M. Tech. (Computer Science) Dissertation Series

Function Extraction & Verification of General MOS Transistor Circuits

DISSERTATION SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENT FOR THE DEGREE
OF

[redacted]

MASTER of TECHNOLOGY

in

COMPUTER SCIENCE

By

Debashis Sarkar

Under the guidance of
Prof. BHARGAB BIKRAM BHATTACHARYA

ADVANCE COMPUTING & MICROELECTRONICS UNIT

INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road
Calcutta-700 035

30th July, 1997

Copy to library for users.

Subarkin Sarkar.

1/8/91.

Certificate of Approval

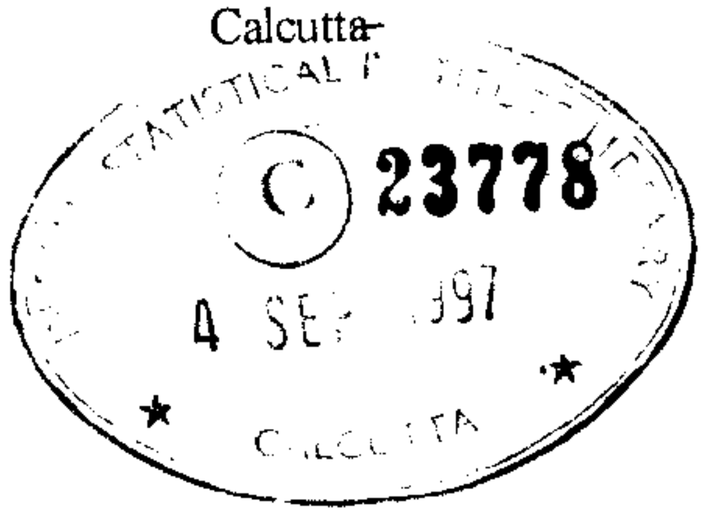
This is to certify that the dissertation titled **Function Extraction and Verification of General MOS Transistor Circuit** *submitted by* **Mr. Debashis Sarkar** *to the* **Advance Computing and Microelectronics Unit, Indian Statistical Institute, Calcutta** *in partial fulfilment of the award of the degree of* **Master of Technology in Computer Science**, *is a bona fide record of the work and investigation carried out by him under my supervision and guidance.*

Date : 30th July, 1997
Indian Statistical Institute
Calcutta.

B. B. Bhattacharya

(Dr. Bhargab Bikram Bhattacharya)

Professor,
Advanced Computing and Microelectronis Unit
Indian Statistical Institute
Calcutta



Acknowledegments

It brings me immense pleasure to express my deep sense of indebtedness to my guide Prof. Dr. Bhargab B. Bhattacharya for his close supervision and constant encouragement throughout the project work. I really left with no phrases to express my gratitude to him for his personal care and timely suggestions. I take this opportunity to thank Prof. Prabal Sengupta who motivated me implicitly, throughout my course of study. I am greatly indebted to him for his friendly cooperation.

I am thankful to all my friends and classmates who helped me directly or indirectly in various stages of my work. I offer sincere thanks to the faculty members, research scholars and lab staffs for the inhelp and cooperation. I specially thank my friend Arijit Laha, and Manoj Barua who have made my staying in ISI enjoyable and helped me a lot whenever I was in problem.

30th July, 1997

(Debashis Sarkar)

Abstract

There is a natural tendency to make the VLSI design as a interconnected set of clusters of transistors, which can be utilized to extract the function from an MOS transistor network. The other requirement of verification of the circuit can also be performed simultaneously. In this dissertation, we present an algorithm for partitioning a switch level (nMOS / pMOS / cMOS) network, which may be a multicell, into several blocks and then extract the logical function from each of the individual components, followed by evaluation of the overall logic. Extension of this technique to sequential circuits is also straightforward.

Keywords :

MOS Networks
Depth First Search
VLSI Design and Verification
Biconnected Components

Contents

Chapter 1

Basics and overviews

1.1 Introduction

The integrated circuit technology has today gone so far that we cannot think of any person to design an efficient or even a working circuit, if it is a complex one, without an aid from the Computer Aided Design (CAD) tools. To reduce the complexity of the design process, it is divided into various stages. After completing the design of each stage, its operation is simulated and verified to check whether it conforms to the previous level design specification. The CAD tools assists the designer by reducing mundane but not the less important design rule checks which may range from mere width checking at the physical level to the synchronization or delay checking at the higher level. As the size of the circuit increases, this checking time also increases. A strive to reduce this design rule verification time is thus always felt. In this thesis, we address a problem of logic extraction i.e., determining the boolean expression corresponding to a switch level circuit description and designing an efficient algorithm for extraction. Logic extraction is useful not only for verification, but is also to perform simulation, test generation and timing analysis.

1.2 VLSI design cycle

System Specification

This is the very first step of the VLSI design cycle. It is concerned with the overall performance requirements of the system. For example, speed, power requirement, interfaces, can be described in system specification.

Functional Design

The behavioural aspects of the system is considered in this design. For the above example, the instruction set, the memory and I/O addressing, the interface between the different module and their synchronization is the design goal.

Logic Design

Here the logic structure that represents the functional design is derived and tested. Usually, logic design is represented by boolean expressions and finite state machines. This logical description of the system is simulated and tested to verify its correctness.

Circuit Design

The circuit representation based on the logic design is developed in this phase. The circuit design is usually done in terms of detailed circuit diagram. Here also, the circuit design is simulated and tested to verify its correctness and to reduce the complexity which might arise in the subsequent phases.

Physical Design

In this step, the circuit representation of each component is converted into a geometric representation. This geometric representation of a circuit is called a layout. The physical design is a complex process, and is split up into various substages like partitioning, placement, routing and compaction.

Design Verification

The layout is verified to ensure that it meets the system specification and the fabrication requirements. Design verification consists of **Design Rule Checking (DRC) and Circuit Extraction.** DRC is a process which verifies that all geometric patterns meet the design rules imposed by the fabrication process. Then the functionality of the circuit is verified by circuit extraction process.

As this checking is at the very last stage, so if any violation is found then all the process up the tree has to be redone. To avoid that problem normally after the design of every stage a verification of the design with the previous stage is done. Then only the next stage is taken up.

Fabrication

After the verification, fabrication is done. This consists of transferring the physical layout on the silicon wafer, deposition and diffusion of various materials on the wafer according to the layout description. This is a very complex and costly process. So it is also divided into various stages, each of which consists of mask transfer, diffusion, etching, ion implantation, deposition etc. (all may not be needed in a particular stage.)

Packaging, Testing and Debugging

Finally the wafer is fabricated and diced in a fabrication facility. Each chip is then packaged and tested to ensure that it meets all the design specifications. This stage also has various substages as connecting the I/O pads with the pin of the package, testing, and packaging into the case.

The present work is concerned with the *verification of circuit design of a switch level circuit.*

1.3 MOS Logic Circuits : An overview

The MOS (or the Metal Oxide Semiconductor) transistors are basically a switching device which can be made **ON** or **OFF** depending on the voltage applied in its gate. It has three terminals, *source* (where the majority carrier are injected), *drain* (from where the majority carriers leave the device) and *gate* (which controls the current flow). Though it can be operated in the active region also, but in digital circuits, which is our field of study, the transistor is either ON (saturate, or the current flowing through it, resulting a near zero voltage across its source and drain) or OFF (no current is flowing through the transistor and the voltage applied appears across

the drain and source terminal).

Depending on the type of the majority carrier used, MOS transistors are of two types. NMOS, where the current carrier is mostly electrons, and PMOS, where the current is mainly due to holes. The PMOS transistors are slow due to the slow carrier, whereas the NMOS transistors are nearly twice as faster as their PMOS counterpart. The structure and representation of the NMOS and PMOS transistors are given later. The CMOS circuits provides two paths from the output to various input points depending on the input combination. For example a NAND circuit is shown in all three technology. Along with these, there are several other techniques also, as pseudoNMOS etc.

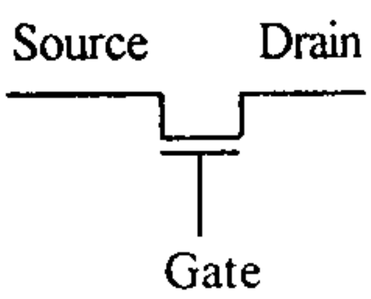
Depending on the type of transistors used the digital MOS circuits are of three types, namely

1) **PMOS** where the transistors used are of PMOS type

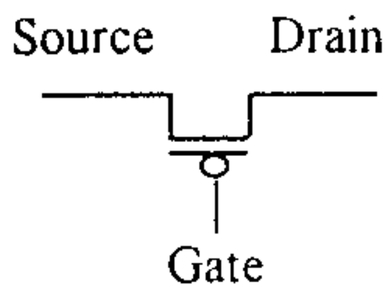
2) **NMOS** where the transistors used are of NMOS type

3) **CMOS** where both type of transistors are used.

Different technologies . . some examples



NMOS circuit representation



PMOS circuit representation

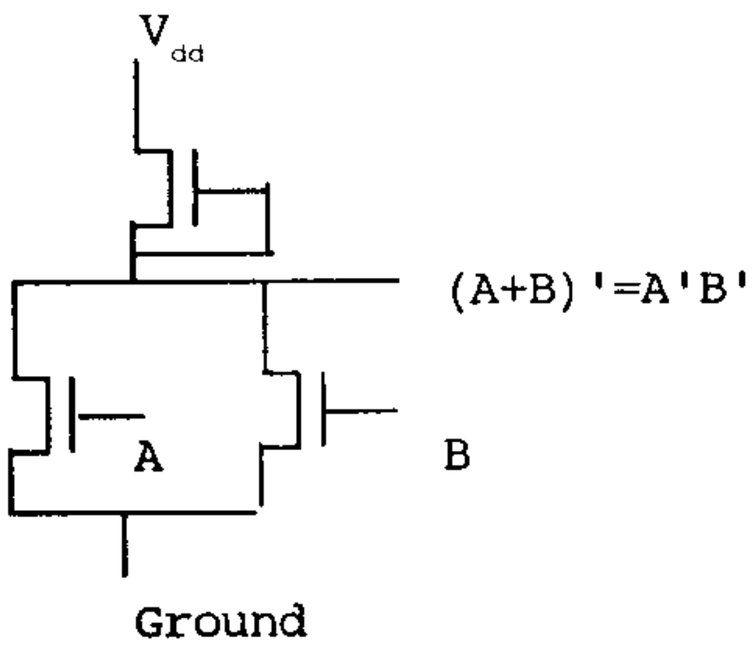


Fig. 1a. NMOS NAND gate

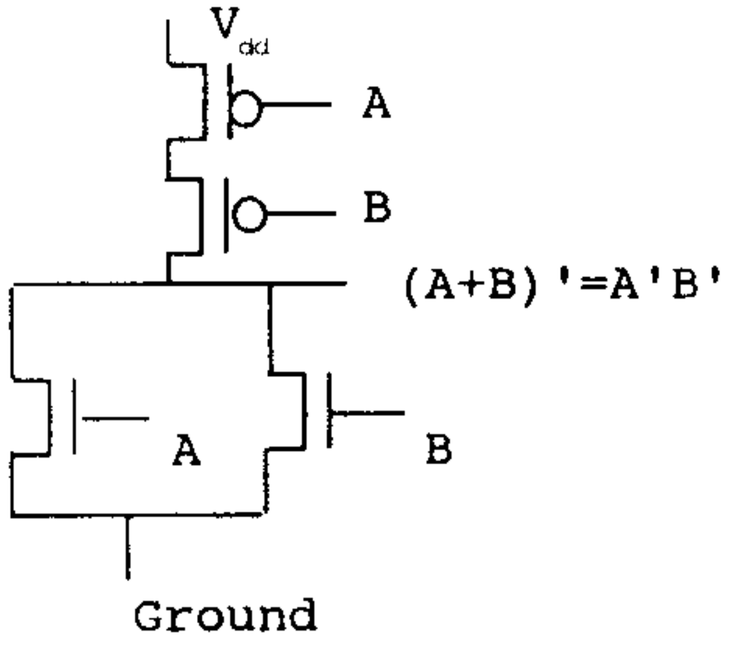


Fig.1b. CMOS NAND gate

1.3 Organization of the report

The next chapter, i.e. Chapter 2 gives an overview of the problem and its scope.

Chapter 3 describes the representation of the circuit, and various examples. We shall describe our algorithm with the help of these circuits. It also gives the format of the input file which

describes the circuit.

Chapter 4 gives a straightforward algorithm for the extraction of the boolean function and also a scheme for verification of some restricted type of circuits. This is the groundwork for the proposed algorithm, on which several modification will be done.

Chapter 5 gives the modified algorithm for the improvement of running time, keeping the constraints as it is. The algorithm for finding the subnetwork and simplifying the circuit for the following extraction and verification pass, is presented here.

Chapter 6 describes logic extraction for sequential circuits. However, it may not handle the floating bus concept.

Chapter 7 talks about the possible future works, and other areas of its application. Limitations of the proposed method and their probable solutions are also discussed.

Chapter 8 reports some case studies on which the algorithm ran successfully.

Some part of the header files are included in the Appendix to clarify any points in the discussion.

Chapter 2

About the Problem

2.1 Motivation

The purpose of the circuit design is to develop the transistor level implementation of the boolean functions designed in the logical level design. The main way to reach the goal is to replace the gate level circuits by the equivalent transistor circuits. It can be done by gate wise replacement which is inefficient but reliable, or by designing the transistor level circuit which is a direct implementation of the function. After that there can be some simplification or compaction so that the redundancy can be reduced. The design can be Automatic (using some program to replace gate or the functional blocks) or by custom (by doing it manually with the help of some CAD tools). The automatic tool's design is fast and reliable but the performance and efficiency increases in custom implementation which takes more time. For a custom made transistor level circuit the reliability depends on the designer and many problems may arise. This is the point where we want to help the designer to verify the designed circuit.

The main verification should be in the following areas :

1. Any short circuit or open circuit conditions, which is undesirable ;

2: The function is really implemented correctly ;

3: The timing delay of the circuit is according to the specification ;

4: The circuit should be testable (which is very important for the complicated and costly systems) ;

5: There should be minimum redundancy in the circuit ;

6: Area optimization ;

Some of the design goals are conflicting, so there should be a compromise among them. Here we are addressing the verification goals of reliability and the short circuit and open circuit condition testing. This includes the extraction of the function and the checking for the possible condition of output for all the input combination.

2.2 Problem Specification

Our problem pinpoints to

1. To automate the testing process of circuit design through devising a strategy for extracting the function out of the general MOS circuit from switch level description, (i.e., given a switch level circuit (NMOS / CMOS / PMOS) with arbitrary interconnections of transistors, derive the boolean expression as a logic sum of product terms.)

2. To test whether for any input combination there is a short circuit between power and ground .

3. If there exists any input combination for which the output is not connected to any other signal node, that is the combination for which the output node floating.

Basic scheme is as follows.

a. Construct the underlying graph from the given circuit.

b. Decompose the graph into several two terminal components, based on the separating pair recognition algorithm developed by Einsper, Seth, & Bhattacharya (1987).

c. Evalute the function for each of such segments by path traversal.

d. Get the overall boolean function by logical substitution of the subfunctions.

e. Check for consistency, i.e, whether there exist any input combination that causes a short circuit in the circuit, via output, or causes an open circuit for both pMOS and nMOS parts.

f. The procedure should handle passmode transistors and multicell circuits.

g. It should be extended to extract sequential circuits, i.e., accross the latch and F/F.

2.3 Definitions of some relevant concepts

As we will be working in a graph theoretic approach, so we better define some things which will be required in later stages. The knowledge of MOS transistor switches are assumed to be known.

Series transistors : If any two transistors in a circuit has only one node (between drain and source, gate is not considered) common, and the common point is not attached to any other transistor or signal then this two transistors are called series transistors. This transistors can be replaced by an equivalent transistor whose gate is controlled by the **logical AND** of the two functions applied at the gate previously.

Parallel transistors : If any two transistors has both of there nodes, source and drain common (here we are not discriminating here between source and drain), then they are said to be in parallel. These transistors can be replaced by a single transistor, whose gate is the **logical OR** of the previous two gate functions.

Path : A path in a graph is a sequence of nodes (u,v,....w) where no nodes appears twice and there exist an edge between any two consecutive nodes in the sequence. Node u will be called the startnode, and node and node w will be called the end node.

Series-Parallel edges : In an undirected connected graph with two specified node, one as source, from which all paths emerges, and another as sink, at which all path ends, an edge (u,v) is

called a series-parallel edge iff every path passing through the edge (u,v) has the same order of u occurring first and v following it or vice versa.

Non-Series-Parallel path. The non-series edge (u,v) is such an edge in the aforesaid graph, so that there exist at least two paths , from the source to sink, so that in one u follows v, and in other v follows u.

Passmode transistors. The passmode transistors are those transistors where one node is connected to a signal node which may take any value of 0 or 1. (i.e., other than power and ground).

Modelling of the problem

The circuit is modelled as a undirected graph . Only the required part of the circuit is to be searched and the corresponding graph is to be formed. Each node in the graph represents a junction point in the circuit or an input of the circuit. Each edge in the graph has a controlling function associated with it which represents the gate variable in the corresponding transistor in the circuit .It can also be thought that the gate node is connected to a node the signal value of which is controlling the transistor. Any node can be associated with a signal which is basically the output at that point, or any input signal applied at that point. So our problem boils down to find the function associated with a particular node which is specified as the output node.

Difficulties

At first sight the problem may seem to be a direct one. Try to find out all the possible combination for which the output node is connected to the supply line or ground or any other signal line, & for that combination the signal value at the output node will be the connected rail or signal. But at second thought we can see that after finding a path between output and one signal node, we have to know the combination for which the specified path will be connected or short. But it is not always trivial. This requires some tricky assumptions and iteration over the circuit.

The next part is the verification. We can get all the combinations for which the output is connected to at least one of the input signals, which may be power or ground also. So we can easily find out the input combinations for which the outputnode is not connected to any of the input signal or power rails, detecting the floating condition. But for the short circuit , it is not that easy. We can easily find out the combination where the output gets affected due to the short circuit, but for those combinations where the output is not affected, and we still don't have the answer.

As it is not possible to extend the verification to those paths all of whose nodes are not reachable from the output without passing through a signal node. Hence we cannot test those paths also.

2.4 An example of the CMOS simple circuit.

Now give an example of an CMOS circuit which we will use for the clarification of the algorithm. Nodes are marked for clarification of representation.

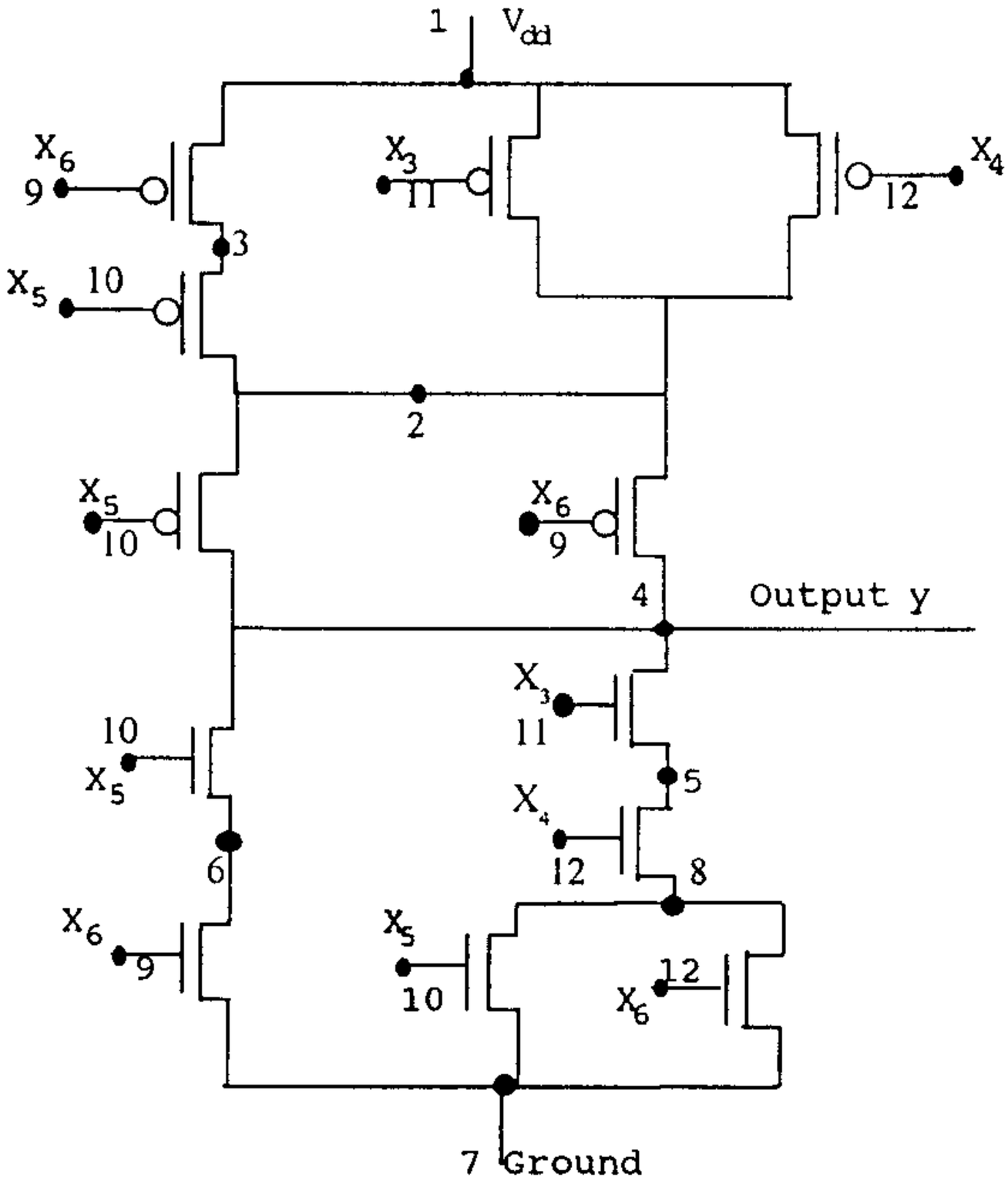


Fig.2 The evaluated function is $X_3'X_5'+X_4'X_5'+X_3'X_6'+X_4'X_6'+X_5'X_6'$
i.e. function = $((X_6+X_5).X_3X_4+X_5X_6)'$

Chapter 3

Tell me the Circuit

This Chapter is about how to input the data about a circuit and how we are going to represent it in the computer. This input part can easily be modified and make the interface as per the requirement of personal use. The goal of our representation is to keep it easily understandable from the file format.

3.1 Circuit Representation and Input format

The circuit representation will be evident from the header file listing. The input format is explained after the listing. The classes MyList and MyArray can be thought as a list and array respectively.

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//// CIRCUIT .H THE DEFINITION OF THE CIRCUIT & TRANSISTOR CLASS///
////////////////////////////////////////////////////////////////////////////////////////////////////
#ifndef __CIRCUIT
#define __CIRCUIT
#include <iostream.h>
#include "graph.h"
#define PTYPE 0
#define NTYPE 1

class node {
private:
  int number,  // this the index in the array in the circuit
      nooftrans,  // to remember how many transistors are connected
      mark,     // for identifying the node as traversed
      evaluated,  // to signifyied that the function associated is a valid one
      signaltype,  // this is the no of variable associated with,
      signalvalue; // this is 0 for inverted signal, 1 for in the normal form
  MyArray<int> transistors;  // the array of indexes of the connected transistors
  Termlist t;      // the function associated with the node, valid if evaluated is 1
public :
  node();
  void  Print() ;  // for printing the node no and the associated transistors and circuit
  friend  class Circuit;
  friend  class Transistor;
} ;
```

Continuation of classes Circuit, Transistor & Node

```
class Circuit;
class Transistor{
private:
    int source,drain,gate,  // the source,drain,and gate node no
        type,   // the type may be PMOS or NMOS, may incorporate pull up transistors
        number,   // index in the entry in the array of transistors in the circuit
        mark;    // used to denote that it has been processed
public :
Transistor();
void SetType(int i);
...
...
friend class Circuit;
};

class Circuit {
private:
 MyArray<Transistor> TransList;  // this the total transistor entry of the circuit
 MyArray<node> Nodes;   // this the total node information of the circuit
 MyArray<int> Outputs;    // this keeps the information about the output to be evaluated
 unsigned int PresentSignalno;  // this is the next variable number to be used, discussed later
public :
 Circuit();
 int ReadCircuit();    // reads the input from a file
 int WriteCircuit();    // writes the circuit information in a file
 Graph *BuildGraph(int);    // function for creating the graph for a specified output node
 int AddtoGraph(int,Graph*); // recursive function for  the graph building procedure
 Termlist* ExtractFunction(int);  // function for extracting the function associated in
                                  //  a particular node.
 int OutPut(int i) ;  // returns the ith output entry.
};

#endif
```

The input file format is given in the next page. The circuit is kept as the node array, and the input signal nodes and the signal applied at that point. This will restrict the program to arbitrarily evaluate the function. Each node has an array of transistor numbers, which will indicate the transistors drain or source node is attached to the specified node. The circuit also has an array of transistors, which will help the program to access the transistors quickly. As this array has a dynamic size, so it is better to have an idea of the no of the transistors and nodes. As the array will always accommodate the largest numbered entry, so it is better to number the transistors from 1 to #transistors. The circuit also maintains an array of output nodes along with no of output nodes. For future use it also keeps track of temporary nodes marked as input nodes.

The Input file for the circuit of figure no.2

The format is written in the side.Obviously those comments were not in the file.

```
0 0 0      // false node, may not require to give this
1 18 0     //  Node number 1, Connected to Vdd, 18 is reserved for
3 0 0      // Power supply,
4 0 0
5 0 0
6 0 0
7 17 0
8 0 0
9 6 1     // node 9 is connected to input signal no 6, and it is in
10 5 1    // normal form, last entry 0 denotes the inverted signal
11 3 1
12 4 1
-1         // end of node information
1 1 3 9 0     // Transistor information, first entry is the transistor
2 3 2 10 0  // number, then the source node, then the drain node,
3 2 4 10 0  // then the gate node, after that there is transistor
4 6 4 10 1  // code, 0 for PTYPE, 1 for NTYPE, and
5 7 6 9 1
6 1 2 11 0
7 1 2 12 0
8 2 4 9 0
9 5 4 11 1
10 8 5 12 1
11 7 8 10 1
12 7 8 9 1
-1              // end of transistor list
4 -1            // output node list
```

The input format is explained above. This circuit has a no feedback in the gate and there is no passmode transistor. The only signals connected are the power supply and the ground. While reading the transistor list, the transistor numbers are kept in the associated with the node. The circuit has a list of transistors, and list of nodes.

Chapter 4

The First Approach

Assumptions

The problem is related to the extraction of the logical function implemented by the MOS transistor circuit. The circuit may contain a number of NMOS and PMOS transistors. Moreover there may be some signals which are connected to the output controlled by some transistors. For the simplicity we will assume that there is no feedback in the circuit and the circuit is not a sequential one. By feedback we denotes all those connections where there is a path between the gate and the source(drain) through the source or drain of other circuit.

Proposed Solution of the Problem

In the first approach we can proceed is that to find all the paths from the output to every input signal, including the Vdd and Gnd also, so that from those paths of the individual input signals we can find out the ultimate function.The steps in this approach are

a) To build the graph

b) To find all possible paths to individual signal nodes

c) To find the function from the path information

d) To verify the short circuit and open circuit conditions

4.1 How to build the corresponding graph

The first problem is to build the graph from the circuit. For this we start with the output node and do a DFS like traversal of the circuit where the terminal nodes are input signals. The transistors are converted to edges and the junction points are converted to vertices. If any gate node is controlled by any other output which is not already evaluated then we try to evaluate that node first, temporarily suspending the previous node. So we are traversing only those transistors which are relevant to the extraction of this output. Any transistor is tested only two times, one for the source node , and another for the drain node. The nodes are traversed only one time. Any node which is declared as the input signal node is entered in a list of signals. The Power supply and ground node is also considered as a signal node, having value 1 and 0 respectively.

The resulting graph has some nodes and edges, where the edges are associated with some function which defines the required combination for the edge to be short. It also has a list of nodes and associated functions which defines those signals are applied to the corresponding node. By the time we finish the building of this graph we have evaluated all the necessary nodes which are controlling the transistors of the required part of the circuit. Once one node's logical function is evaluated, it is marked and the evaluated function is used later.

The Complexity of creating the graph is proportional to the no of transistors present plus the no of nodes traversed.

4.2 Algorithm for the creation of the required graph

This algorithms is for creating the graph from the given circuit. The function AddtoGraph can call BuildGraph through another function ExtractFunction to evaluate the gate node signal. The function AddtoGraph() is the recursive function that does the job of adding a node and exploring all the transistors associated with the node.

This is the calling function to build the graph from the circuit.

Input : A circuit and a node in that specified for the evaluation

Output : A graph corresponding to the part of the circuit required for the evaluation.

```
Graph* BuildGraph (node nd )
{  Graph *grf;
   Mark the node nd as the Startnode for grf.
   AddtoGraph(nd,grf);  // adds the node and its relevant part in the graph
   Unmark all the nodes and Transistors used in this process.
   return grf;
}
```

The function **AddtoGraph()** is the basic recursive part of the graph building function.
Input : A node in the circuit which is to be added in the partially built graph for a output node.
Output : Augments the graph by adding the supplied node and its related nodes, which is not already added.

```
int  AddtoGraph(node nd, Graph *grf)
{ Start at the node nd.
  Mark node nd.
  for all outgoing unmarked transistor do
  { mark the transistor.
    if the gate node function is already evaluated
     { add an edge in the graph grf between drain node and source node
         the corresponding function is the gate function
      }
    else
    { gate controlling function = ExtractFunction(gate_node)
      add an edge in the graph grf between source node and drain node
      with the controlling function
    }// remember that the combination for this function will short
      // this transistor, so for PMOS the function should be inverted.
   if the other end of the transistor is not marked
          AddtoGraph(other_end,grf);
  }  // for each unmarked transistor
}
```

4.3 How to find all the paths

First we have to clear ourselves why we are looking for the paths. The requirement lies in the switching concept. If we can identify all the possible paths, we can associate an input signal combination for which those two points will be connected. If these two points are the output node and a signal node then for that particular combination the input signal will be available in the output. This way the output will be the combination of these outputs for all the input signal nodes.

To find all the paths, we define one function which, from a node in the circuit, will look for all possible paths to a particular node, called **endnode.** This will return the input combination for which at least any of these paths exists, if at all exists. The algorithm is given bellow. The gist is to try to proceed through all outgoing edges to the endnode avoiding any previously traversed node. If it can be done then the path from this node to endnode will be the summation of individual function with the transistor controlling function in series, i.e., if we have to go to endnode from startnode, and startnode has three outgoing edges to v1,v2 and v3, and v2 is only marked, then let v1 has a path from v1 to endnode for function T1, v3 has T3, and the edges (startnode,v1) and (startnode,v3) has controlling functions f1 & f2 respectively, then the path from startnode to endnode will be f1.T1 + f2.T2.

For avoiding loops in the path, while entering a node we are marking it.For the next step we will proceed towards a unmarked node. But while returning from one node it unmarkes the current node because there may be a next path which passes through this the current node but doesn't uses some of the nodes used in the previous path, which may be utilised now. This explores all the possibilities, but make the algorithm to run in exponential time with the number of nodes.

The algorithm for finding all the paths possible is given in the following section.

Algorithm for finding all the path from startnode to endnode

Input : A graph, and two nodes in it.
Output : A boolean expression for which there will be at least one path between the two nodes.

```
FindAllPath(node endnode)  // for node st say
{ Termlist t=NULL;        //  t  is a bollean expression.
 mark the current node;   // so that it is not used in a path more than once
 for all the outgoing edges e = (st,ot) and ot not marked do
  { if (ot = = endnode) t = t + e.Function;   // there is an edge to the target node
    else t = t + e.Function * ot.FindAllPath(endnode); // try to proceed from ot
    }
 unmark the node st;
 return t;
}
```

4.4 Finding the total output function.

The extraction part is done in the following way.

1. First for each input signal find the possible combination for a path to that node, say for I1 it is T1, for I2 it is T2 etc.

2. If the signal is not associated with ground, then multiply the path function with the input signal. The sum of all these product term will give the function output at the required node.

4.5 How to verify the short circuit and open circuit conditions

The verification process for short circuit is done in the following way

1. First for each input signal find the possible combination for a path to that node, say for I1 it is T1, for I2 it is T2 etc.

2. Then find all the combination, for which a path will exist in between two input signal, e.g., between I1 and I2 it is T1* T2.

3. Now if the existence of path has a combination which makes the corresponding signals opposite values, i.e.. 1 & 0, then for this combination input signals I1 and I2 will be shorted.

For the verification of the floating output condition we are doing the following algorithm.

1. First find out the combination for each signal as in the first case.

2. Then add up all the functions, keeping in mind that they are not to be multiplied with the input signals. The sum will indicate the combinations for which it will be connected to at least one of the input signals.

3. Now check whether any combination is left. If exists, it will be the combination for which the output floats.

Limitations.

The limitations of the checking for short circuit is that it cannot detect any short circuit combination which doesn't affect the output node. Testing for those combination in this frame-work is costly as that can be the finding all paths between any two input signal node.

The relevant part of the output the program for circuit in figure no 2.

Enter the filename from read : circ2.dat
Creating a Vdd connection
Added one Ground term
Compacting in 8

continued ..

Continued output for the circuit in figure 2.

```
Compacting in 7
Compacting in 1
Compacting in 4
Compacting in 2
Compacting in 2
Graph Number 0
Start vertex is 4 & End vertex is
Vertex No : 8
Edge between : From 8 to 7  The term is : x5 + x6
Edge between : From 8 to 5  The term is : x4
Vertex No : 5
Edge between : From 5 to 8  The term is : x4
Edge between : From 5 to 4  The term is : x3
Vertex No : 7
Edge between : From 7 to 8  The term is : x5 + x6
Edge between : From 7 to 6  The term is : x6
Vertex No : 6
Edge between : From 6 to 7  The term is : x6
Edge between : From 6 to 4  The term is : x5
Vertex No : 1
Edge between : From 1 to 2  The term is : x3' + x4'
Edge between : From 1 to 3  The term is : x6'
Vertex No : 3
Edge between : From 3 to 1  The term is : x6'
Edge between : From 3 to 2  The term is : x5'
Vertex No : 4
Edge between : From 4 to 5  The term is : x3
Edge between : From 4 to 6  The term is : x5
Edge between : From 4 to 2  The term is : x5' + x6'
Vertex No : 2
Edge between : From 2 to 4  The term is : x5' + x6'
Edge between : From 2 to 1  The term is : x3' + x4'
Edge between : From 2 to 3  The term is : x5'

Input signals are as follows...
Node no 1  ... Input Signal is  1
Node no 7 ...  Input signal is  0

No of Signals are 2

Looking for path from 4 to 1
The term is : x3'.x5' + x4'.x5' + x3'.x6' + x4'.x6' + x5'.x6'

Looking for path from 4 to 7
```

The term is : x3.x4.x5 + x3.x4.x6 + x5.x6

The function got in the Extract function is ...
The term is : x3'.x5' + x4'.x5' + x3'.x6' + x4'.x6' + x5'.x6'

The Extracted Function is as follows as in main
The term is : x3'.x5' + x4'.x5' + x3'.x6' + x4'.x6' + x5'.x6'

For the algorithm FindAllPaths to work correctly it is required that there should not be any **parallel edges between** two nodes. For this we run a parallel edge compacting function after **creating the primary** graph. This will replace parallel edges with an edge controlled by the parallel **equivalent function** .

This algorithm has complexity in the exponential range and it is mainly due to the back and **forth propagation** of the path exploration scheme. **This algorithm** also tries all possible outgoing **edge to find the** destination vertex, thus roaming **many edges which** is not at all required. This **deficiencies are** tried to solve in the next algorithm by reducing the no of equivalent edges.

Chapter 5

For a little less...
The Improvement over the first Approach

5.1 Scope of Improvement

The main time consuming part of the algorithm **FindAllPath** is the recursive call of FindAllPath(endnode) for various nodes. The nodes away from the output node will execute this function many times than the near one, where the near nodes will take larger time to execute. This process is necessary for finding all the possible paths.

If we take a second look at the graphs and the paths, we can see there exists some subnetwork which is connected to the graph at two points. So if we can replace this subnetwork by a simple edge controlled by a boolean function, then we need not roam within the subnetwork many times. Thus the required edges will be reduced and the overall performance will be better. Now we present an algorithm to find out the separation pairs and the associated subnetwork. The algorithm is based on the *depth first search* and *segmentation* of the graph. This increases the running overhead for the algorithm, but if the no of edges and nodes are large then it can be shown taking smaller time than the simple and direct one. This algorithm will run in order of n time, where n is the no of edges, but the improvement of the following verification algorithm depends on the no and size of the subnetworks present in the circuit.

Once a subnetwork is identified, it is deleted from the original graph and a simple edge equivalent to the subnetwork is added to the graph between the separation pair. If the subnetwork does not include any signal node other than the two in the pair, then it doesn't hinder the primary objective of finding the output function by finding the paths as no signal node is lost. But if the subnetwork has any signalnode, then we cannot replace the subnetwork as it will delete at least one signal node.

Now we present the algorithm for finding the separation pair and corresponding subnetwork. This algorithm can handle parallel, series, and non-series-parallel edges also. For finding the equivalent edge-function we are using the previous path finding algorithm from one vertex of the pair to the other.

Before going to the algorithm we should define some terms which will be used frequently.

5.2 Some Graph relevant definitions

Form now onwards we will forget the circuit representation as the transistors and junction points along with the input signals. We will talk about the underlying graph, where the junction points are vertices or nodes, and the transistors are edges, may be thought as weighted, which is the controlling function. We will define the graph as (V,E) where V is the set of nodes and E is the set of edges, directed or undirected.

Biconnected component: A biconnected component of a graph is a subgraph of the graph such that for any two nodes in the subgraph, there exists two node disjoint paths between them.

Cycle : A path which starts and ends at the same node is called a cycle.

Separation pair. If there is a connected component of a graph which has exactly two points common with the parent graph then the common points are called the separating pair. The speciality of the separating pair is if any path starting from a vertex and ending at a vertex ,both of which are not in the aforesaid connected component, and if it enters through any one vertex of the separating pair then it must leave through the other vertex of the pair.

Tree: A connected graph without any cycle is called a tree.

Rooted tree : A tree in which a node , say r, is singled out as the root is called a rooted tree.

In a rooted tree we can always assign directions to the edges away from the root. In such a directed tree, a node v is called **ancestor** of the node w if there exists a directed path from v to w. In that case w is called a **descendant** of the node v and w and v are **lineally related**. The lineal relation defines a partial ordering of the set of vertices of the tree.

Lineal numbering: Let (T,r) be any rooted tree with vertex set W. A lineal numbering of (T,r) is a bijection LN: $W \leq \{1,2,...|W|\}$ that is compatible with the lineal order of T. That is, $LN(v) \leq LN(w)$ whenever $v \leq w$ is in the lineal order in T.

A Subgraph $T = (V,E_T)$ is a **Spanning tree** if T is a tree. E_T is the set of directed edges.

A rooted spanning tree (V,E_T,r) of G is said to be lineal if for all edge $(s,t) \in E\text{-}E_T$, s and t are lineally related. Such an edge is directed from s to t, if $t \leq s$, in the lineal order. Then (s,t) is called a **backedge.** Let $E_B = E\text{-}E_T$. The subgraph of G with edge set E_B is called the backedge graph B. Let (T,r) be a lineal spanning tree of G and B be the backedge graph. Suppose that we have a lineal numbering LN of the vertices of G, i.e. a numbering consistent with the lineal order induced by (T,r). For any edge e = (a,b) in E_T, let R(e) be the set consisting of LN(t) where (s,t) is a backedge, and s is a descendant of b including b, i.e., R(e) is the set of LN(w) where there is a path from a to w through zero or more tree edges and the last edge is a backedge. Now for all edges $e \in E$, we define LOW1(e),LOW2(e) (the first and the second low points of e) as

a: If $e = (a,b) \in E_T$, then :

$LOW1(e) = \min \{LN(b)\} \cup R(e)$, $LOW2(e) = \min \{LN(b)\} \cup (R(e)\text{-}\{Low1(e)\})$

b: If $e = (s,t) \in E_B$, then $LOW1(e) = LN(t)$, $LOW2(e) = LN(s)$;

Now we can give the procedure to produce this lineal numbering of the nodes and calculate the low points of the edges. Along with this we can find the biconnected component of the relevant node so that we can easily identify those nodes not to be considered in the later path finding stage, for two given nodes. This also helps to locate the transistor which can contribute nothing in the

circuit extracting procedure for the given nodes.

All the header files for the class Vertex, Edge, dVertex (Vertex in the directed graph) and dEdge (edges in the Directed graph) are given at the appendix. So for any clarification please refer to the listing in the appendix.

5.3 Algorithm for doing the DepthFirst Search

The algorithm DFSearch is many used algorithm, so we are not explaining vividly. A stack is maintained to keep track of the biconnected component.

DFSearch(Edge e=(u,v))

```
{ dfsnum=dfsnum+1;
 LOW1(e)=LOW2(e)=LN(v)=dfsnum;
 push v in the stack.
for each edge e'=(v,w) do
 { if (LN(w)==0)
    { add e' to E_T,
     DFSearch(e');
     AdjustLowPoints(e,e');
     if ( LN(v) = = LOW(v) )   // a biconnected component detected not required
       { pop all the vertices in the stack and there associated edges
         untill there is a vertex z such that LN(z) < LN(v);
         push v in the stack.
       }
     }
   else if(w≠u)   // not the edge by which we came to this node
    { add (v,w) to E_B; // that is mark the directed edge as backedge
      LOW1(e')=LN(w);   LOW2(e')=LN(v);
     AdjustLowPoints(e,e'); // that is the corresponding dEdges
     }
}

AdjustLowPoints(dEdge e,dEdge e')
{
  if ((LOW1(e')<LOW1(e))
     { LOW2(e)=min {LOW2(e'),LOW1(e) };
       LOW1(e)=LOW1(e');
     }
   else
    { if (LOW1(e') == LOW1(e) )
        LOW2(e)= min { LOW2(e), LOW2(e') };
      else
        LOW2(e) = min { LOW2(e), LOW1(e') };
    }
}
```

We have to keep in mind that in our implementation, we are creating another graph from the original keeping the edges of any one direction from the two. So we have to manipulate many things to keep track of the correspondence.

For the time being we will forget the original undirected graph and work on the directed graph, created by the process of the depth first search algorithm. So any reference to the graph will denote the directed graph after the DFS, unless otherwise specified .

We associate with each directed edge e = (a,b) a triplet (a,LOW1(e),x) where x = 0 if LOW2(e) < a, or 1 otherwise. This is done because we want an ordering among the edges going out from a node. The ordering is such that if we choose an edge earlier in the list then we can go to smaller numbered nodes, at least as small as the other later edges in the list lead to.

Now we will define the path starting from any edge e. A **Path(e)** is constructed as follows. The edge e will be the first edge of the path. If e is not a backedge, let e' be the first unused edge in the outgoing list of b (e = (a,b)). Add e' to the Path(e) then repeat the process until e' is a backedge. The ordering of the outgoing edges in the list in each vertex is such that this will lead the path to the vertex of minimum lineal number possible with the last edge as backedge. Let us denote this path as Path(e) = $(e_1,e_2,...e_k)$ where e_1= e and e_k is a backedge. Note that e_1 may be equal to e_k.

Now we define another concept called **Son(e).** It is the list of all edges coming out from the nodes of the Path(e), but not used in the Path(e), except the start node and the end node of the path where the nodes will be considered in the reverse order it was added .That is the last (not the end node) will be considered first and its remaining edges will be added at the start of the list, and the next to the start node will be considered last.

All the path will have a **Segment List** which is basically the paths starting at the Son(e) list. We will call it the PathList(e) also. The Segments **Seg(e)** is defined as follows.

Seg(e) = All the Paths Path(f) and its Segments where edge f is in the son list of the path(e).

For the algorithm for identifying the separation pair we will identify one edge which is the starting edge.

The Path(e) has the following property.

1. **Path(e).Tail** = The sating node's number (i.e., the lineal number)

2.**Path(e).LOW1** = The low1 value of e

3.**Path(e).EdgeList** = The list of the edges present in the path(e)

4.**Path(e).PathList** = The list of the paths directly going out from the path(e).

6.**Path(e).ENUM** = The no of edges present in the path and its segments.

7.**Path(e).Father** = The edge of the parent path from which the present path **has emerged**.

Now we just present two Lemmas [2] .

Lemma 1: Let X < Y (i.e., X is generated before the generation of Y) be **segments in** Path(e).PathList. Then X and Y are directly linked relative to Cycle(e) (i.e. , the cycle in which e is an edge) iff X has a backedge (s,t) so that LOW1(Y) < t < Tail(Y).

This lemma proposes that as X < Y, then for X and Y to be within a separation pair there must be a backedge in X which goes to a node in the parent path between the lowest point and the highest point of the segment Y. Otherwise the segment X and Y are in different separation pairs.

Lemma 2 : Let X,Y be the segments in the Seglist of path(e),If X bridges Y but Y doesn't bridge X then X < Y.

This lemma puts an order in the creation of the paths/segments for any path(e). Thus we can be sure the ordering of the generation of the segments, unless they are irrelevant to the identification of separation pairs.

5.4 Detection of Separation Pairs .

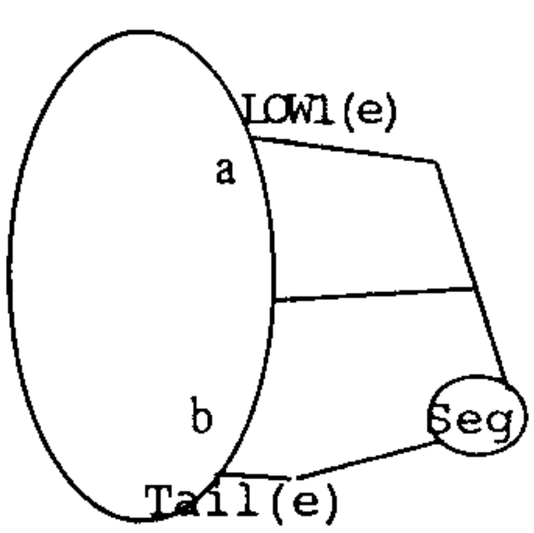
The separation pairs will be detected in the recursive algorithm **BuildPatr**. We can divide the separation pairs (a,b) of G into three types :

1.**Type 1 pair :** This corresponding to the TAIL and LOW1 points of a segment that has only two point in common with the parent cycle. If this segment is Seg(e) then LOW2(e) = Tail(e).

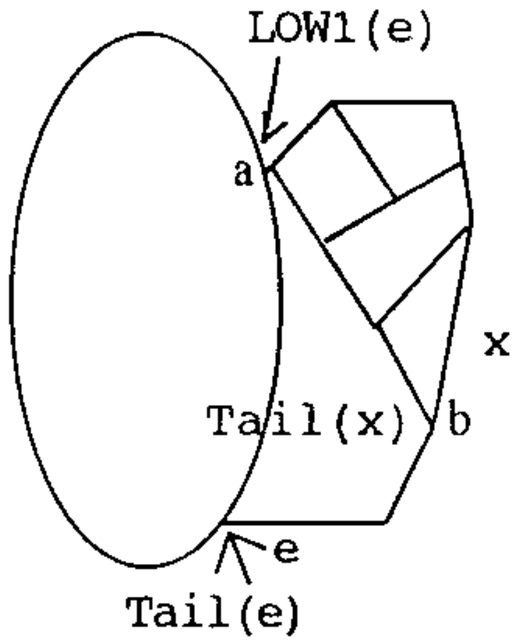
2. Type **2A pair :** This is the LOW1 and LOW2 points of an edge with no internal segments. Let x be the tree edge on Path(e) going out from the LOW2 point as in the figure, then LOW2(x) = Tail(x). So this type of separation pairs can also be checked in the same way as the type 1 pairs.

3. Type **2B pair:** This is Tail and LOW1 points of a component without internal segments as in the picture. This the most difficult type to detect.

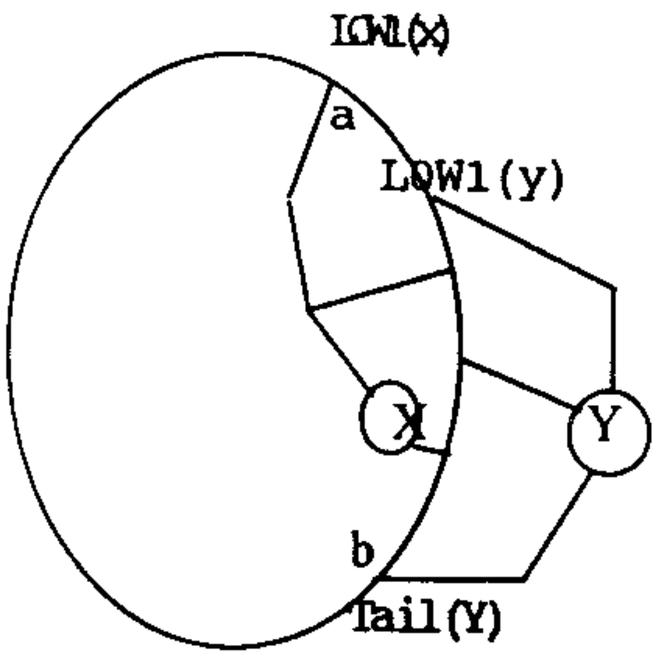
In the following figures , the separation pairs are (a,b).



Type 1 Pairs



Type 2A pairs



Type 2B pairs

For detecting the Type 2B pair, we are taking several steps. The first one is to associate a number with each node which denotes the lowest number of all the path number which is coming in at that vertex. This is done by the DFSearch() while assigning Low numbers to nodes.

The next step is to keep track of the lowest point which is the last point of the current path, we keep a local stack in each call of the BuildPatr. The stack has two items, LS or the path number of the earliest partially built segment, and the LOWC, i.e. the current low point of this component. When the minimal component is detected the corresponding separation pair will be (LOWC,TAIL(LS)).Let us call the stacks for LS as LSatck, and for LOWC as LOWStack. Just keep in mind that the pops or pushes in the stack will be simultaneous.

The stacks are updated as follows.

1. When a new segment (here a path) with edge x is started, we check if there is any entry in the LOWStack such that LOWC > LOW1(x). If exists , it denotes that the new segment will go beyond the present lowpoint, so it is not required. We pop all such entry from the satcks. Now if LOW1(x) ≥ Tail(e) then the current segment is not going to the parent cycle or path , so it can result in a separating pair one of which is LOW1(x). So if LS is i.e. last entry poped from LStack then push LS and LOW1(x) in the stacks. If no entry was deleted, then push SEG(x) in LStack, and LOW1(x) in LOWStack.

2. When we backtrack past an edge x on Path(e), any entry which has LS>FirstPath(Tail(x)) is no longer can give a separation pair, because there is another path coming from further away. So Pop all such entry.

The total no of entries in the various stack is proportional to the number of generated paths, so it is $O(V + E)$ [2].

Now for finding the all the separation pairs in the path from one node the output, to another node which is the input to be considered, we add a new directed edge from the input node to the output node, and mark it as the first node. This is done so that the components found will be related to both this nodes. All other edges which are not in the same biconnected component in which the starting edge is, will be discarded at this stage. For a large circuit with various input signals, specifically if input signal is connected to the output node in separate biconnected components then it will stop the roaming of the algorithm FindAllPath() in the nonrequired edges. This is described in the figure in a better way.

5.5 Algorithm : RunSeparation()

Input : A graph having a specified startnode and endnode
Output : A graph with two edges and two nodes. One will be the equivalent of all the paths possible between the two nodes in the graph.

RunSeparation()

```
{ for all v ∈ V do v.FirstPath=MAXINT;  // a large number, representing ∝
  Add an edge from the end node to the startnode, and mark it as startedge
  BuildPatr(startedge);
}
```

We are assuming LS is the top entry of LStack, and LOWC is the top entry of LOWStack. The Stacks are the last in and first out type data structure implemented by singly linked list. The only visible element in the stack is the last entry. The function BuildPatr() is a recursive function which creates the paths and detects the separation pairs.

```
Algorithm BuildPatr(Edge e)
{ Create the path Path(e).
// Fix LOW1 and Tail of Path(e) and FirstPath(LOW1(e)) also
 for a ∈ Path(e).Edges  // list creation was in the reverse order
 { t=Tail(a); low2 = LOW2(a);
   // a separation pair extraction may delete this information, so keep it
  while top entry of LOWStack = t    // the partially built component goes at most to t
  { // one separation pair of Type 2B, LS.Tail and LOWC
   TYPE2B(Current Path,LS,LOWC);
   If this results any series edges, combine in into a single edge.
   Delete the top entry in the Stacks.
   }
  if a is not a backedge then
   { if possible replace any series edge
     else
    if (low2≥ t) // a separation pair of a.Tail and LOW1(a).
        TYPE1_2A(path(e),a);
    }
  pop from stacks any entry if LS>FirstPath(t);
   // all processing has been done for the detecting separation pair, now do the
   // pathfinding part for the unused edges going out
  for all unused edges f going out from t do
   { BuildPatr(f);
     if ( Path(f).Enum==1 )
// it may be parallel to any previous edge or segment or path, so try to find and combine.
     if CompParallel(Path(e),e,Path(f)) succeeds then skip the rest of loop
     add Path(f) to Path(e).PathList.
     Path(f).Father=a.
     Pop all the Stack entry with LOWC > LOW1(f).
     if (Path(f).LOW1 ≥ Tail(e) then
     if LS is the last Pop from LOWStack then
       push LS and Path(f).LOW1 in the stacks
     else push Path(f) and Path(f).LOW1 in the stacks
  }  // all edges and paths have been explored, so update the no of edge information.
path(e).Enum = Sum of all the Path(f).Enum in the PathList
}
```

Remarks :

The function **GetPath(e)** simply proceeds through the unused edgelist until it gets a backedge. It then number the Path and evaluates its Tail, LOW1, and ENUM. It keeps the list of edges in such a way that the last edge comes first, and the first edge comes last in the edgelist. It also

assigns FirstPath for the node Path(e).LOW1 to path(e).number if it is the lowest numbered path ending at that node.

The function TYPE12A deletes all the Paths and it children from the current pathlist and the part of the present path that is between the Tail(e) and LOW1 point. This function also add an edge in replacement of this component. It first makes a new temporary graph with the separation pairs as the start node and the end node. Then transfer all the segments in the path Path(e) to the new graph. Then it transfers the part of the path from Tail(e) to PATH(e).LOW1. It then runs FindAllPath() in the newly created graph to find the equivalent function. Rest is the creation of a new edge in the original graph and associating the function evaluated with it. The edges transfered to the new graph is deleted from the old one. The nodes, except the separation pairs are also deleted from the original graph.

The function TYPE2B also does the more or less same thing but it considers the Path(e).PathList and selects all the segment which has number greater than LS and LOW1 points greater than LOWC. As in n case of the previous procedure (TYPE12A), all the edges transfered are deleted, and a new edge is added. Both of the functions uses TxPath (Path FromPath ,...) to delete the path from the graph. It also has some other parameters due to implementation details and equivalent function evaluation for replacement.

The function CompParallel(Path (e),Edge a,Path(f)) tries to find if there exists any tree edge, backedge or path of length 1 parallel to this edge of Path(f), and returns 1 if succeeds.

Though this algorithm has not been tested in the circuit extraction procedure but it has been tested separately, and it ran successfully on a complicated graph and extracted the function for the equivalent circuit for only one output node and signal node. This can easily be implemented, taking care of the original graph modification problem.

5.6 The Improvement of running time for this preprocessing algorithm.

1. This will only consider the biconnected component of the whole graph, which is relevant to the corresponding input signal. Thus for a large CMOS circuit it will not consider any PMOS for Ground connection, and any NMOS for the POWER connection, thus greatly improving the performance of FindAllPath(). This will stop all the unneccesary traversing through unrelated nodes.

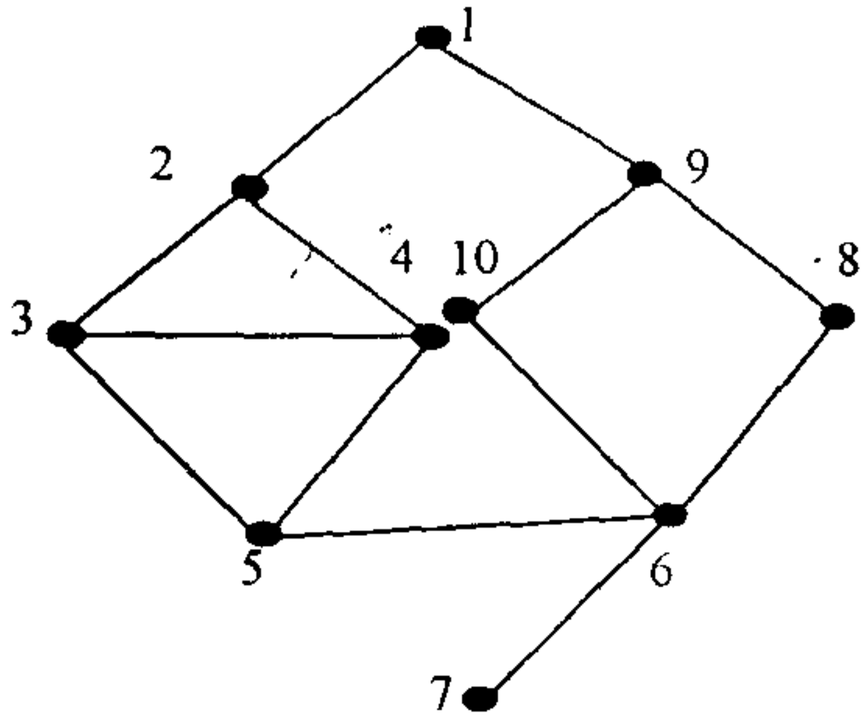
2. This algorithm will compact all possible subgraph which can be replaced by a single edge, thus improving the running time of the next pass by reducing the no of edges (equivalent transistors).

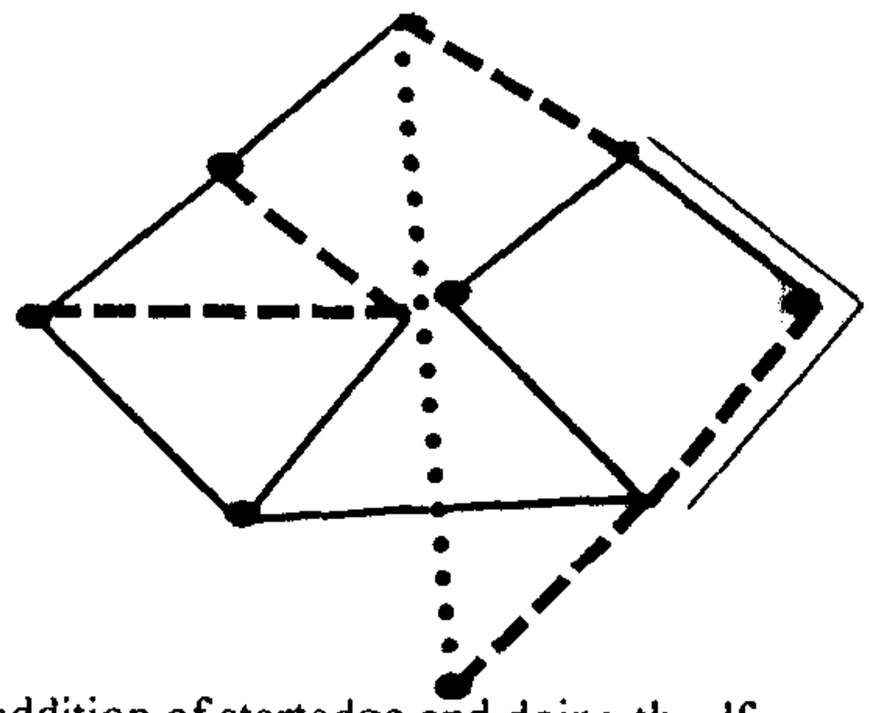
3. The algorithm can identify any transistor which is not at all required in the circuit (probably one whose one node is floating or a subgraph which is connected to the rest of the circuit at a single point,). Those extra transistors can be removed, or it may be due to some forgotten connection which is to be made.

4. All the series and parallel compaction can be done in a single pass, thus avoiding the procedure of iterating over series and parallel compaction on the circuit.

The result of the separation pair extraction over a graph is shown. Though we are not showing the equivalent function evaluation, but it was done by calling the FindAllPath() function between two separation pairs. The edge is replaced and may be included in a subsequent separation pair.

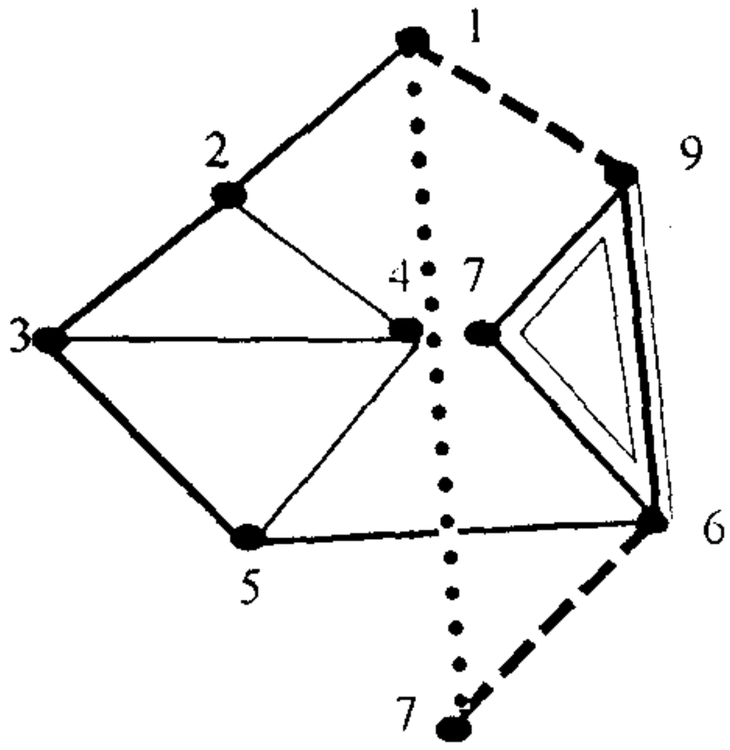
An example of the Algorithm RunSeparation()
The start node is 1, and end node is 7

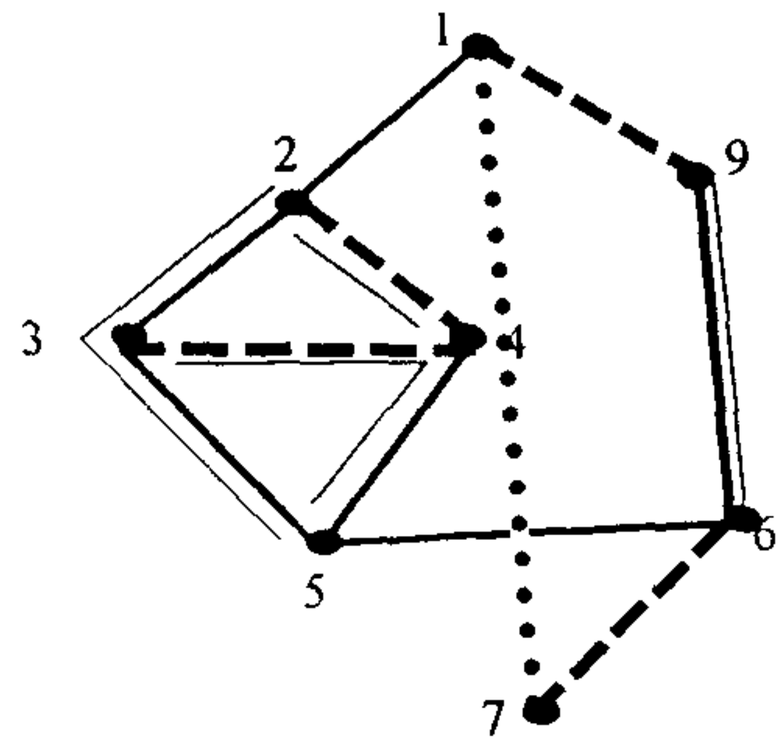




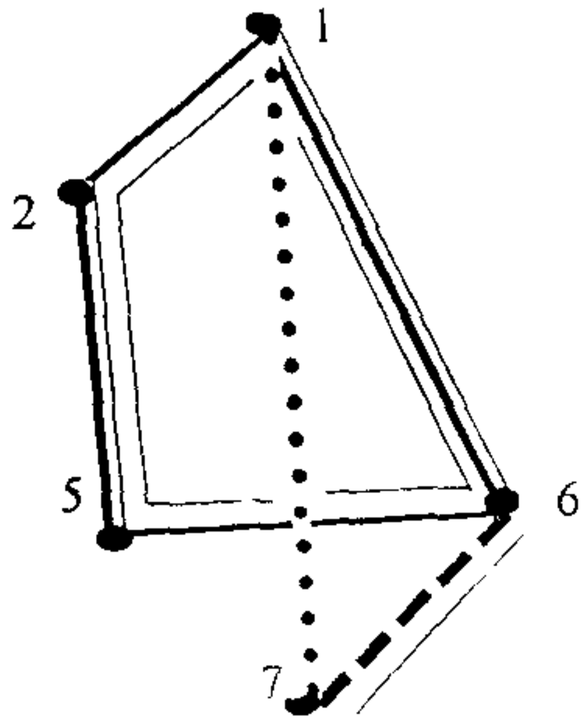
After addition of startedge and doing the dfs,
the first series component is detected

First one is original graph build from the circuit, other is the corresponding directed graph, created by dfs, dotted line is added line and starting edge, dark line is tree edge, dashed ones are backedge.





The fine lines along with the other lines denotes the edges in the next component to be replaced. The double line denotes the modified lines by the algorithms We are not compacting the startedge. The series edges replacement are not shown explicitly.



The final graph after compaction and separation pair replacement. The startedge is not to take part in the separation pair

Chapter 6

Welcome Feedbacks

Relaxing the Constraints

Until now all our discussions were based on the assumptions that any transistor's gate node is not connected to its drain or source node , directly or may be through a series of transistor's source and drain nodes, which we denote as the feedback in the circuit. This will obviously happen in the case of Sequential Circuits, and also may happen in the case of some combinational circuits. Now we relax this constraints by allowing some feedback of both of the above types. We have tested our algorithm on some test circuits. The results are satisfactory in the limit that they represent the true function of the circuit and does not take care of the invalid input combination.

There may be two types of feedbacks experienced. The first kind is that while evaluating function at a gate node, we face that the gate node is already marked. This indicates that for evaluating the gate node we have to know the status of this transistor. This kind of feedback is found mainly in the combinational circuit The other kind of feedback is that while a recursive call of ExtractFunction(node) sees a node, not a gate node, which is marked, but not by itself, i.e., it is marked by some call of BuildGraph() for a partially build graph down in the execution stack. This normally happens for a sequential circuit.

The proposed solution of this problem is to temporarily take those nodes as input signal nodes, and mark it in that way, but do not declare it as evaluated. Use this temporary variable in the extracted function and then try to replace that in the final result after evaluating it separately. If the next evaluated function contains its own term or any term of the previously evaluated term, then don't try to proceed. Otherwise replace the temporary variable with the evaluated function to get the final result. The combinational circuit may be evaluated by the first way, if temporary node is chosen properly. For the next type of feedback, which is mostly found in sequential circuit, it is better to identify the temporary inputs manually, such as the output of the latch or Flip/Flop.

6.1 Modification in AddtoGraph

This will require a modification in the function **AddtoGraph(node)** to identify the cases, and modify it accordingly. To identify the nodes marked by itself, it has to check for the presence of every marked node it faces while proceeding through a unmarked transistor. The other type of checking is whether the gate node is marked but not evaluated, which means a failure of the previous algorithm. In this case also it will mark the gate node as a temporary input node and proceed in the normal way. These checkings will obviously increase the running time but one has to pay the cost of evaluating sequential circuits.

6.2 Modification of function ExtractFunction()

The function ExtractFunction() should be modified to have a try to get rid of those temporary input signals marked by the AddtoGraph(), if possible. It will try to evaluate the functions associated with the corresponding node. If again some temporary input node is created, it will not proceed. The other way round is to tell the output node where the signal is feed back. Then it will reduce the unnecessary evaluation of the temporarily marked input signal node.

6.3 Modification in Verification

The verification algorithm also to be modified in a sense that it will also wait until all the temporary signals are evaluated, if possible. Then only it will declare the verification results. This will help to avoid any misguidance by the program.

Example of an sequential circuit : SR FLIP-FLOP

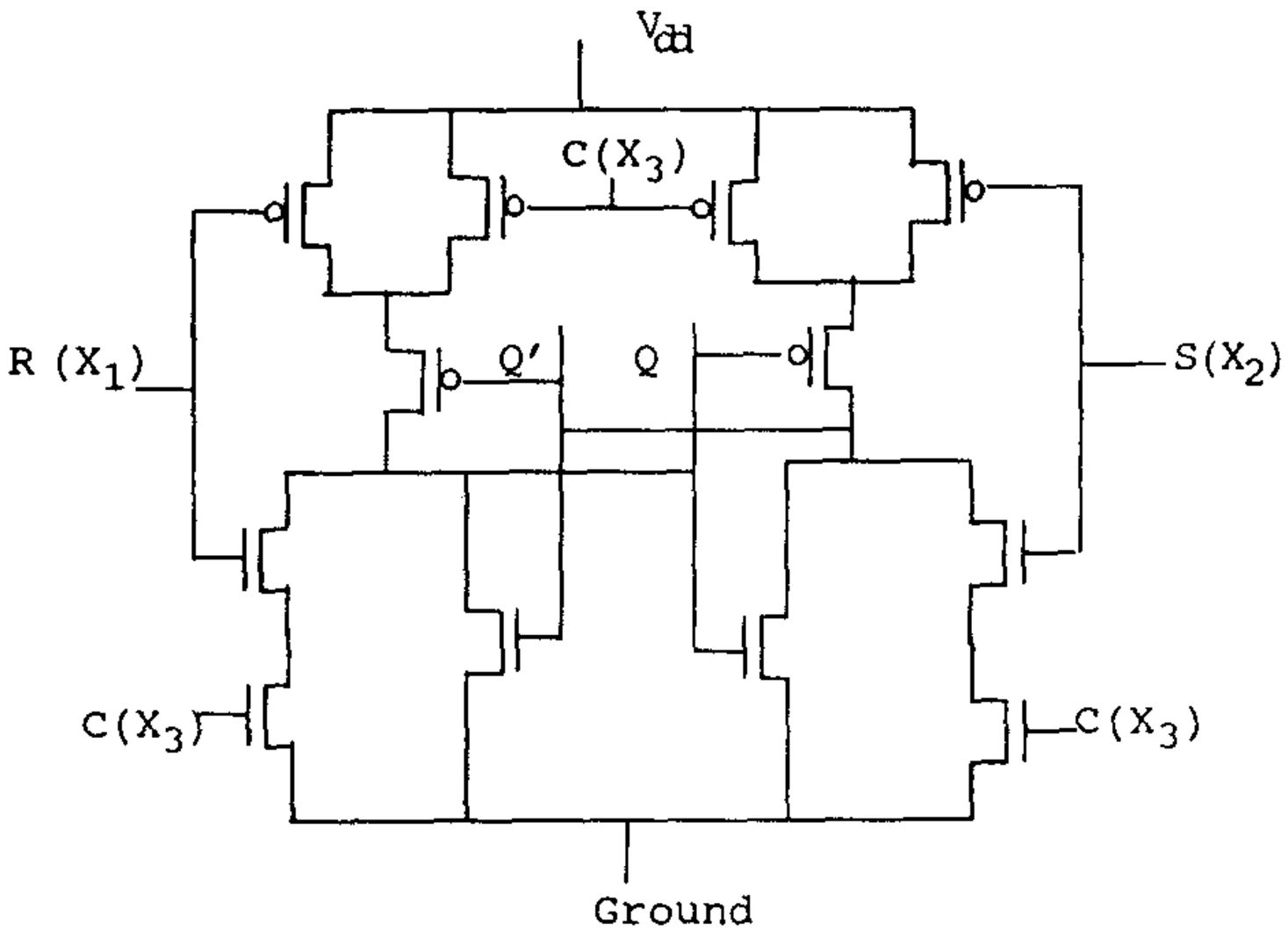


Fig. Clocked SR Flip Flop

The above circuit while tried to be extracted for the output node Q then it marks the node Q' as temporarily variable Y_3 and evaluates the function in form of Y_3 also. The function evaluated are given bellow. But we have to consider that the invalid input combinations C=1, R=1, and S=1 is taken as Q=0 and Q'=0 output instead of the general convention of Q=1 and Q'=1.

The function evaluated for the output Q is

$$Q = X_1'Y_3 + X_2'Y_3 + X_1'X_2X_3$$

which comes to

$$Q = R'.Q + C'.Q + R'.S.C$$

Chapter 7

Looking Beyond
Possible Future Work

The range and type of future work in this area is quite large. The circuit extraction process can be provided with the output point of the F/F (Flip Flops) so that it should not mark any arbit node as temporary node. It also prevents the algorithm from evaluating the unnecessary nodes. This also can be extended to the state transition table extraction where one signal will be marked as the clock signal. If that can be done then it can be easily used to identify any counter, and the type of the counter and its stages. It can then also verify from any arbitrary sequential circuit, whether it is robust in terms of homing sequence. This can be extended to the testability of any circuit by identifying the paths for testing. In the following sections we are present some of the future works.

7.1 The Problem of Function Matching

The function extraction and verification is also associated with another problem where a function is given and a circuit is given. The problem is to test whether this function can be implemented withe given circuit or not. This requires not only the knowledge of the circuit function, but for the equivalence we have to find the correspondence between the literal in the given function with the circuit input and the equivalence between the different representation of the same function.

7.2 The State table generation by marking the clock signal

The Algorithm can be extended to finding the state transition table of a sequential circuit in a sequential machine. For that we have to define the clock input and the output nodes for the state table entry. Then the program can automatically do the 0 and 1 assertion of the clock signal and find the state transition table for the designer. Thus we can identify any counter type sequential circuit from the switch level circuit description.

7.3 Homing sequence finding/ Robustness testing

The homing sequence of a sequential circuit is a sequence of input signals for which starting from any state the circuit will come to a known state. Knowing the state transition table, it can be automatically evaluated. This also can be used for robustness testing of the circuit, that is for any starting state it should always come to the required state transition sequence. This is required for those circuits where not all the possible states are used.

7.4 Circuit Delay Estimation

The work in other direction may be the estimation of the circuit delay by finding the longest active path and estimating the delay along this. This would be a complicated one as a path may be long but can never cause a problem to the circuit delay, i.e., there may exist a path which is longer than another path but the longer path can not change the output due to the other polarity set by another path which is shorter, and there may not be any combination where such a shorter path doesn't exists. This would be an interesting work.

7.5 Limitations of the current approach

The main drawback of the extraction of the function in the algorithm is the FindAllPaths() which runs in the exponential times with the # of nodes and # of edges. If this can be reduced then the total algorithm would be faster. This can be approached by identifying each possible path and guiding the selection of the transistors from a node towards the required direction.

Chapter 8

Some test Circuits and Results

The input file for the circuit in figure no 3, for space efficiency it is written continuously, and the commas (,) are introduced. Hope it will not misguide.

```
1 0 0 ,2 0 0 ,3 0 0 ,4 0 0 ,5 17 0 ,6 18 0 ,7 0 0 , 8 0 0 ,
9 0 0, 10 0 0 ,11 0 0 ,12 0 0 ,13 1 1 ,14 2 1 ,15 3 1 ,
16 4 1 ,17 7 1,18 5 1 ,19 6 1 ,20 8 1 , 21 9 1 ,-1,
1 2 1 13 1, 2 3 2 14 1 ,3 5 3 15 1 ,4 2 1 16 1 ,5 4 2 18 1,
6 5 4 19 1,7 4 1 17 1,8 3 4 20 1, 9 5 3 21 1, 10 6 11 17 0,
11 11 10 16 0, 12 10 1 13 0,13 11 9 18 0,14 7 9 20 0,
15 9 1 14 0, 16 6 7 19 0, 17 7 8 21 0, 18 8 1 15 0 ,-1
1 -1
```

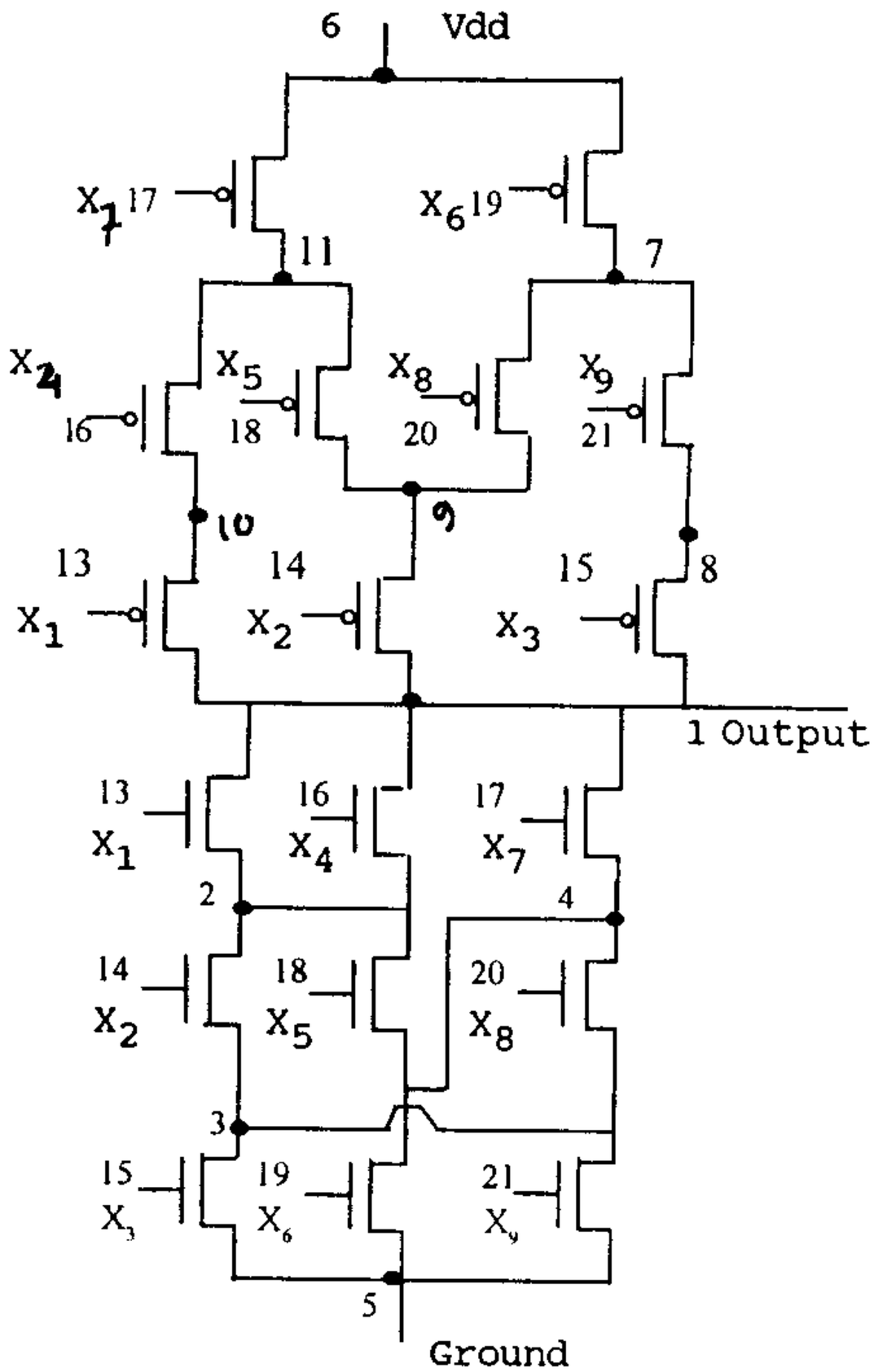


Figure 3 Circuit no 2.

Function : x1'.x4'.x7'+x2'.x5'.x7'+x2'.x6'.x8'+x1'.x4'.x5'.x6'.x8'+x3'.x6'.x9'+x3'.x5'.x7'.x8'.x9'

The output of the algorithm on the previous circuit

Enter the filename from read : circ4.dat

Extracting the function for node 1
Adding signal node 5
Adding signal node 6
Compacting in 5
Compacting in 3
Compacting in 1
Compacting in 2
Graph Number 0 Start vertex is 1 & End vertex is
Vertex No : 8
Edge between : From 8 to **1** The term is : x3'
Edge between : From 8 to **7** The term is : x9'
Vertex No : 7
Edge between : From 7 to **8** The term is : x9'
Edge between : From 7 to **6** The term is : x6'
Edge between : From 7 to **9** The term is : x8'
Vertex No : 9
Edge between : From 9 to **1** The term is : x2'
Edge between : From 9 to **7** The term is : x8'
Edge between : From 9 to **11** The term is : x5'
Vertex No : 6
Edge between : From 6 to **7** The term is : x6'
Edge between : From 6 to **11** The term is : x7'
Vertex No : 11
Edge between : From 11 to **9** The term is : x5'
Edge between : From 11 to **6** The term is : x7'
Edge between : From 11 to **10** The term is : x4'
Vertex No : 10
Edge between : From 10 to **11** The term is : x4'
Edge between : From 10 to **1** The term is : x1'
Vertex No : 4
Edge between : From 4 to **5** The term is : x6
Edge between : From 4 to **1** The term is : x7
Edge between : From 4 to **2** The term is : x5
Edge between : From 4 to **3** The term is : x8
Vertex No : 5
Edge between : From 5 to **3** The term is : x3 + x9
Edge between : From 5 to **4** The term is : x6
Vertex No : 3
Edge between : From 3 to **5** The term is : x3 + x9
Edge between : From 3 to **4** The term is : x8
Edge between : From 3 to **2** The term is : x2

Vertex No : 1

Continuation of the output for circuit no 2

Edge between : From 1 to 9 The term is : x2'
Edge between : From 1 to 8 The term is : x3'
Edge between : From 1 to 10 The term is : x1'
Edge between : From 1 to 4 The term is : x7
Edge between : From 1 to 2 The term is : x1 + x4
Vertex No : 2
Edge between : From 2 to 1 The term is : x1 + x4
Edge between : From 2 to 4 The term is : x5
Edge between : From 2 to 3 The term is : x2

Inputsignals are as follows...
Node no 5 ... Input Signal Sorry, no term is associated .
Node no 6 ... Input Signal The term is : 1

No of Signals are 2
Looking for path from 1 to 5
Looking for path from 1 to 6

The node 1 is evaluated as
The term is : x1'.x4'.x7' + x2'.x5'.x7' + x2'.x6'.x8' + x1'.x4'.x5'.x6'.x8' + x3'.x6'.x9' + x3'.x5'.x7'.x8'.x9'
The Extracted Function is as follows as in main
The term is : x1'.x4'.x7' + x2'.x5'.x7' + x2'.x6'.x8' + x1'.x4'.x5'.x6'.x8' + x3'.x6'.x9' + x3'.x5'.x7'.x8'.x9'

Bibliography

[1] A. V. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] Kiem-Phong Vo, Finding Triconnected Components of Graphs, *Linear and Multilenear Algebra, Vol 13* , pp 143-165, 1983.

[3] K. Einspahr, S.C.Seth, & B. B. Bhattacharya, Two-terminal Decompositions of Switch-level Circuits with Application to testing, *IEEE Annual International Workshop on Design for Testability*, Colorado, 1988.

[4] Naveed A. Sherwani, *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publications, 1993.

[5] Gary L. Miller & Vijaya Ramachandran, A new graph triconnectivity algorithm and its parallelization, *Combinatorica, Vol 12(1)*, pp 53-76, 1992 .

Appendix

Header file listing for some classes

```
////////////////////////////////////////////////////////////////////////////
////  HEADER FILE FOR CLASS VERTEX AND EDGE  /////
////////////////////////////////////////////////////////////////////////////
#ifndef _VERTEXEDGE
#define _VERTEXEDGE
#include "mtclass.h"
#include "list.h"
#include "termlist.h"
#include <iostream.h>
#include <process.h>
extern int NoOfPaths;
class Edge;     // forward declaration for the edgelist;
class dVertex;  // forward declaration for the corresponding vertex.
class Vertex : public Object {
 private :
  int number,visit,deg;
  dVertex *DgVertex;
  MyList<Edge> Adjlist;
  int x,y;
public:
  Vertex() { DgVertex=NULL;number=0;visit=0; deg=0; }
  ~Vertex();
  void SetNumb(int i)   { number=i;  }
  void SetNumber(int i); // provisions for inputing the coordinates also
  int Number()   const  { return number; }
  void Print()   const  { cout<<endl<<"Vertex No : "<<number<<"  ";}
  void PrintEdges();    // Done
  void Visit()          { visit=1; }
  int IsVisited() const { return visit; }
  int Degree() const    { return deg; }
  void UnVisit()        { visit=0; }
  void StartList()      { Adjlist. Restart(); }
  int Next();
  Edge *GetEdge()       {  return Adjlist.GetItem(); }
  int AddEdge(Edge*);   // Partially done
  int DeleteEdge(Edge*);
  int DetachEdge(Edge *e) { return Adjlist.DetachItem(e,0); }
  void CompParallel();      // replaces all the parallel edges by their equivalent
Edge *FindEdge(Vertex *);
```

```
Edge *FindNext(Vertex *);   // finds the next edge going to the said vertex
int X() const { return x;}  // for storing the coordinates, not used now
int Y() const { return y;}
Termlist* FindAllPath(Vertex*);  // returns the combinations for the paths
void SetCorrespond(dVertex *v) { DgVertex=v;}
dVertex *GetCorrespond() const { return DgVertex; }
// the following functions are for saving the data structure in the file
Object *Vertex::Vertex_FTAG() { return new Vertex; }
int thecode(){return MetaClass::thecode((ICP)&Vertex::Vertex_FTAG);}
void ArchiveRead(Archiver*);
void ArchiveWrite(Archiver*);
void ArchiveGet(Archiver *a) { Object::ArchiveGet(a);ArchiveRead(a);}
void ArchivePut(Archiver *a) { Object::ArchivePut(a);ArchiveWrite(a);}
  };
class Edge : public Object {
private:
  int edgetype,edgeno;
  Vertex *head,*tail;  // for storing the from vertex address and to vertex address
  Termlist function;   // associated function
 public:
  Edge() {}
  Edge(Vertex *t,Vertex *h);
  Vertex *Head() const { return head; }
  Vertex *Tail() const { return tail; }
  void SetHead(Vertex *h) { head=h; }
  void SetNumber(int i) { edgeno=i; }
  int Number()  const   { return edgeno; }
  void SetTail(Vertex *t); // takes care of deleting from previous list and
                           // adding to the new one.
  void Print() const;
  void SetTerms(Termlist& func) { function.Copyterm(func);}
  Termlist& Function() { return function; }
  void Display(int) const;
  // following functions are for saving the data structures along with the pointers
  Object* Edge_FTAG() { return new Edge; }
  int thecode() { return MetaClass::thecode((ICP)&Edge::Edge_FTAG); }
  void ArchiveRead(Archiver*);
  void ArchiveWrite(Archiver*);
  void ArchiveGet(Archiver *a) { Object::ArchiveGet(a);ArchiveRead(a);}
  void ArchivePut(Archiver *a) { Object::ArchivePut(a);ArchiveWrite(a);}
 };
#define TREEEDGE 1
#define BACKEDGE 0
class dEdge;  // forward declaration for directed edge in dfs
class dVertex {
 private :
  int no,dfsno,father,firstp,mark,lowpt;
```

```
  MyList<dEdge> Adjlist;
  Vertex *original;
public:
  dVertex() ;
  dVertex( Vertex *v);   // creates the corr. dvertex
  dVertex(int n,Edge *e); //creates the dVertex corresponding to the head of the edge e.
  void StartList()      { Adjlist.Restart(); }
  int Next();
  dEdge *GetEdge() const { return Adjlist.GetItem(); }
  int AdjustList();  // adjusts the lists after doing the dfs
  int Degree()  const  { if (original != 0) return original->Degree(); else return 0; }
  int Dfsnum()  const  { return dfsno; }
  void SetDfs(int i)    { dfsno=i; }
  int Low() const       { return lowpt; }
  void SetLow(int i)    { lowpt=i; }
  void SetNumber(int i) { no =i;}
  int Number()  const   { return no; }
  void SetFather(int i) { father=i; }
  int Father()  const   { return father; }
  int AddEdge(dEdge*);
  int DeleteEdge(dEdge*);
  Vertex* Original() const { return original; }
  void SetFirstPath(int i) { firstp=i; }
   int FirstPath() const { return firstp; }
   void Print();
   void Display(int) const;
  };

class dEdge  {
 private:
   int low1,low2,edgetype,edgeno,mark;
   dVertex *head,*tail,*low;
 public:
   dEdge(Edge*,int);  // original edge,tree/back edge .
   dVertex *Head() { return head; }
   dVertex *Tail() { return tail; }
   void SetHead(dVertex *h) { head=h; }
        void SetTail(dVertex *t); // take care of deleting from previous list  and adding to the new
one.
        int Low1() const { return low1; }
        int Low2() const { return low2; }
       void SetLow1(int i) { low1=i; }
       void SetLow2(int i) { low2=i; }
      void AdjustLowPoint(dEdge*);
       void Mark()       { mark=1 ;}
      void UnMark() { if(mark==0) cout<<"Already unmarked"; else mark=0; }
      int IsMarked() const { return mark ;}
```

```
int Type()  const   { return edgetype; }
void SetType(int i) { edgetype=i; }
void Print() const
    { cout<<" to "<<head->Number()<<",("; cout<<low1<<","<<low2<<") "; }
void Display(int) const;
};

#endif
```

Header file listing for the graph class .. directed and undirected

```
////////////////////////////////////////////////////////////////////////
//// HEADER FILES FOR THE GRAPH CLASS ////////
////////////////////////////////////////////////////////////////////////
#ifndef __GRAPH
#define __GRAPH
#include "veredge.h"
#include "list.h"
#include "termlist.h"
#include "path.h"
#include <iostream.h>

class DirectedGraph;  // forward declaration for the corresponding  directed graph.

class Graph {
private:
static int num;
int        no,noofsignal;   // keep track of the no of signalnode connected
MyList<Vertex> nodelist;
Vertex     *start,*end;   // start is the outputnode. end is set to one signal node at a time
MyArray<Vertex*> InputNodes;    // array of input signal nodes
MyArray<Termlist*> InputSignals;   // array of input signals at the node of same index
Edge       *StartEdge;
Termlist   function;      // function value at the output node
dEdge *DFS(Edge*);   //  used internally for the depth first search
DirectedGraph *dGraph;
public:
Graph();
 ~Graph();
int AddEdge(Edge*);      // adds new vertex(ices) if required int
AddEdge(int i,int j); // adds two new edges between node i and node j int
AddEdge(int,int,const Termlist&);  // adds an edge of function termlist between two vertices
int DetachEdge(Edge*);   // deletes the vertex(ices) if degree is 0.
void DeleteVertex(Vertex*);
int AddVertex(Vertex *v)    { return nodelist.AddItem(v); }
void DetachVertex(Vertex *v)
{ if(nodelist.DetachItem(v,0)==0) cout<<"\nCouldnot detach vertex"; }
```

Continuation of the source listing for graph.h

```
Vertex *Start() const      { return start; }
Vertex *End() const        { return end;   }
int AddSignal(int,Termlist* );   // adds one signal node in the arrays
void SetStart(Vertex *v)   { start=v;      }
void SetEnd(Vertex *v)     { end=v;        }
int Number() const         { return no;    }
const MyList<Vertex> *Nodelist() const { return &nodelist; }
int DFSearch() ;
int AdjustList() ;  // order the adjacency list at each node depending on low numbers
void Compact(); // compacts any parallel edge in the graph
void PrintGraph() ;
Termlist* FindAllPaths() ;
void RunSeparation();
void PrintDirected() ;
void Display();
// following function are for saving the graph structure as it is
void SaveGraph();
void RetrieveGraph();
};
class DirectedGraph
{ private :
MyList<dVertex> nodelist;
dVertex *Start,*End;
Graph   *parent;
dEdge   *startdedge;
public :
DirectedGraph(Graph *gr)     { Start=End=NULL;parent=gr;}
int AddEdge(dEdge*);      // add and edge, adds new vertex(ices) if required
int AddVertex(dVertex*);
void DeleteVertex(dVertex*);
void DetachVertex(dVertex* dv) { nodelist.DetachItem(dv,0); }
Path *BuildPatr(dEdge*);     // the function for detecting the separation pairs
dEdge *StartEdge() const     { return startdedge; }
void SetStartEdge(dEdge *de) { startdedge=de; }
int AdjustList();        // adjust the edgelist according to their lowpoints and tail
void PrintGraph();
void Display();
Graph *Parent() const { return parent; }
void CompSeries(Path *p,dEdge *de);
// replace the series edge de, and the edge going out from its head in the path p
int CompParallel(Path *p1,dEdge *de,Path *p2);
// looks for a parallel edge with path p2, as p2 has only one edge which starts at the tail of de
void Type2B(Path *p,int LS,int LOWC);  // deletes one type 2B component from the graph
    // and  replace it by a single edge in original graph as well as in the directed graph
```

```
void Type1_2A(Path *p,dEdge *de);  // deletes one type 1 or 2A component from the graph
    // and  replace it by a single edge in original graph as well as in the directed graph
void TxPath(Graph *grf,dVertex *dv1,dVertex *dv2,Path *p);
// transfers the edges between the vertices v1 and v2 of the path p and paths originating from
// it from the directed graph to the component graph grf,
};

#endif
```

Listing of the template class MyList and MyArray

```
////////////////////////////////////////////////////////////////////////
////  HEADER FOR THE LIST MANAGEMENT  ////
#ifndef __MYLIST
#define __MYLIST
#define NULL 0
#include <process.h>
template <class T> class ListItem {
private : T  *data; ListItem<T> *next;
public : ListItem()
{ data=NULL;next=NULL; }
void SetItem(T* newitem) { data=newitem;}
void InsertNext(ListItem<T> *nt)
{ if(next!=NULL) nt->InsertNext(next); next=nt; }
friend class MyList<T>;
friend class MyListIterator<T>;
};

template <class T> class MyList
{ private:
ListItem<T> *header;
ListItem<T> *current;
int totalitem;
public:
MyList()                { header=NULL;totalitem=0; }
~MyList();              // deletes the list item also
int ScanItem(T*);   // returns the item no in the list if exists, else returns zero.
int DetachItem(T*,int);  // int = 1 deletes the data item
int AddItem(T*);        // this data to add doesn't repeat
T*  GetItem() const
{ if(current!=NULL) return current->data; else return NULL;}
void Restart()          { current=header; }
int EolReached() const   { return (current->next==NULL)? 1 : 0 ; }
void Step()             { if (current!=NULL) current=current->next; }
int Total() const       { return totalitem; }
friend class MyListIterator<T>;
};
```

Continuation of the source listing of header files for the class array and list...

```
//// List Iterator Class and functions ////
template <class T> class MyListIterator
{ private :
const MyList<T> *thelist;
ListItem<T> *current1;
public:
MyListIterator() { thelist=NULL; current1=NULL;}
MyListIterator(const MyList<T> *lst) { thelist=lst; current1=NULL;}
 virtual ~MyListIterator() { thelist=NULL; current1=NULL;}
void Restart() ;  // set the current pointer at the very begining of the list
T *GetCurrent() const { if (current1) return current1->data; else return NULL;}
T *Next();    // returns the current and move the current pointer to the next entry
T *Find(T *t) const; // search for the item t, if found, returns its address, else return NULL.
T *SetCurrent(T*);  // sets the current pointer to a specified entry
};

////  Template Class Array and its implementation   /////
template <class T> class ArrayItem
{ private:
int size;
T *items;
ArrayItem<T> *next;
public:
ArrayItem(int);
~ArrayItem();     // deletes all the array content for this item
T& operator[](int );
ArrayItem<T>* Next() const { return next; }
friend class MyArray<T>;
};

template <class T> class MyArray
{ private :
int basesize,increment,currentsize;
ArrayItem<T> *header;
public:
MyArray() { basesize=0; increment=10; currentsize=0; header=NULL; }
MyArray(int,int);       // initial size, and incrementing size can be set
T& operator[](int);
// overloaded index operator, adds dynamically if indexing crosses currentsize
int CurrentSize()       { return currentsize; }
};

#endif
```