block Diagram of Data Selector

3.5 Macros

Macros are user-defined elements that could be added to software's predefined library. They are one of the most powerful features that this software present. Macros are Modular. Therefore, They reduce the amount of work because they can be used in the same circuit more than once.

3.6 Source Code Description and Data Dictionary

ACTIVITY.H

It defines the abstract class Activity; this is responsible of choosing the required function to be activated.

ACTIVITY.CPP

It includes the code implementation for the members of the instance class Activity.

ACTWIND.H

It defines the abstract class Activity Window. This is responsible for choosing the required function to be activated from a menu.

ACTWIND.CPP

It includes the code implementation for the members of the abstract class Activity Window.

BRDWIND.H

It defines the instance class Border-Window, which directly inherits from Window. It is a variation of Window that contains a single or double border and an optional shadow.

Border-Window have characteristics similar to Window add to that the capability of storing the overlapped area during pop-up and restoring it during retrieval. It also has a double or single border, an optional shadow, resizing and dragging capabilities

Constructors

- BorderWindow(int x,int y,int length,int height,char attrib,Device* baseWindow)

Constructs the `BorderWindow` and initializes its coordinates, size, `baseWindow` with a default double border and shadow configuration. It also allocates enough memory to pertain the overlapped area.

Public Members

- `IsA` : Returns an unique identifying quantity (`borderWindowClass`).
- `NameOf` : Returns a pointer to the character string "BorderWindow".
- `resize(Point newDimensions)`: Changes the dimensions and updates the current window display.
- `move(Point relative_displacement)`: Changes the anchor point of the window by the given relative displacement and updates its displayed position.
- `show()`: Pertains the overlapped area to the allocated buffer and displays the `BorderWindow` in its current configuration.
- `hide()`: Retrieves the overlapped area from the allocated buffer implicitly hiding the `BorderWindow`.

Inherited Members

- `displayCH (x , y , no. , char ,attribute)`: Displays a certain char and attribute horizontally.
- `displayCV (x , y , no. , char ,attribute)`: Displays a certain char and attribute vertically.
- `cHAttribut (x , y , no. , attribute)`: Changes the attribute of a line horizontally .
- `cVAttribut (x , y , no. , attribute)`: Change the attribute of a line vertically .
- `getText (x , y , length , height , buffer (int *))`: Copies a text area to an allocated part of the memory pointed to by the buffer pointer.
- `putText (x , y , length , height , buffer (int *))`: Returns a text from the memory area pointed to by the buffer pointer .
- `scroll` :
 - [Up

- [Down
- [Right
- [Left

Scrolls a certain rectangular area of anchor point (x,y) and dimensions (length, height) in the given direction.

- `getDimension()` : Returns the current dimension .
- `writeStr(int x, int y, char attribute, char* string)`: Prints string in the Window at location x,y with the given attribute.
- `fill(int x,int y,int length,int height,char fillChar,char fillAttrib)`: Fills an area of Window with anchor point x,y and dimensions length, height with the given character or/and attribute.
- `getBaseWindow()` : Returns the current basePointer indicating which output Screen is currently being used.
- `putBaseWindow(Device* newDevice)`: Modifies the current basePointer to a new one given in newDevice, so that the output area and mechanism could be changed easily at any time.

Protected Members

- `ShowOn`: Flag indicating the current (shown/hidden) state of the window.
- `Color`: The window color.
- `Background` : Pointer to the allocated text buffer containing the pertained text.
- `DoubleBorder` : Flag indicating the current (double/single) border of the window.
- `Shadow` : Flag that indicates the existence of a shadow.

BRDWIND.CPP

It includes the code implementation for the members of the instance class `BorderWindow`.

CBM.H

It defines the abstract class `CBM`. `CBM` is the Circuit Builder and modifier.

CBM.CPP

It includes the code implementation for the members of the abstract class CBM.

DESKTOP.H

It defines the instance class DeskTop. DeskTop inherits directly from PrimerWindow. It constitutes the background surface of the desktop, all other windows emerge from DeskTop or from a window already opened from within the DeskTop. It could be thought of as the base of all activities and status informing.

Constructors

- DeskTop() : Constructs the PrimerWindow and initializes the screen.

Public Members

- isA(): Returns an unique identifying quantity (primerWindowClass).
- nameOf(): Returns a pointer to the character string "PrimerWindow".

Inherited Members

- getKey(): Returns the main Key confined to the PrimerWindow. Where Key contains the scan code and ASCII code of the pressed keyboard key.
- getCursor(): Returns the mouse's cursor current position.
- displayCH (x , y , no. , char ,attribute) : Displays a certain char and attribute horizontally.
- displayCV (x , y , no. , char ,attribute): Displays a certain char and attribute vertically.
- cHAttribut (x , y , no. , attribute): Changes the attribute of a line horizontally .
- cVAttribut (x , y , no. , attribute): Change the attribute of a line vertically .
- getText (x , y , length , height , buffer (int *)): Copies a text area to an allocated part of the memory pointed to by the buffer pointer.

- `putText (x , y , length , height , buffer (int *))` : Returns a text from the memory area pointed to by the buffer pointer .
- `scroll` :
 - [Up
 - [Down
 - [Right
 - [Left

Scrolls a certain rectangular area of anchor point (x,y) and dimensions (length,height) in the given direction.
- `resize(Point newDimensions)` : Changes the dimensions .
- `getDimension()` : Returns the current dimension .
- `move(Point relative_displacement)`: Changes the anchor point of the window by the given relative displacement and updates its displayed position.
- `writeStr(int x, int y, char attribute, char* string)`: Prints string in the Window at location x,y with the given attribute.
- `fill(int x,int y,int length,int height,char fillChar,char fillAttrib)`: Fills an area of Window with anchor point x,y and dimensions length, height with the given character or/and attribute.
- `getBaseWindow()`: Returns the current basePointer indicating which output Screen is currently being used.
- `putBaseWindow(Device* newDevice)`: Modifies the current basePointer to a new one given in newDevice, so that the output area and mechanism could be changed easily at any time.

Protected Members

- `Cursor` : The mouse's Cursor current position and status.
- `Key`: The keyboard's pressed key ASCII and scan code handling object.

DESKTOP.CPP

It includes the code implementation for the members of the instance class DeskTop.

DEVICE.H

It defines the abstract base class Device. Device is the class at root of the desktop hierarchy. Device is derived from the Object abstract base class defined in the TC++ library.

Constructors

- Device(int Length,int Height): Sets the dimensions.

Destructors

- ~Device(): Used for setting breakpoints during debugging.

Private Members

- Dimension : the dimension of the device .

DEVICE.CPP

It includes the code implementation for the members of the abstract base class Device.

EDITLINE.H

It defines the instance class EditLine that directly inherits from window. EditLine could be considered as an elementary editor. It could be used to get input from the user in a comprehensive form allowing deletion, insertion and manipulation of the input text. Creating an array of EditLine (with appropriate adjustments) constitutes a basic editor. EditLine could be used any place where a text input is expected from the user.

Constructors

- EditLine() : Constructs EditLine and initializes its coordinates, size and base pointer, also creates a text buffer to hold the text being edited.

Destructors

- `~EditLine()` : Destructs the `EditLine` and frees the allocated memory buffer created beforehand to contain the text being edited.

Public Members

- `isA()` : Returns a unique identifying quantity (`editLineClass`).
- `nameOf()` : Returns a pointer to the character string "EditLine".
- `redraw(int redrawAlways)` : Updates the portion of the string to be displayed on the screen after checking if it is necessary to update. If `redrawAlways` is non-zero (i.e. true) it will redraw even though it is unnecessary to update.
- `read(char* defaultString)` : Displays a portion of the `defaultString` and waits to read further input from the user. This further input could contain actual editing of the displayed `defaultString` or even complete deletion of it and insertion of new text. In other words this is the heart of `EditLine` where most of other `EditLine`'s member functions meet.
- `scanKey(char ch, char cl)` : Recognize whether the pressed key is a normal character or an editing key and responds consequently. If it is an editing key it analyzes it and call its corresponding editing functions. If the ESC key is pressed it returns a false flag.
- `shiftRight()` : Moves the cursor to the right and check bounds.
- `shiftLeft()` : Moves the cursor to the left and check bounds.
- `insertChar(int& place, char character)` : Inserts the given character at the given place in the string.
- `overWriteChar(int& place, char character)` : Puts the given character over the old character at the given place.
- `deleteChar(int& place)` : Deletes a character at the given place and pulls the following group of characters.

- `show()` : Redraw(s) the string for the first time. Overrides the inherited `show()`.
- `getLength()` : Returns the length of the string buffer.
- `getStringLength()` : Returns the displayed string segment length. i.e. the width of the EditLine window.
- `getString()` : Returns the string pointer.
- `putString(const char* string)` : Puts a string in the allocated buffer.
- `getPosition()` : Returns the current cursor position.
- `getAttribute()` : Returns the current attribute.
- `setAttribute(char newAttrib)` : Changes the current attribute to a `newAttrib`.

Inherited Members

- `getDimension()` : Returns the current dimension .
- `writeStr(int x, int y, char attribute, char* string)` : Prints string in the Window at location x,y with the given attribute.
- `fill(int x,int y,int length,int height,char fillChar,char fillAttrib)` : Fills an area of Window with anchor point x,y and dimensions length, height with the given character or/and attribute.
- `getBaseWindow()` : Returns the current basePointer indicating which output Screen is currently being used.
- `putBaseWindow(Device* newDevice)` : Modifies the current basePointer to a new one given in `newDevice`, so that the output area and mechanism could be changed easily at any time.

Protected Members

- `String` : Pointer to the allocated text buffer in the memory.
- `StringLength` : The length of the displayed part of the string, i.e. the width of the EditLine window.
- `Length` : The length(size) of the allocated string buffer. This parameter is overridden if a string with length greater than it is supplied.

- **Start** : The start position from within the string buffer for the segment to be shown within the width of the EditLine window.
- **Pos** : The position of the cursor within the string.
- **Attribute** : The color of the EditLine's foreground and background.

EDITLINE.CPP

It includes the code implementation for the members of the instance class EditLine.

OURLIST.H

It defines the class list.

OURLIST.CPP

It includes the code implementation for the members of instance class list.

SCREEN.H

It Defines the instance class Screen. Screen inherits from Device and is used to define an Actual or Virtual screen. Screen is the base of the physical display hierarchy. All member functions of screen dealing with display are implemented via direct memory access to optimize the performance of the desktop package.

Constructors

- **Screen()**: Initializes the actual screen according to the detected hardware.
- **Screen(int Length,int Height,int Seg,int Ofs)** : Used to create a user virtual screen.

Private Members

- **ScreenSeg** : The segment of the screen memory address.
- **ScreenOfs** : The offset of the screen memory address.

SCREEN.CPP

It includes the code implementation for the members of the instance class Screen.

TIMING.H

It defines the TDElement class, it is the unit of time diagram. It also defines the instance class TDManager. TDManager is responsible for the timing diagrams of the circuit.

USERDEFS.H

It contains the "unique identifying quantity" definitions for all the user-defined classes within the project.

VIEWPORT.H

It defines the instance class ViewPort, which directly inherits from BorderWindow. It is a variation of BorderWindow that contains a single or double border and an optional shadow. ViewPort have characteristics similar to Window add to that the capability of storing the overlapped area during pop-up and restoring it during retrieval. It also has a double or single border, an optional shadow, resizing and dragging capabilities.

Constructors

- ViewPort(int x,int y,int length,int height,char attrib,Device* baseWindow)
: Constructs the ViewPort and initializes its coordinates, size, baseWindow with a default double border and shadow configuration. It also allocates enough memory to pertain the overlapped area.

Destructor

- ~ViewPort : Destruits the ViewPort and releases the pre-allocated buffer area.

Private Members

- ShowOn : Flag indicating the current (shown/hidden) state of the window.
- Color : The window color.
- Background : Pointer to the allocated text buffer containing the pertained text.
- DoubleBorder : Flag indicating the current (double/single) border of the window.
- Shadow : Flag that indicates the existance of a shadow.
- ShowOn : Flag indicating the current (shown/hidden) state of the window.
- Color : The window color.

VIEWPORT.CPP

It includes the code implementation for the members of the instance class ViewPort.

WINDOW.H

It defines the instance class Window. Window inherits directly from Device and is the base of the logical display hierarchy. It uses the code implementation of Screen via base pointer after checking and validating the input parameters. Inherently, it allows recursion (i.e. a window inside a window and so on). It is the logical counterpart of Screen.

Constructors

- Window(int x relative, int y relative,int length,int height, Device*) :
Constructs the Window, and initialises its coordinates, size and base pointer (by base pointer it is meant that Device logical descendant "Screen" which Window use in physically accessing the display area).

Private Members

- BaseWindow : Points to the base Device (e.g. Screen, Window, etc..) which the current window uses for physical code implementation.
- RelPoint : The relative position of the anchor point of the current window from the anchor point of its base device.

WINDOW.CPP

It includes the code implementation for the members of the instance class Window.

3.7 Generated files

There are mainly two files, which are generated by this software.

3.7.1 *.CRC files

These are circuit files. Whenever, A new circuit is being created and stored, it get stored on the disk with this default extension.

3.7.2 *.MAC files

These are macro files that can be used as a building block for designing a circuit, or even in creating bigger macros. Note that *.CRC files and *.MAC files having the same name are the same circuits but in circuit and macro forms respectively. Whenever, A new circuit is stored as a macro, it get stored on the disk with this default extension.

3.8 Timing Diagrams

A Timing diagram is the logical waveform of the output of one of the elements in the circuit, i.e. the timing diagram is a plot of the variation of the value of the output of one of the elements in the circuit with respect to time. Since this value could be either "Logic One Level" or "Logic Zero Level", therefore the timing diagrams have a rectangular waveform nature.

3.8.1 Generation of Timing Diagrams

This software has a Timing Diagrams Editor. Timing diagram entries could be added or removed to the Timing Diagram List which software scans periodically while running to display the entries init on the Timing Diagrams Display. What actually happens inside it is that it scans the Timing Diagrams List every USTS (Unit Simulation time slot), and while scanning it, it updates the Timing Diagrams Display. The scaling of the Timing Diagrams Display is relative to the USTS, where the smallest unit on the Timing Diagrams Display is the USTS itself. Hence the speed of flow of timing diagrams depend on the number of USTS which it process per second, and consequently on the speed of simulation which could be controlled by pressing 'F' for "Faster" and 'S' for "Slower".

RESULTS AND DISCUSSION

This circuit simulation software is working properly. Some of the basic circuits are already tested whose Screenshots are given here. Some of the features that forms a basis for its extension and further work to carry over it to make it more sophisticated and user friendly.

- ✓ Mouse Support.
- ✓ Macro Frame Editing.
- ✓ Drafting Support.
- ✓ Printing Support.

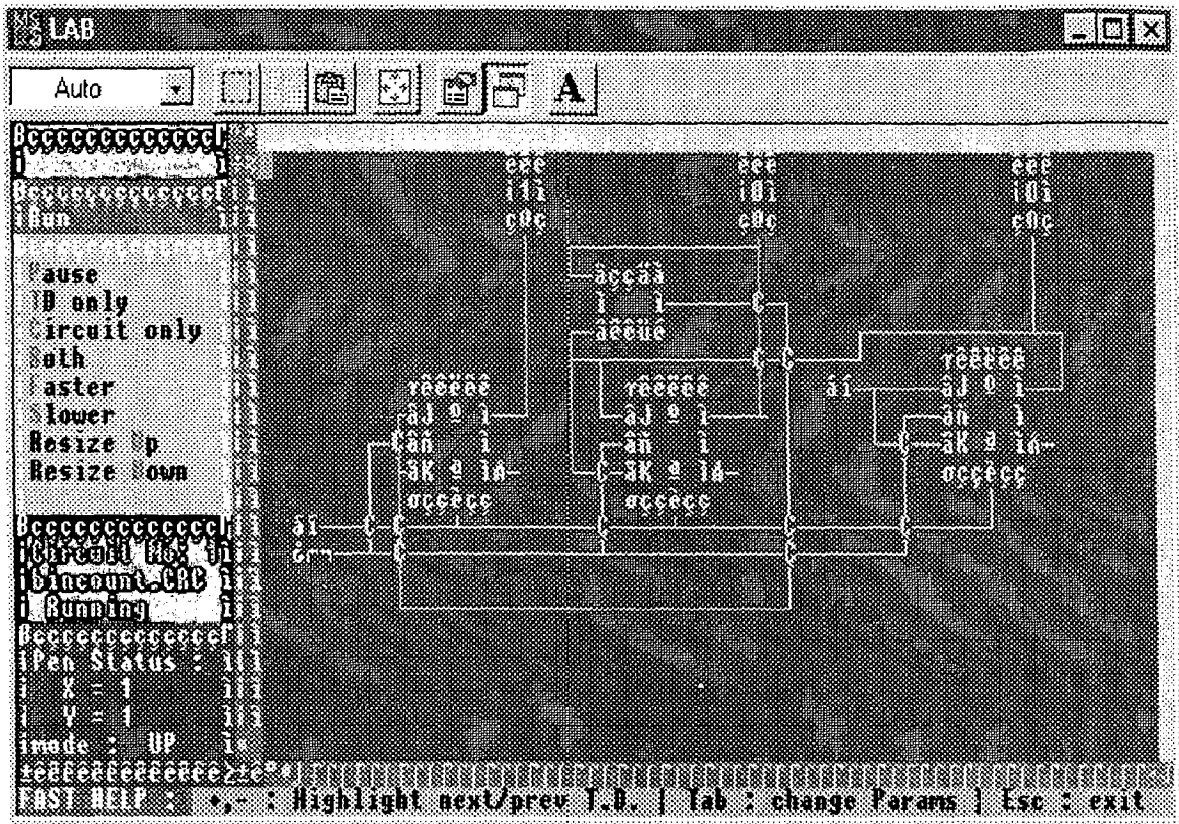


Figure 5.1: Output of Binary counter

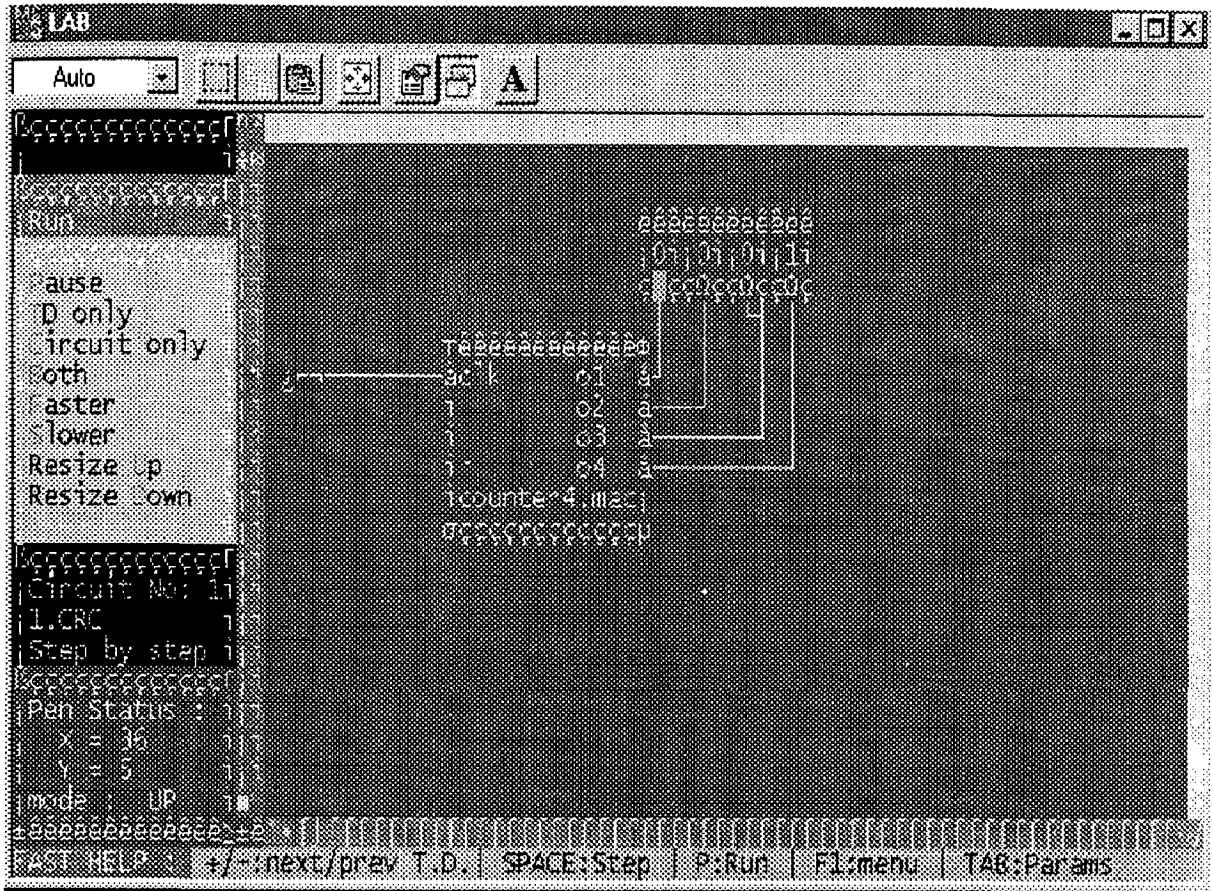


Figure 5.2: Binary counter as a Macro

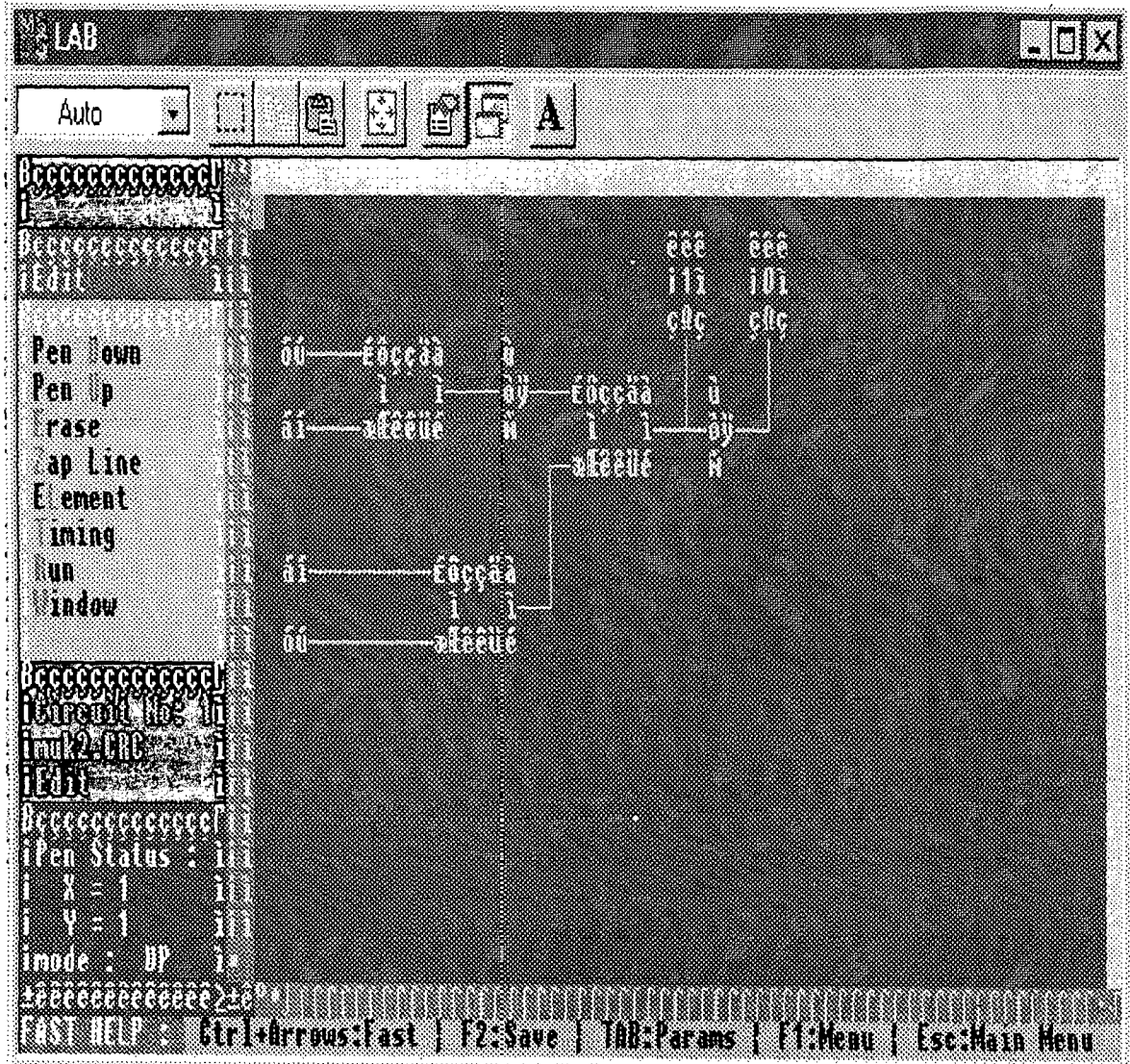
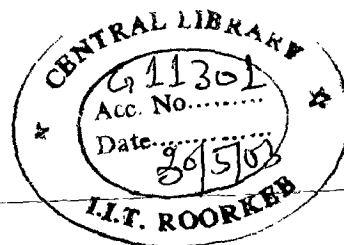


Figure 5.3: Output of a simple combinational circuit



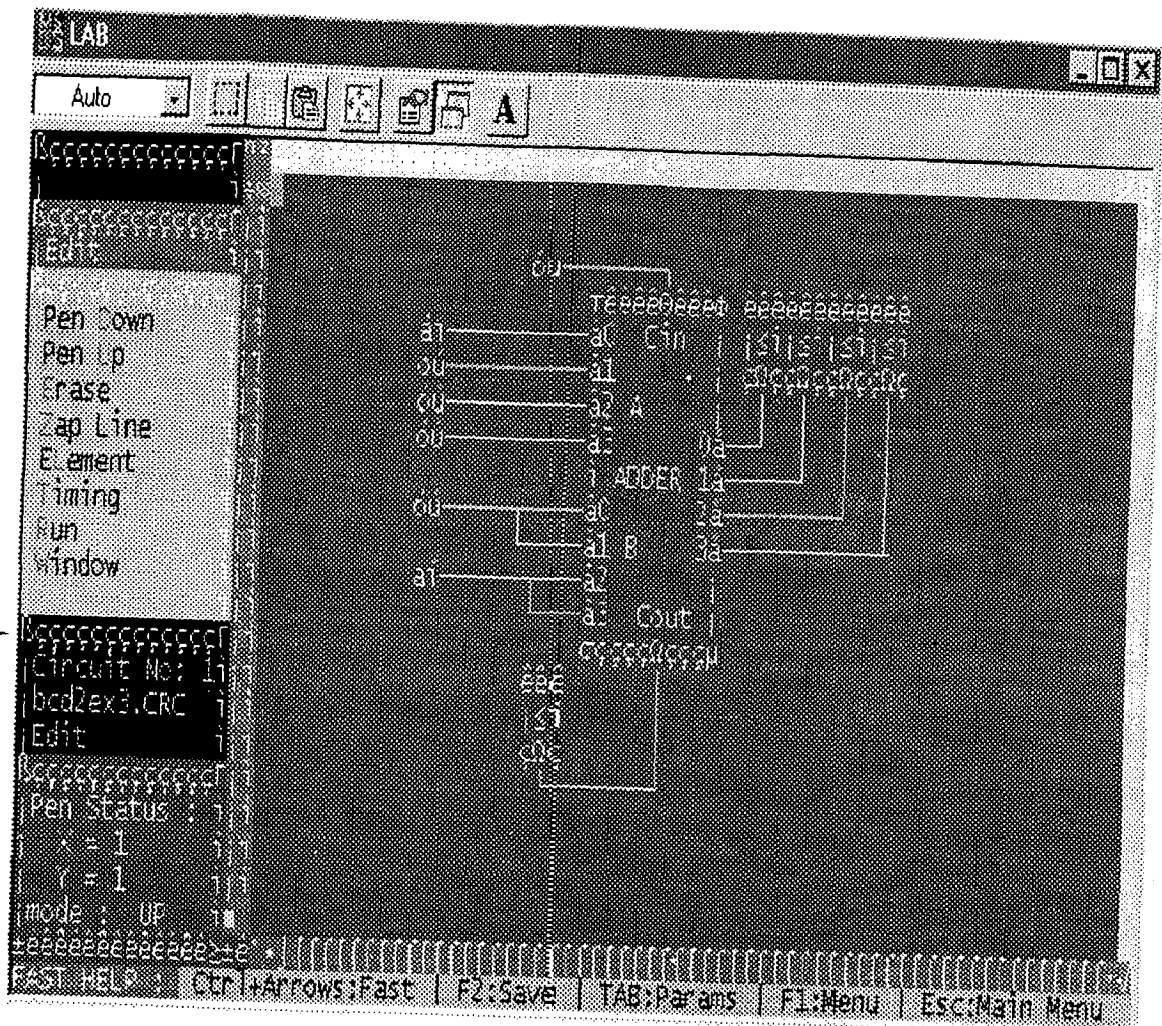


Figure 5.4 : Digital circuit for the BCD to Extended -3 Decimal code convertor.

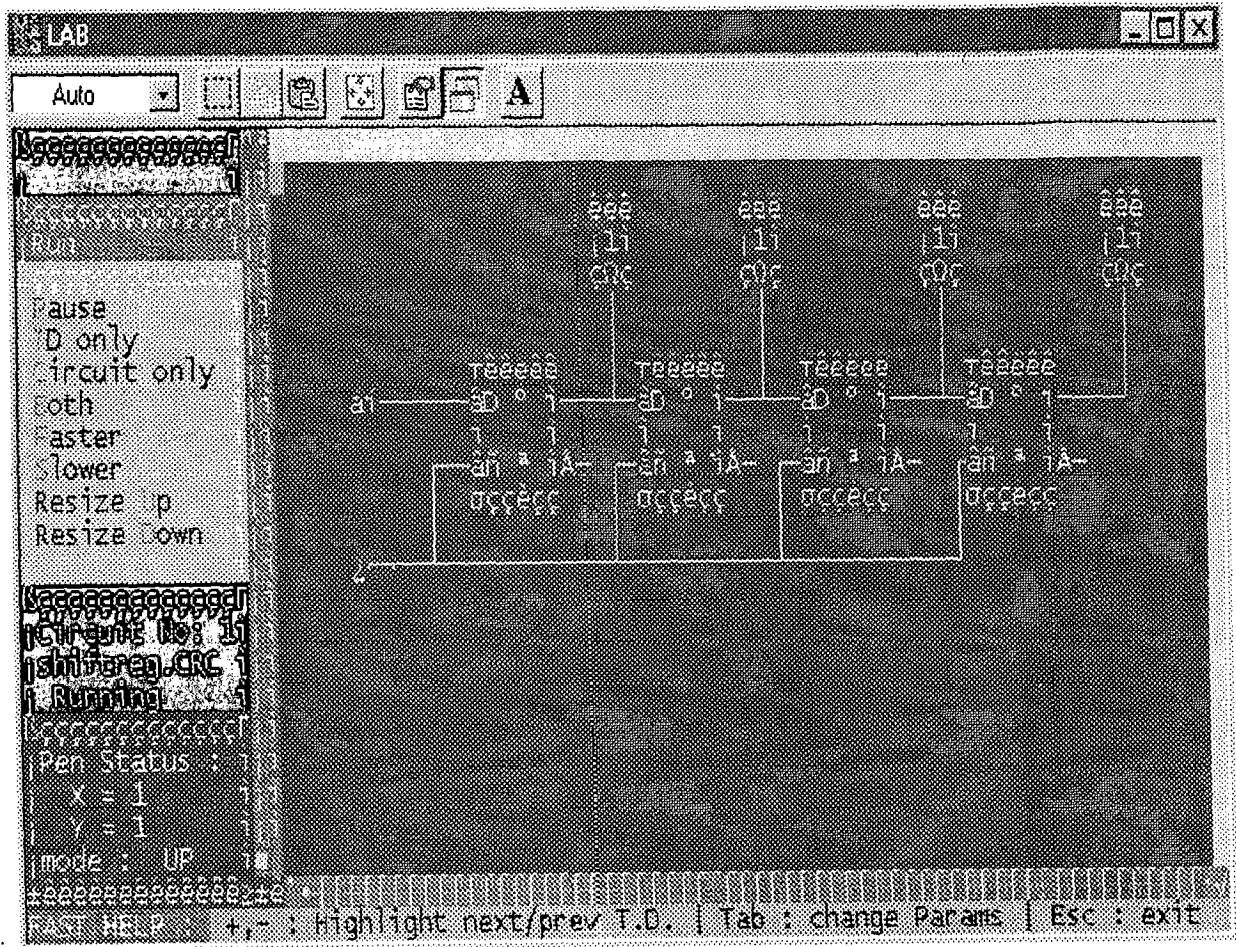


Figure 5.5: Logical diagram of shift-register

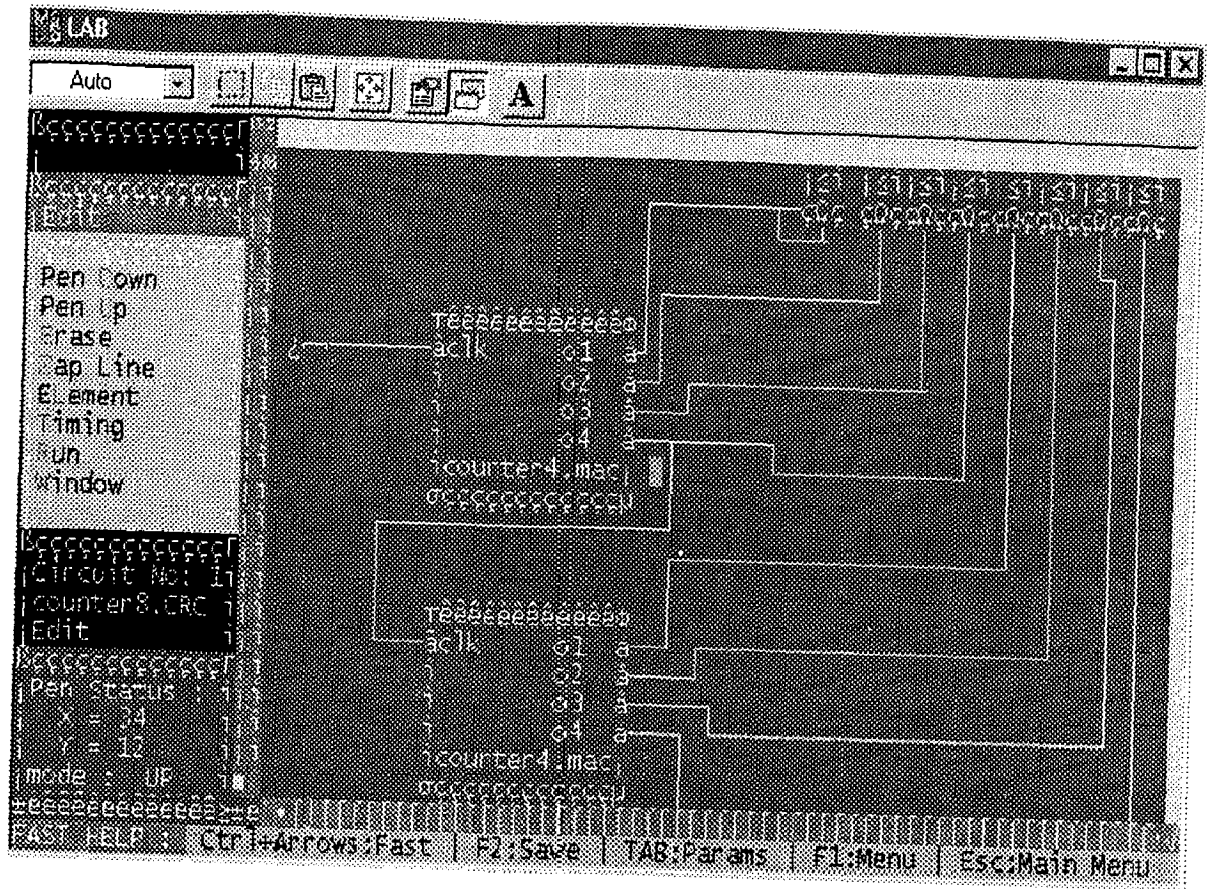


Figure 5.6: Logical diagram of 8 bit counter using 4 bit counter macros as its elements

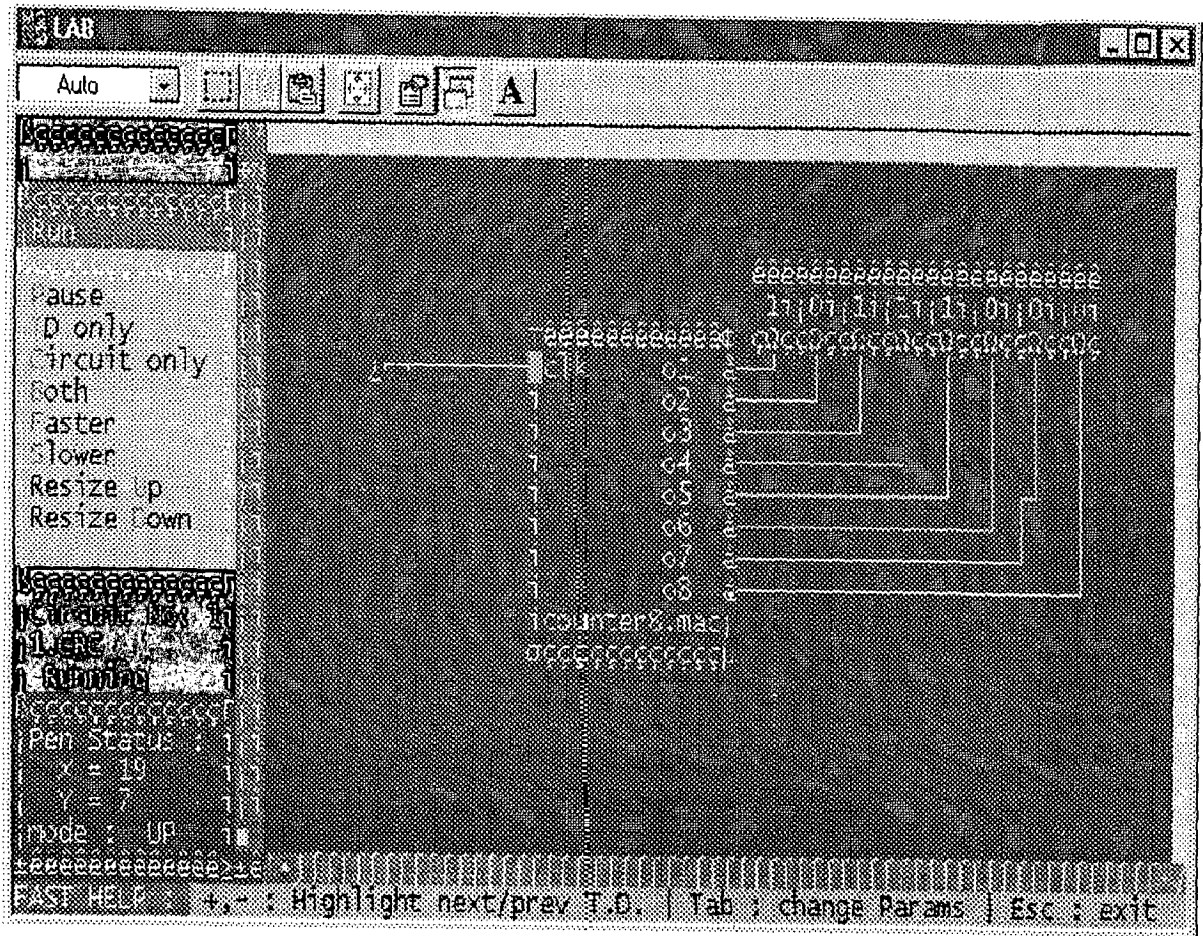


Figure 5.7: Macro version of 8 bit counter

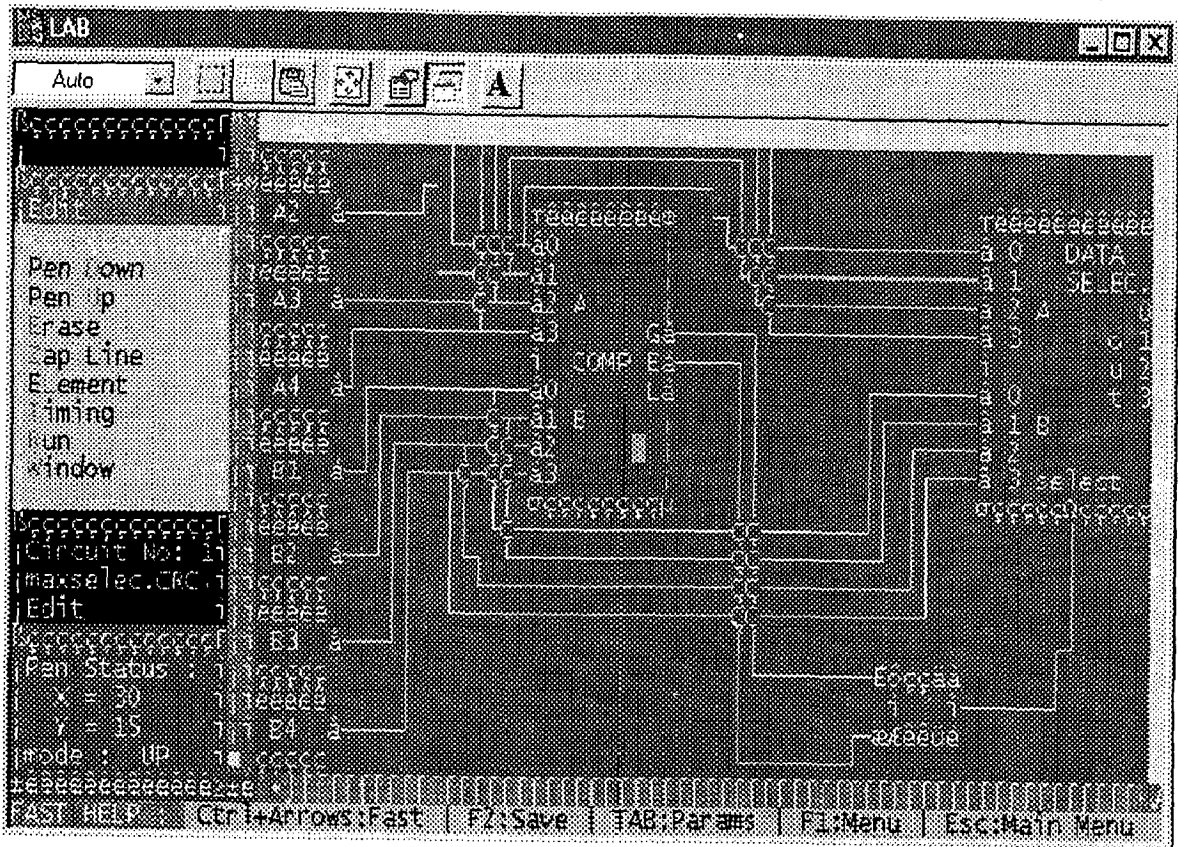


Figure 5.8: Logical diagram of Comparator Circuit

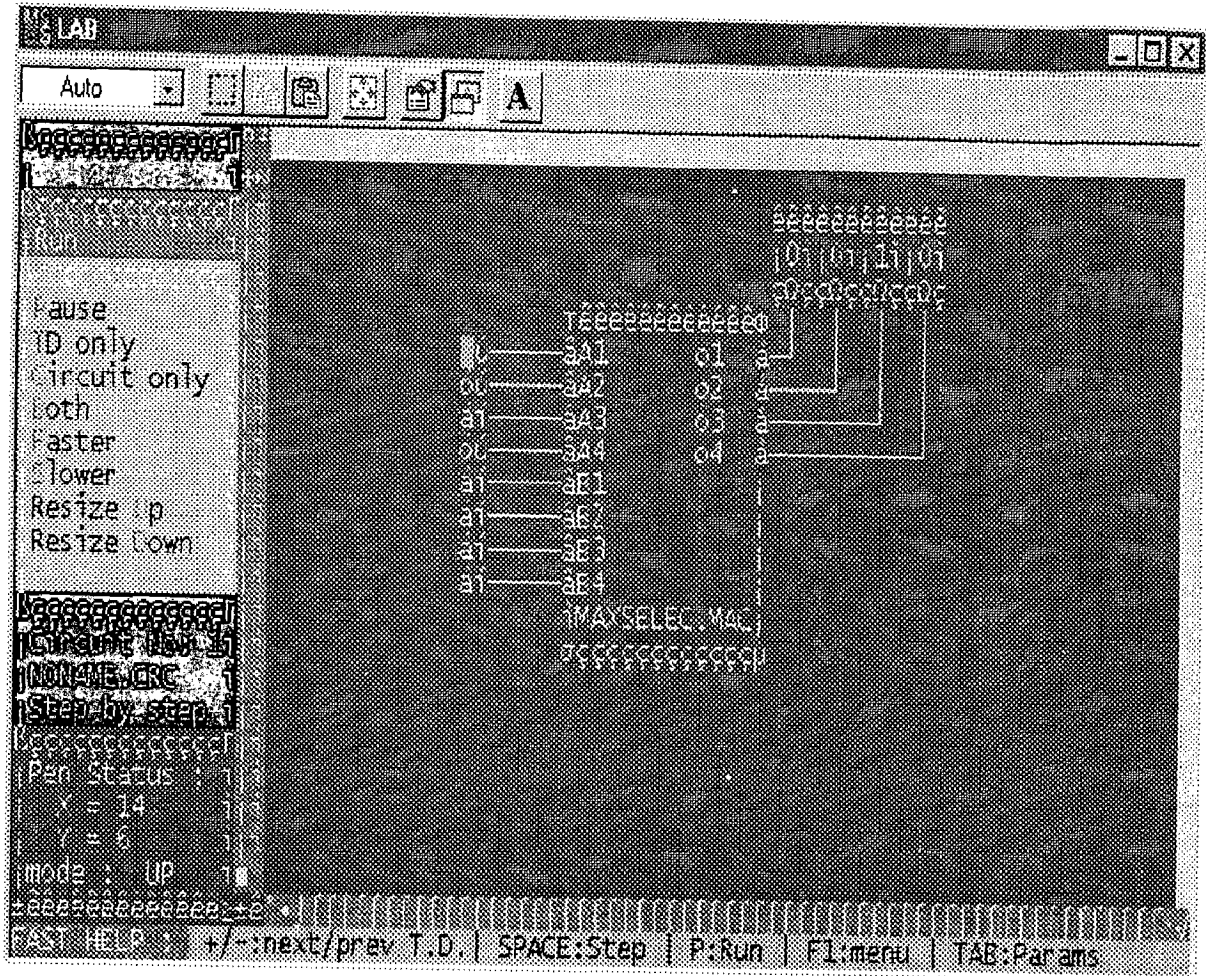


Figure 5.9: Comparator circuit as a Macro

REFERENCES :

- [1] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen. Object Oriented Modeling and Design, Ninth Edition, Eastern Economy Edition, 2000.
- [2] Albert Paul Malvino, Donald P. Leach. Digital Principles and Applications, fourth Edition, Tata Mc-Graw Hill Edition, May 2000.
- [3] Ronald J. Tocci, Neal Widmer. Digital Systems – Principles and Applications, Sixth Edition, Eastern Economy Edition, August 2001.
- [4] M. Morris Mano, Digital Logic and Computer Design,
- [5] M. Morris Mano, Digital Design, Eastern Economy Edition, September 2000

APPENDIX A

List of Hot-Keys and their Description:

- <LEFT>, <RIGHT>, <UP>, <DOWN> cursor keys: Press these keys to travel up, down, left and right within the text. Fast word movement can be achieved by pressing down the <CTRL> key.
- <Home>: Pressing this key moves the cursor immediately to the start of the string.
- <End>: Pressing this key moves the cursor immediately to the end of the string.
- <Delete>: Pressing this key deletes the character under the cursor, and pulls the remaining string.
- <Backspace>: Pressing this key deletes the character to the left of the cursor, and pulls the remaining string.
- <Esc>: Pressing this key abandons the input operation.
- <Enter>: Pressing this key accepts the input operation.
- < ' O ' >: Pressing this key open up an already existing circuit.
- < ' N ' >: Pressing this key open up a new circuit.
- < ' E ' >: Pressing this key will open up a new window for editing the circuit.

- < ' X ' >: Pressing this key will open up a new circuit
- < ' M ' >: Pressing this key will save the active circuit as a macro.
- < ' C ' >: Pressing this key closes the existing circuit.
- < ' Q ' >: Pressing this key closes the main window and returns to windows.
- < ' U ' >: Pressing this key turns the status of pen to UP
- < ' D ' >: Pressing this key turns the status of pen to DOWN.
- < ' E ' >: Pressing this key turns the status of pen to ERASE.
- < ' Z ' >: Pressing this key will trace this line and delete it.
- < ' L ' >: Press this key when it is required to Add, Move, Remove or Change the parameters of one of the logical elements within the logical circuit.
- < ' T ' >: Pressing this key gives an access to the Timing Diagrams Editing Panel.
- < ' R ' >: Pressing this key starts running the circuit simulation.
- < ' L ' >: Pressing this key gives an access to Element menu.
- < ' A ' >: Pressing this gives an access to library elements to add them on worksheet.

- < ' C ' >: Pressing this key gives an access to combinational logic elements.
- < ' R ' >: Pressing this key removes the selected element from the worksheet.
- < ' P ' >: Pressing this key open up a window to change the element parameter.

