Intelligent Circuit Recognition

for VLSI Layout Verification

by

Glenn Griffin

Report submitted to the Faculty of the

Virginia Polytechnic Institute and State University

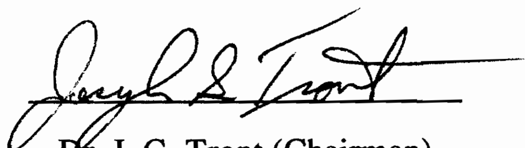
in partial fulfillment of the requirements for the degree of

Master of Science

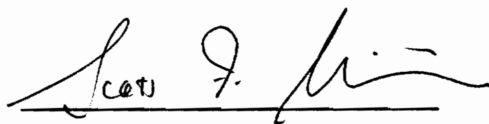
in

Electrical Engineering

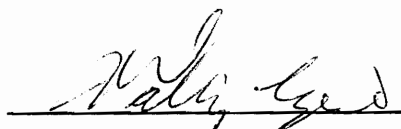
APPROVED:



Dr. J. G. Tront (Chairman)



Dr. S. F. Midkiff



Dr. W. R. Cyre

April, 1993

Blacksburg, Virginia

LD
5655
V851
1993
G754
C.2

c.2

Intelligent Circuit Recognition for VLSI Layout Verification

by

Glenn Griffin

Dr. J. G. Tront (Chairman)

Electrical Engineering

Abstract

The ability to extract higher level information from a circuit netlist is useful for VLSI layout verification. An extracted gate level description may be used as input to a gate level simulator for analysis or alternatively may be used as input to a rule-based expert system that performs verification checking at a higher level of abstraction. As a VLSI design evolves it is continually checked for correctness. This implies that the extraction of higher level information is a recurring activity and should be performed as efficiently as possible. This paper describes an alternative method that uses intelligence to quicken the extraction process and compares this method's performance to a more common method.

Acknowledgements

I would like to thank my advisor, Dr. Tront for his help and patience.

I would also like to thank Dr. Midkiff and Dr. Cyre for serving as members of my committee.

Table of Contents

Section 1. Introduction 1

Section 2. Structural Circuit Recognition Techniques 3

Section 3. An Intelligent Recognition Method 5

- 3.1 Basic Program Operation 6
- 3.2 Circuit Data Structures 6
- 3.3 Library Data Structures 7
- 3.4 Intelligent Search Engine 8
- 3.5 Intelligent Matching and Binding 9

Section 4. Experimental Results 13

Section 5. Conclusions 15

References 16

Appendix 21

Section 1

Introduction

Today's VLSI circuits are growing increasingly larger in size and complexity. Ever improving fabrication technologies combine with sophisticated VLSI CAD methodologies to enable the design and fabrication of larger, more complex systems. This increase in the scale of integration places higher demands on the verification methods used to insure the correctness of these VLSI layouts.

One common verification method is to use simulations to check that a VLSI layout is correct [1,2,3]. Both circuit timing and function can be checked by using a switch level simulator which operates on the VLSI layout's extracted netlist. Unfortunately this method becomes prohibitively time consuming as the number of transistors on the chip rises [1,2,4]. If a "circuit recognizer" is used to transform the switch level description into a gate level description, then a gate level simulator may be used. Gate level simulators offer significant improvements in simulation speed without losing much accuracy.

Another verification method is to check a VLSI layout for functional correctness [5,6,7,8]. A functional correctness check disregards timing information and concentrates on whether or not the logic implemented on the chip matches the intended logic of the designer. One method of performing this check is to employ a circuit recognizer to extract the higher level information based on logical equivalence [8]. Under the logical equivalence basis, several different

implementations of a circuit that all yield the same boolean equation are all mapped to the same higher level construct. The resulting high level description contains the logical characteristics of the circuit and can be used as input to an expert system that attempts to formally prove correctness [3,6].

Increasingly layout verification is being aided by expert systems designed to perform more rigorous checking at high levels of abstraction. These types of systems discriminate between different implementations of the same function and consider each implementation's particular timing, fan-in, and fan-out characteristics [1,2,4]. In this case the circuit recognizer must preserve implementation details and extract high level information based on circuit structure. This type of structural circuit recognition is the focus of this report.

From what has been said it can be seen that circuit recognizers have many applications in VLSI layout verification. Each time a layout is altered the corresponding higher level description must be updated and rechecked. Thus circuit recognition is a recurring engineering activity worthy of investigation.

Section 2

Structural Circuit Recognition Techniques

Before a circuit recognizing program makes any attempts at subcircuit identification it must first partition the transistor netlist into a collection of sets called blocks [9]. After this has occurred then attempts can be made at matching these unidentified blocks with known reference blocks that are stored in the recognizer's library.

Many of the early circuit recognition systems were written in procedural languages in which each individual block structure in the library of recognizable blocks (viz., the library of recognizable gates) had its own specially coded recognition procedure [11,12,13,14]. Each procedure performed a brute-force search on the entire extracted netlist for all instances of one particular block structure. Each of these recognition procedures was successively called by a main program until either there were no more unrecognized blocks or the library of recognition procedures had been exhausted. At this point all recognizable blocks had been identified.

This type of recognition system had two primary deficiencies, one of which was the difficulty involved with expanding the library of recognizable blocks. In order to expand the library so that a new block could be recognized, a new procedure had to be coded and the recognition program had to be recompiled.

This time-consuming, mistake-prone task required that the programmer have detailed knowledge of recognition system.

The second short coming of this type of recognition system was its inefficient brute-force searching method. As the number of devices on the VLSI layout increases, each recognition procedure performs more and more wasteful searching on the increased number of unrecognized blocks. Furthermore, as the library of recognizable blocks is expanded, a tremendous amount of wasteful searching occurs because more recognition procedures are called, each of which performs a brute-force search on the entire circuit. These inefficiencies are particularly troubling considering today's trend toward larger, more complex layouts.

The ease of library extendibility was improved on by future systems that accepted circuit recognition rules as input [1,2,8,10]. Some of these systems took advantage of the superior pattern matching abilities of Lisp and Prolog [1,2,8]. In these cases the circuit recognition rules corresponded to simple Lisp or Prolog statements. These systems were indeed more flexible than their predecessors but they still suffered from inefficient brute force searching techniques. Efforts were made to reduce the number of rules in the library and thus improve performance by taking advantage of redundancies between rules [1,2]. This strategy had the undesirable effect of making rules more difficult to code and although system performance improved, the underlying inefficiencies of the brute-force method still remained.

Section 3

An Intelligent Recognition Method

In reviewing these different methods for circuit recognition, it is apparent that they employ brute-force searching methods in which the entire circuit is searched for all instances of a certain type of block (e.g., 3 input NAND) before the next alternative is considered. Another approach to the problem is to concentrate on identifying a single unidentified block. Once this single unidentified block is recognized as an instance of a certain type then the next unidentified block can be considered. This method enables the program to study the discriminating characteristics of each unidentified block and to use knowledge of the problem domain to limit the number of possibilities to be considered.

Although this method spends more time examining the unrecognized circuit blocks than does the brute-force method, this lost time should theoretically be offset by the gains of a more efficient search. A further advantage of this method is that it should be relatively insensitive to increases in the size of the library of recognizable blocks. To test this theory an experimental program called REBIS (REcognition By Intelligent Search) was written that recognizes static, pseudo-nMOS, and dynamic implementations of CMOS circuits.

3.1 Basic Program Operation

REBIS consists of approximately 3500 lines of C code and accepts Spice-like circuit descriptions of both the library of recognizable blocks and of the circuit to be processed. The program partitions the circuit using a linear time partitioning procedure [9] and then begins the block by block recognition process. As can be seen in Figure 3.1.1, REBIS first computes discriminating characteristics and then uses these characteristics to select possible reference blocks. The pattern matcher checks if a given reference block is a match and the program continues working on a given unidentified block until a match is found or all possibilities are exhausted . After all blocks have been processed then REBIS reports the recognized and unrecognized blocks.

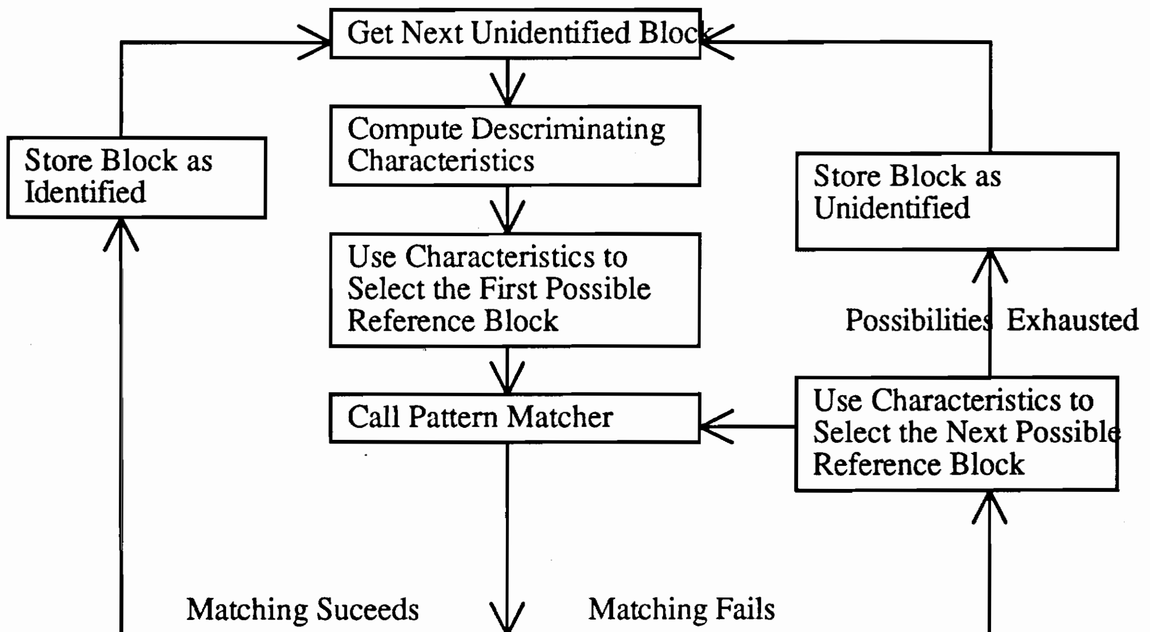


Figure 3.1.1. REBIS flowchart

3.2 Circuit Data Structures

REBIS accepts as input a circuit description file that consists of transistor statements. Each transistor statement denotes the source, drain, and gate connections as well as the transistor type (viz., n-type or p-type). As REBIS reads a line from the input file, each transistor terminal (node) name is hashed into an open hash table. The hash table is used to associate a unique number with each unique node name. When a node name is hashed into the table, it may either be inserted or found. In either case a unique node number is returned. This node name to node number mapping enables the rest of the program to use integer comparisons instead of more time consuming string comparisons.

After all of the node numbers are assigned for a given transistor, then an edge is inserted into a graph that represents the circuit. Each vertex in the graph corresponds to a source node or drain node of a transistor. The graph is represented using an adjacency matrix which is stored as an array indexed by node number where each array position contains a pointer to a linked list of all adjacent vertices. Two vertices are adjacent if an edge (transistor) exists that joins them. The array structure and linked lists have been optimized to allow for very fast insertion, deletion, and searching. These features help accelerate graph construction and partitioning.

3.3 Library Data Structures

In order for REBIS to quickly locate possible matches, the reference blocks need to be grouped according to certain discriminating characteristics. The selection of these characteristics was the first step in designing the library data structures.

There are many different characteristics that are properties of a given reference circuit block; the goal was to select attributes that would be most effective in discriminating between different reference circuits (e.g., between 2-NAND and 3-NAND or between 2-NAND and 2-NOR). The following characteristics were considered: the number of p transistors, the number of n transistors, the total number of transistors, the number of VDD connections (VDD degree), the number of GND connections (GND degree), the number of unique nodes, and the number of nodes with a given degree. The first three characteristics are equally effective at discriminating a 2-NAND from a 3-NAND, while VDD degree and GND degree are equally helpful at discriminating a 2-NAND from a 2-NOR. Using this reasoning, the total number of transistors and the VDD degree were selected as the most useful characteristics.

Using these two characteristics, the library of reference blocks were stored in a doubly sorted linked list. Each block in the list is first sorted by the number of transistors in the block and then is sorted by its VDD degree. Figure 3.3.1 depicts this list structure where each element in the list corresponds to a reference block. The name field stores the name of the reference block, the next two fields store the discriminating characteristics, and the last field stores a pointer to the next element in the list. The sorted list structure enables the search engine to quickly locate the first possible matching. If any other matchings are possible, they will be stored immediately following the first one, and thus all possible matchings can be quickly located.

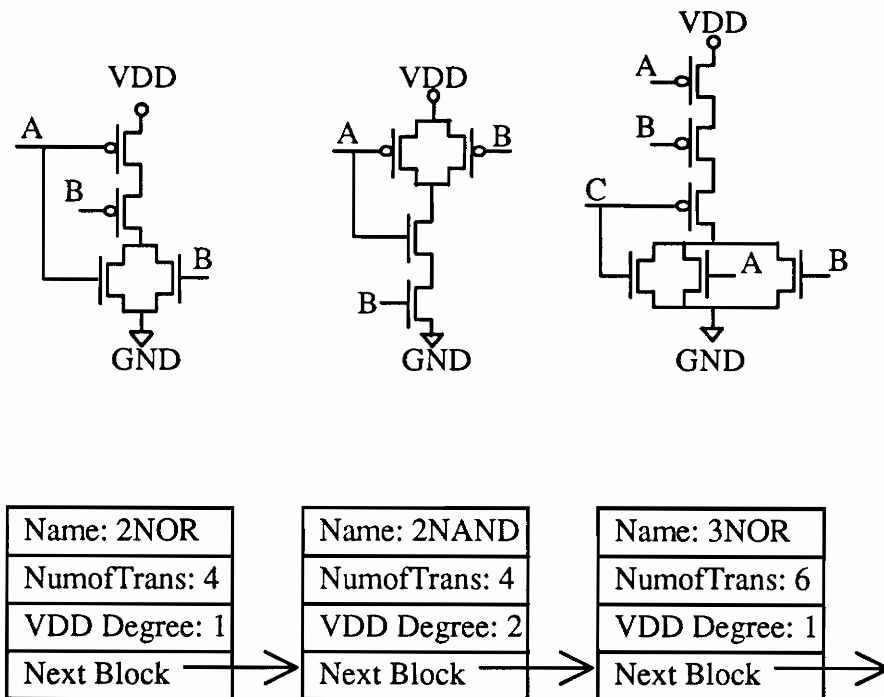


Figure 3.3.1. Reference library linked list sorting order.

The library list is created at program run time. REBIS reads reference block descriptions from a library file, and inserts each reference block into its sorted position in the library list. This allows additional reference blocks to be added anywhere in the library file without any change in system performance. This system feature is very useful in that it facilitates system library extendibility.

3.4 Intelligent Search Engine

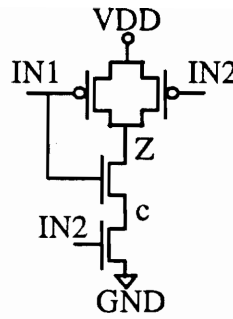
Operating on the previously defined data structures, REBIS uses knowledge of the problem domain to speed the recognition of unknown blocks. The recognition procedure advances through the partitioned circuit considering each individual block. Compiled statistics on the total number of transistors and the number of VDD connections are used to reduce the number of possibilities to be

searched. Very often there will be only one possible reference block for a given unrecognized block. Another advantage of this technique is that if there are no possibilities for a given unknown block, either because the library is incomplete or the block is erroneous, then the search ends quickly. The brute force method will continue to search on unrecognizable blocks to no avail.

3.5 Intelligent Matching and Binding

In order to have intimate control of the data structures and the searching procedures REBIS was implemented in the C programming language. This choice had the detrimental effect of increasing the difficulty of the coding because C does not support pattern matching. The pattern matcher in REBIS has to find a one-to-one and onto mapping from the transistors and nodes in a reference block to the transistors and nodes in an unidentified block. Because the pattern matcher was designed specifically for REBIS it was possible to optimize it for its specialized task.

Before any mapping is attempted, the pattern matcher first checks to determine if a matching is possible by performing a node degree check. This check stems from a lemma in graph theory that states that if two graphs G and H are isomorphic then for every vertex v in G with degree k there must be a unique vertex w in H with degree k [15]. This check is performed by creating and comparing two lists called node degree lists. These lists contain the node degree information in a structure that allows for quick comparison. Figure 3.5.1 shows a static CMOS two input NAND gate and the corresponding node degree list. Each element in the list corresponds to a node in the circuit. The list is sorted from the highest degree node to the lowest to allow for fast node degree list comparisons.



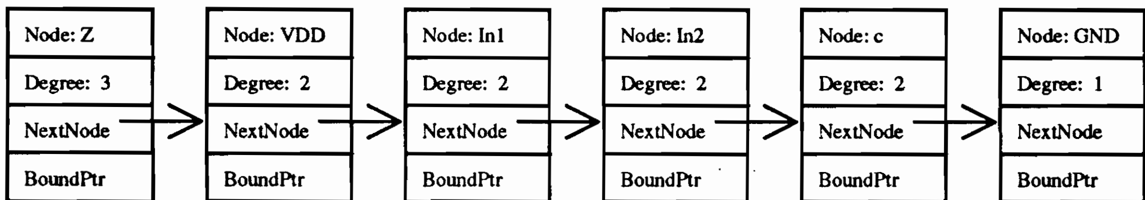
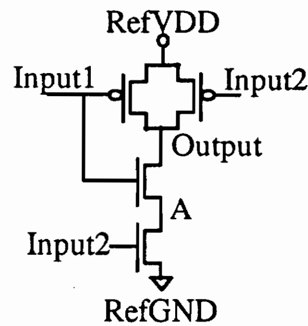


Figure 3.5.1. Node-degree-list for a static 2 input NAND gate.

At this point, one might ask if too much overhead processing time is being spent on creating and comparing these lists instead of proceeding with actual matching attempts. It should be noted that the node degree list for each reference block and the list for each unrecognized block are created only once per program execution and are reused thereafter. Furthermore, these lists are not created for this single purpose; they both are eventually used in the binding procedures by the pattern matcher. Thus only the quick list comparison and not the more time intensive creation of the lists should be counted as overhead.

If the node degree check succeeds, then the pattern matcher has to determine if the reference block and the unrecognized block have identical structural connectivity. The structural connectivity information that describes each

block is stored in a list of transistors similar to a netlist. Figures 3.5.2a shows a reference static two input NAND gate and its corresponding transistor list while Figure 3.5.2b shows an unrecognized static two input NAND gate and its corresponding transistor list.



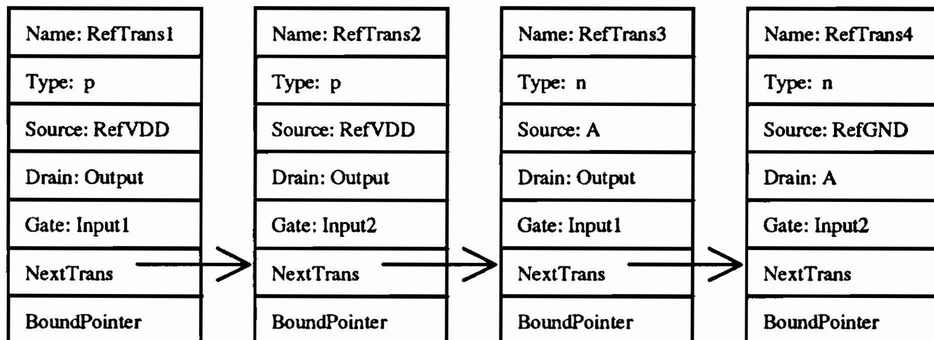
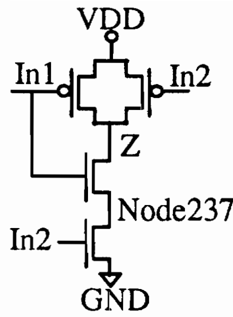


Figure 3.5.2a. Reference transistor list for a static 2 input NAND gate.



Name: UnRec1	Name: UnRec2	Name: UnRec3	Name: UnRec4
Type: p	Type: n	Type: p	Type: n
Source: VDD	Source: Node237	Source: VDD	Source: GND
Drain: Z	Drain: Z	Drain: Z	Drain: Node237
Gate: In1	Gate: In1	Gate: In2	Gate: In2
NextTrans →	NextTrans →	NextTrans →	NextTrans
BoundPointer	BoundPointer	BoundPointer	BoundPointer

Figure 3.5.2b. Unrecognized transistor list for a static 2 input NAND gate.

The pattern matcher operates by associating or "binding" transistors in the reference block's transistor list to transistors in the unrecognized block's transistor list. If the pattern matcher succeeds in binding all the transistors then the unrecognized block has been identified. However, if a binding conflict arises, then the pattern matcher must unbind the most recently bound pair of transistors and try to find a different binding until either a binding succeeds or all possible bindings have been attempted.

The "BoundPointer" of each transistor list element is used to record the transistor binding information for that transistor. For example, if the transistor RefTrans2 in Figure 3.5.2a was to be bound to the transistor UnRec3 in Figure 3.5.2b then the BoundPointer of RefTrans2 would be set to point at UnRec3 and

the BoundPointer of UnRec3 would be set to point at RefTrans2. Before two transistors can be bound they must be: 1) currently unbound, 2) of the same type (viz. n-type or p-type), and 3) their terminals must have no node binding conflicts. The first two of these three conditions are easily checked using information stored in the transistor lists but the third condition requires additional information. This information is stored and maintained in the node degree lists of both the reference and unrecognized blocks. The method used to represent bound nodes is identical to the method used to represent bound transistors; two nodes are bound to each other by setting the appropriate pointers.

REBIS detects a node binding conflict between two nodes by checking if the two nodes to be bound satisfy the node binding criterion. A necessary and sufficient node binding criterion is as follows: two nodes may be bound to each other if and only if they are both currently unbound. REBIS uses knowledge of the problem domain to strengthen this node binding criterion to the following: two nodes may be bound to each other if and only if they are both currently unbound and *are of like degree*. Without this stronger binding criterion it would be possible for REBIS to incorrectly bind nodes with unequal degrees. For example, the node "Output" in Figure 3.5.2a with degree three could be incorrectly bound to the node "Node237" in Figure 3.5.2b with degree two. This wasteful binding is avoided by using the more intelligent criterion.

In addition to the intelligent binding criterion, the pattern matcher in REBIS performs "prebinding." At the beginning of the matching process the pattern matcher binds any VDD and GND nodes that might exist in the unrecognized block to the corresponding RefVDD and RefGND in the reference block. This prebinding technique further constrains the pattern matching procedure and helps eliminate wasteful rebinding (viz., repeatedly binding and unbinding the same node), and thus increases matching efficiency.

Section 4

Experimental Results

In order to gain some insight into the relative performance of the intelligent search a second program was written. This program, also written in C, used the exact same data structures as REBIS but employed the brute-force search method. Four test circuits of varying sizes that contained a random mixture of static CMOS gates were used as input. To measure the sensitivity of program performance to changes in the library size, each of the four test circuits was processed using three different libraries. The first and smallest library contained eight static CMOS blocks. This library was sufficient to recognize all of the gates in all of the test circuits. The second library added eight pseudo-nMOS blocks to the first library, while the third library added eight dynamic CMOS blocks to the second library. Both programs were run on a 486 33MHz IBM clone with 4 Megabytes of memory. The run-time results of these tests are tabulated below.

Circuit Name	# of Trans	# of Gates	Brute-force Method (seconds)			REBIS (seconds)		
TEST216	6048	1080	0.66	1.64	2.53	0.22	0.22	0.22
TEST491	9820	1473	1.37	3.19	5.05	0.34	0.34	0.34
TEST563	15764	2815	1.76	4.18	6.54	0.55	0.55	0.55
TEST2034	24408	4068	3.52	7.47	11.65	0.82	0.83	0.83
Library Size (in recognizable blocks)			8	16	24	8	16	24

From the above results it can be seen that REBIS outperforms the brute-force method by a significant amount on all circuits for a given library size. As the circuit size grows the inefficiencies of the brute-force method become more apparent. More importantly, it should be noted that REBIS experiences no significant additional time delay as its library size increases. The same cannot be said of the brute-force method which takes substantially more time with each increase in the library size. This is due to the fact that increasing the library size effectively increases the search space for the brute-force method. Unlike REBIS, the brute-force method cannot recognize that all the addition pseudo-nMOS and dynamic CMOS blocks in the larger libraries are impossible and thus valuable time is wasted searching for them.

Section 5

Conclusions

The circuit recognition method implemented by REBIS represents a viable alternative to the brute-force approach. Although both methods allow for easy library extendibility, the results of this experiment effectively demonstrate the relative inefficiencies of the brute-force technique. Of particular significance is the brute-force method's high sensitivity to increases in library size. This deficiency is a real concern since the libraries employed by VLSI designers will most likely continue to grow as more specialized circuitry and more logic families are used in designs. As the size of these libraries increase the performance of the brute-force method may degrade until it eventually becomes unacceptable. If this occurs, it may be worthwhile to consider a REBIS-type solution since it is substantially faster and is relatively insensitive to increases in the library size.

References

[1] Kostelojik, A.P., "VERA, a Rule-based Verification Assistant for VLSI circuit Design", VLSI 89, Elsevier Science Publishers B. V. (North-Holland), pp. 89-98, 1990.

[2] Jacques Wenin, Johan Verhasselt, Marc Van Camp, Jean Leonard, Pierre Guebels, "RULE_BASED VLSI VERIFICATION SYSTEM CONSTRAINED BY LAYOUT PARASITICS", Proceedings of the 26th ACM/IEEE Design Automation Conference, pp. 662-667, 1989.

[3] Asher Wilk and Amir Pnueli, "Specification and Verification of VLSI Systems", Proceedings of the 7th IEEE International Conference on Computer Aided Design, pp. 460-463, 1989.

[4] B.J.S. De Loore and A.P. Kostelijk, "Automatic Verification of Library-based IC Designs", Proceedings of the IEEE Custom Integrated Circuits Conference, pp. 30.6.1-30.6.5 , 1990.

[5] Richard H. Lathrop, Robert J. Hall, Gaven Duffy, K. Mark Alexander, Robert S. Kirk, "Advances in Functional Abstraction From Structure", Proceedings of the 25th ACM/IEEE Design Automation Conference, pp 708-711, 1988.

[6] Jean Christophe Madre, Oliver Coudert, Jean Paul Billon, "Automating the Diagnosis and the Rectification of Design Errors with PRIAM", Proceedings of the 7th IEEE International Conference on Computer Aided Design, pp. 30-33, 1989.

[7] C. Leonard Berman, Louise H. Trevilllyan, "Functional Comparison of Logic Designs for VLSI Circuits", Proceedings of the 7th IEEE International Conference on Computer Aided Design, pp. 456-459, 1989.

[8] Inderpreet Bhasin, "Recognition of Logic Blocks in CMOS Circuits", Masters Thesis at Virginia Polytechnic and State University, 1988.

[9] Vasant B. Rao and Timothy N. Trick, "Network Partitioning and Ordering for MOS VLSI Circuits", IEEE Transactions on Computer-Aided Design, Vol. 1, pp. 128-144, January 1987.

[10] F. Luellau, T. Hoepken, E. Barke, "A Technology Independent Block Extraction Algorithm", Proceedings of the 21rst Design Automation Conference, pp. 610-615, 1984.

[11] K. Imafuji, M. Nagano, K. Ogama, M. Iwatzuki, "Conversion Systems from Pattern Layouts to Logic Equation (PALLEQ)", Proceedings of the 4th International Conference and Exhibition on Computers inDesign Engineering, pp. 438-445, 1980.

[12] G. Russel, "Automatic Mask Function Checking of LSI Circuits", Proceedings of the 2nd International Conference and Exhibition on Computers in Design Engineering, pp. 182-194, 1978.

[13] H. Kawanishi, A. Kishimoto, K. Kani, "An Automatic Layout-Logic Verification Algoritym for VLSI", Proceedings of the European Conference on Circuit Theory and Design, pp. 436-447, 1980.

[14] L. Scheffer and R. Apte, "LSI Design Verification Using Topology Extraction", Proceedings of the 12th Asilomar Conference on Circuits, Systems, and Computers, pp. 149-153, 1978.

[15] J. A. Bondy and U:. S. R. Murty, Graph Theory with Applications, New York: North-Holland, 1982.

Appendix

```
/**********************************************************************************/
/*                                                                                */
/*      Program Name:     Circuit Recognition Experimental Program               */
/*                                                                                */
/*      Programmer :      Glenn Griffin                                          */
/*                                                                                */
/*      Description :     This program was written to satisfy part of the       */
/*                         requirements of a Masters of Science Degree in EE.   */
/*                        The program is used to compare the relative           */
/*                        efficiency of and intelligent and non intelligent     */
/*                        search techniques.  The program prompts the user      */
/*                        for an input circuit file an then reads and           */
/*                        partitions the entered file.  The program then        */
/*                        prompts the user for a library file.  After the       */
/*                        library file has been read the program asks which     */
/*                        search technique it should use.  After this answer    */
/*                        is supplied, the program performs the recognition     */
/*                        and reports how much time was consumed by the         */
/*                        selected technique.                                    */
/*                                                                                */
/*      Computer:         IBM 486 with 4 Meg of Memory with Rational Systems    */
/*                        DOS/4GW Dos Extender program                          */
/*                                                                                */
/*      Language:         Watcom C/386                                           */
/*                                                                                */
/*      Date:             April 26th, 1993                                       */
/*                                                                                */
/**********************************************************************************/

//############################################################################
// Definitions Section
//############################################################################

// Constant Definitions

#define HASHVALUE          12547    // mod 12547 in the open hash table
#define MAXNODES           12547    // maximum node allowed 12547 increase for
                                    // for larger circuits
#define S_CHARS     12     // maxsignif chars in ascii nodenames
#define MAXOBJS     500    // max num of different object types!! types!!

                                                    // block type defines
#define PASS        0
#define DRIVER      1
#define LOAD        2

                                                    // ref circuit node name constants
#define REFVDD      1
#define REFGND      2
                                                    // reference circuit type defines
#define LOADDRIVER         3

// nodetype defines for partition and recognition procedures

#define NORMAL 0
#define EXTERNAL 1
#define PULLUP 2
```

```
#define INPUT 3

#include <stdio.h>                                          // for printf
#include <string.h>                                         // for string functions
#include <stdlib.h>                                         // for malloc
#include <time.h>                                           // for time and difftime


// this type is used for all node numbers and objectnumbers

typedef unsigned int UN_INT;


// ################################################################
// The following are declarations of the object type
//

typedef struct v_node {                                     // nodes used for adjacency list
        UN_INT          vertexnum;                          // they contain all trans info
        UN_INT          sourcenum;                          // then they are used in objects
        UN_INT          gatenum;
        char            ptype;                              // boolean value for trans type
        char            bound;                              // boolean flag used in recognition
        struct v_node * nextvertex;
} TRANS_NODE;

typedef struct i_node {
        UN_INT          key;
        struct i_node * next;
} INT_NODE;

typedef struct obnode {
        char            objectname[12];                     // user defined name
        UN_INT          objectnum;                          // assigned and used by program
        INT_NODE *      inputlist;                          // input node numbers
        INT_NODE *      outputlist;                         // output node numbers
        TRANS_NODE *    translist;                          // list of trans that makeup obj
        struct obnode * nextobject;                         // next object of similar type
} OBJECT;

//#########################################################################
// type defs for the recognition section
// ########################################################################
// node degree list defines
// note that a node is a vertex (source or drain) or a gate
// used by recognize routines

typedef struct nd_node {
        UN_INT                  nodenum;
        UN_INT                  degree;
        struct nd_node *        next;
        UN_INT  bound;                  // zero means unbound a node number means bound to that nodenumber
        UN_INT  numofbindings;          // the # of times the node has been bound
} ND_NODE;


// type defs for referrence library circuits

typedef struct r_trans {        // Reference Transistor Node Type
        UN_INT  tnum;           // used to descriminate between different transistors in the same list
                                // since pointer comparisons aren't working
        UN_INT  drain;          // the node numbers
        UN_INT  source;
        UN_INT  gate;
        char            ptype;          // boolean value for trans type
        TRANS_NODE *    edgeptr;        // points to edge that the reference
                                        // transistor is bound to
        struct r_trans * nextt;         // list next pointer
} R_TRANS;

typedef struct ref_node {
```

```
        char                objectname[12];
        UN_INT              objnum;
        char                objtype;            // load driver both or pass
        UN_INT              numoftrans;         // second descrimiantor
        UN_INT              vdddegree;          // third descriminator
        UN_INT              compsneeded;        // fix this later!!!! comp list?
        ND_NODE*            degreelist;         // final descriminator
        R_TRANS *           reftranslist ;      // edge connectivity info list
        struct ref_node *   nextref;            // ref list pointer
} REF_NODE;


// ####################################################################
// The following are the declarations for the open hash table
// structure used to store the text names of the nodes


typedef struct h_node {
        char                nodename[S_CHARS];
        UN_INT              nodenum;
        struct h_node *     nextnode;
} HASH_NODE;

typedef struct {
        HASH_NODE * lastaccess;
        HASH_NODE * firstnode;
} HASH_TABLE_NODE;


// ##########################################################################
// these declarations are for the array of adjacency lists representation
// of the graph of the electrical network


typedef struct g_node {
        char                ptype;
        struct g_node *     nextedge;
} GATE_LIST_NODE;

typedef struct op_node {
        OBJECT *            object;
        struct op_node *    nextobj;
} OPL_NODE;                                     // object pointer list node type

typedef struct {
        char                blocktype;
        UN_INT              blocknum;
        INT_NODE *          inputlist;
        INT_NODE *          outputlist;
        TRANS_NODE *        translist;
} BLOCK;

typedef struct bp_node {
        BLOCK * block;
        struct bp_node *    nextblk;
} BPL_NODE;


typedef struct {
        HASH_NODE *             nameptr;        // points to name in hash table
        char                    nodetype;       // normal , external, pullup ect.
        TRANS_NODE *            vertexlist;     // linked list of reachable vertices
        GATE_LIST_NODE *        gatelist;
        BPL_NODE *              blockptr;       // list of unclassified driver and load circiut blocks
        BLOCK *                 passblk;        // unclassified pass block pointer
        OPL_NODE *              outputobjects;  // list of all objects that node is an output of

        OPL_NODE *              inputobjects;   // list of all objects that node is a input to
} CIRCUIT_NODE;
```

```
// ####################################################################
// All variables used in this module (with static storage and
// internal linkage) are declared here

static HASH_TABLE_NODE    HashTable[HASHVALUE];    // array of hashtablenodes

static UN_INT             NodeCount = 0;             // number of distinct nodes found
static CIRCUIT_NODE       CircuitGraph[MAXNODES];    // graph represenation of network

static INT_NODE *         CompTable[MAXNODES];       // table of complement nodes

static UN_INT             vdd,gnd;

UN_INT                    BlockCount = 0;            // number of partitioned blocks created


static UN_INT             ObjectTypeCount = 2;       // the number of different object types
static OBJECT *           ObjectTable[MAXOBJS];      // open hash table of object types
static REF_NODE *   RefObjectList = NULL;            // the sorted linked list of reference circuits


//##################################################################
// external function prototypes declared here

extern void HashTableInit();

extern UN_INT GetNodeNum(char nodelabel[S_CHARS]);

extern void InsertEdge(UN_INT source, UN_INT drain, UN_INT gate, char ptype);

extern void UpdateGateList(UN_INT gate, char ptype);

extern void FindInverters();

extern void FindTransmissionGates();

extern void ObjectTableInit();

extern void CompTableInit();

extern void ClassifyNodes();

extern void PartitionCircuit();

extern void ReadRefCircuits();

extern void CreateNand();

extern void LDBlocksRecognize();

extern void CountObjects ();

main ()
{
        char      filename[12];
        int       sourcenum,drainnum,gatenum,ptype;
        int       nlength;
        char      source[12],drain[12],gate[12],type;
        char      buffer[80];
        char * pstart;

        FILE      *fileptr;

        time_t start, fileclosed;
        clock_t partitioned, ldblocksrec;

        int i = 0;

        char circfile[12];
```

```
char mode;

printf("\n\nWelcome to the circuit recognition experimental program.\n\n");

printf("Please enter the circuit filename that you wish to process.\n");
scanf( "%s", circfile);


fileptr = fopen( circfile, "r");                         // open circuit file for reading

HashTableInit();                                          // initialize data structures
ObjectTableInit();


// Read all transistors in input file and insert them into the graph

while ( fgets(buffer, 80, fileptr) != NULL) {

        pstart = buffer;
        while ((*pstart) == ' ') pstart++;                // strip blanks

        if (strncmp( "trans(", pstart, 6) == 0) {
                pstart = pstart + 6;
                while ((*pstart) == ' ') pstart++;        // strip blanks

                nlength = strcspn(pstart,", ");
                strncpy( source, pstart, nlength);        // get source
                strcpy(&source[nlength],"");

                pstart = pstart + nlength -1;
                while ((*pstart) != ',') pstart++;        // go to next string
                pstart++;
                while ((*pstart) == ' ') pstart++;

                nlength = strcspn(pstart,", ");
                strncpy( drain, pstart, nlength);         // get drain
                strcpy(&drain[nlength],"");

                pstart = pstart + nlength-1;
                while ((*pstart) != ',') pstart++;        // go to next string
                pstart++;
                while ((*pstart) == ' ') pstart++;

                nlength = strcspn(pstart,") ");
                strncpy( gate, pstart, nlength);          // get gate
                strcpy(&gate[nlength],"");

                pstart = pstart + nlength-1;
                while ((*pstart) != ')') pstart++;        // go to next string
                pstart++;
                while ((*pstart) == ' ') pstart++;

                type = *pstart;


                sourcenum = GetNodeNum(source);                   // Hash Table name search
                drainnum = GetNodeNum(drain);
                gatenum = GetNodeNum(gate);

                ptype = (type == 'p');                            // set p-type flag

                                                                  // do graph insertion
                InsertEdge(sourcenum, drainnum, gatenum, ptype);
                InsertEdge(drainnum, sourcenum, gatenum, ptype);

                UpdateGateList(gatenum, ptype);
        }
}

vdd = GetNodeNum("Vdd");                          // Retreive the node numbers that
gnd = GetNodeNum("GND");                          // been assigned to VDD and GND
```

```
        CompTableInit();                        // initialize the complement table

        FindInverters();                        // find all the inverters

        FindTransmissionGates();                // find all the transmision gates

        ClassifyNodes();                        // Classify the nodes so we can
                                                // perform partitioning

        PartitionCircuit();                     // Partition circuit into Blocks

        ReadRefCircuits();                      // Read in the Reference Library

        // Decide weather or not to use the inteligent or the brute force method

        printf("\nUse Intelligent Search Method (Y/N)?");
        scanf("%s", &mode);

        partitioned = clock();

        if ((mode == 'Y') || (mode == 'y')) LDBlocksRecognize();

        else BruteLDBlocksRecognize();

        ldblocksrec = clock();

        printf("\n");
        CountObjects();                         // Report on the objects found

        fclose( fileptr);
                                                // Report program run time

        printf("\n\nBlock Recognition Process finished in %f  seconds.\n\n",((float)(ldblocksrec - partitioned))/100.0 );
}



//###########################################################################
// internally used function prototypes are here
//
// A few of these functions access external variables
// these accesses are explicitly commented
//
//###########################################################################


// this function initializes all the lists in the hash table to empty

extern void HashTableInit()
{
        int i;

        for ( i=0 ; i <= (HASHVALUE - 1) ; i++) {
                HashTable[i].firstnode = NULL;
        }
}


// this function initializes all the lists in the object table to empty

extern void ObjectTableInit()
{
        int i;

        for ( i=0 ; i <= (MAXOBJS - 1) ; i++) {
```

```
			ObjectTable[i] = NULL;
		}
}



static UN_INT DoHashInsert(HASH_TABLE_NODE * htptr,char nodelabel[S_CHARS])
{
		HASH_NODE * newptr;

		newptr = (HASH_NODE *) malloc( sizeof(HASH_NODE) );

		if (newptr == NULL) {						// failed allocation

			printf("failed hashtable insert insufficient memory for malloc call\n");
			exit( EXIT_FAILURE );
		}
		else {									// initialize node values & insert
			strcpy(newptr->nodename, nodelabel);

			if (NodeCount > (MAXNODES - 2) ) {		// error !!!!!!!!

				printf("too many nodes in circuit, increase MAXNODES value\n");
				exit( EXIT_FAILURE );
			}
			else {

				// do hashtable insert and initialize new node in circuit graph
				// note that NodeCount[0] never gets used

				newptr->nodenum = ++NodeCount;			// !!! external var update

				CircuitGraph[NodeCount].nameptr = newptr;  // graph rep points to name
				CircuitGraph[NodeCount].nodetype = NORMAL;
				CircuitGraph[NodeCount].vertexlist = NULL; // new name gets new node
				CircuitGraph[NodeCount].gatelist = NULL;  // in circuit graph
				CircuitGraph[NodeCount].blockptr = NULL;
				CircuitGraph[NodeCount].passblk = NULL;
				CircuitGraph[NodeCount].outputobjects = NULL;
				CircuitGraph[NodeCount].inputobjects = NULL;


				newptr->nextnode = htptr->firstnode;		// insert at front of hash list
				htptr->firstnode = newptr;
				htptr->lastaccess = newptr;		// update last access
			}
		}
		return NodeCount;
}



// the function first generates the hash value then checks
// whether the node name is in the table and if not, inserts it, and returns
// its nodenumber

extern UN_INT GetNodeNum(char nodelabel[S_CHARS])
{
		int				strlength;
		int				hx = 0;			// hx is the h(x) hashvalue
		int				i,found;
		UN_INT		returnnum;
		HASH_TABLE_NODE * htptr;
		HASH_NODE *	nptr;

		strlength = strlen(nodelabel);
		for ( i=0 ; (i < strlength) ; i++) {			// compute h(x)
			hx = 10 * hx + (nodelabel[i] - '0');
// printf(" %i %i %i,",hx,nodelabel[i],'0');
		}
		hx = abs(hx % HASHVALUE);			// mod operation
```

```
	htptr = &HashTable[hx];

// printf("s = %s , hx = %i \n",nodelabel,hx);

	if (htptr->firstnode != NULL) {

		if (strcmp(htptr->lastaccess->nodename, nodelabel ) == 0) {
			returnnum = htptr->lastaccess->nodenum;
//printf(".");
//printf("found quickly\n");
		}
		else {								// look for node in list
			nptr = HashTable[hx].firstnode;
			found = 0;
			while ( (nptr != NULL) &&
			!( found = (strcmp(nptr->nodename, nodelabel) == 0) ) ) {

				nptr = nptr->nextnode;
			}
			if (found) {
//printf("found less quickly\n");
				returnnum = nptr->nodenum;
				htptr->lastaccess = nptr;
			}
			else {				// insert a new node
				returnnum = DoHashInsert( htptr, nodelabel);
//printf("not found inserting\n");
			}
		}
	}
	else {												// insert
new node
		returnnum = DoHashInsert( htptr, nodelabel);
//printf("inserting first node\n");
	}
	return returnnum;								// return nodenum
}


// used to build cirucit graph

extern void InsertEdge(UN_INT source, UN_INT drain, UN_INT gate, char ptype)
{
	TRANS_NODE * newptr, * backptr, * curptr;

	newptr = (TRANS_NODE *) malloc( sizeof(TRANS_NODE) );

	if (newptr == NULL) {				// failed allocation

		printf("failed graph edge insert insufficient memory");
		exit( EXIT_FAILURE );
	}
	else {						// initialize node values & insert
		newptr->vertexnum = drain;	// this is the key
		newptr->sourcenum = source;		// this is not!!!! the key
		newptr->gatenum = gate;
		newptr->ptype = ptype;
												// insert in
decreasing order

		backptr = CircuitGraph[source].vertexlist;

		if ( (backptr == NULL) || (backptr->vertexnum <= drain) ) {
//printf(",");
			newptr->nextvertex = backptr;
			CircuitGraph[source].vertexlist = newptr;
		}
		else {	// thiss code will not be tested with inorder files!!!!!!!!!!!!!!!!!
```

```
                                curptr = backptr->nextvertex;
                                while ((curptr != NULL) && (curptr->vertexnum > drain)) {
                                        backptr = curptr;
                                        curptr = curptr->nextvertex;
                                }
                                backptr->nextvertex = newptr;
                                newptr->nextvertex = curptr;
                        }
                }
}


extern void UpdateGateList(UN_INT gate, char ptype)
{
        GATE_LIST_NODE * newptr;

        newptr = (GATE_LIST_NODE *) malloc( sizeof(GATE_LIST_NODE) );

        if (newptr == NULL) {                           // failed allocation

                printf("failed graph gate list update insufficient memory");
                exit( EXIT_FAILURE );
        }
        else {                                                  // initialize node values & insert

                newptr->ptype = ptype;
                                                                                        // insert at
front

                newptr->nextedge = CircuitGraph[gate].gatelist;
                CircuitGraph[gate].gatelist = newptr;
        }
}


static int InIntList( INT_NODE * headptr, UN_INT keyvalue)
{
        int             found;

        found = 0;
        while ((headptr != NULL) && (!(found))) {

                found = (headptr->key == keyvalue);
                headptr = headptr->next;
        }
        return found;
}


// calling with void, key returns a pointer to a one node int list

static INT_NODE * InsertIntList( INT_NODE * headptr, UN_INT keyvalue)
{
        INT_NODE *          newptr;

        newptr = (INT_NODE *) malloc( sizeof(INT_NODE) );

        if (newptr == NULL) {                           // failed allocation

                printf("failed integer list creation insufficient memory");
                exit( EXIT_FAILURE );
        }
        else {                                                  // initialize node values & insert at the front

                newptr->key = keyvalue;
                newptr->next = headptr;

        }
```

```
        return newptr;                                                  // return pointer to front of list
}

// this function initializes all the lists in the complement table to empty
// then sets up vdd and gnd as complements

extern void CompTableInit()
{
        int i;

        for ( i=0 ; i <= (MAXNODES - 1) ; i++) {
                CompTable[i] = NULL;
        }

        CompTable[vdd] = InsertIntList(CompTable[vdd], gnd);
        CompTable[gnd] = InsertIntList(CompTable[gnd], vdd);

}



static void AssertComplement(UN_INT node1, UN_INT node2)
{
        if ( InIntList(CompTable[node1], node2) ) {

                // do nothing, the complement has already been asserted
        }
        else {
                // add complement value..  this is an external reference!!!

                CompTable[node1] = InsertIntList(CompTable[node1], node2);
                CompTable[node2] = InsertIntList(CompTable[node2], node1);
        }
}



// calling with void, tpr returns a pointer to a one node trans list
// this routine doesn't allocate memory!!

static TRANS_NODE * InsertTransList( TRANS_NODE * headptr, TRANS_NODE * tptr)
{

        tptr->nextvertex = headptr;

        return tptr;                                    // return pointer to front of list
}



// calling with void, key returns a pointer to a one node object pointer list

static OPL_NODE * InsertOp_Node(OPL_NODE * headptr, OBJECT * objptr)
{
        OPL_NODE * newptr;

        newptr = (OPL_NODE *) malloc( sizeof(OPL_NODE) );

        if (newptr == NULL) {                           // failed allocation

                printf("failed object pointer list creation, insufficient memory");
                exit( EXIT_FAILURE );
        }
        else {                                          // initialize node values & insert at the front

                newptr->object = objptr;
                newptr->nextobj = headptr;

        }
        return newptr;                                                  // return pointer to front of list
}
```

```
// Creates and inserts object in the objecttable

static OBJECT * CreateObject (char objname[12], UN_INT objnum, INT_NODE * ilist,
                                                    INT_NODE * olist, TRANS_NODE * tptr)

{
        OBJECT * newptr;

        newptr = (OBJECT *) malloc( sizeof(OBJECT) );

        if (newptr == NULL) {                               // failed allocation

                printf("failed object creation insufficient memory");
                exit( EXIT_FAILURE );
        }
        else {                                              // initialize node values
                strcpy(newptr->objectname, objname);
                newptr->objectnum = objnum;
                newptr->inputlist = ilist;
                newptr->outputlist = olist;
                newptr->translist = tptr;

        // insert at the front

        // external ref!!!!!
                newptr->nextobject = ObjectTable[objnum];
                ObjectTable[objnum] = newptr;

        }
        return newptr;                                      // return pointer to object
}


// Counts the number of objects in the objecttable

extern void CountObjects ()

{
        OBJECT * objptr;
        int             i, count, totl;

        totl = 0;

        printf("Recognition Report\n\n");

        for ( i=0; i<=(MAXOBJS - 1); i++ ) {

                objptr = ObjectTable[i];
                count = 0;
                while ( objptr != NULL ) {

                        count++;
                        objptr = objptr->nextobject;
                }

                if (count > 0) {
                        printf("%d %s found.\n", count, ObjectTable[i]->objectname);

                        totl = totl + count;
                }

        }
        printf("\n\nTotal Gates Recognized = %d.\n",totl);
}


// removes node from list but doesn't un allocate the node's memory
```

```
// node must!!!!!! be present or else routine will crash!!!!
// accesses CircuitGraph[source]

static TRANS_NODE * DeleteEdge(UN_INT source, UN_INT drain, UN_INT gate, char ptype)
{
	TRANS_NODE * cptr, * bptr;				// current and previous pointers

	cptr = CircuitGraph[source].vertexlist;

	if ( (cptr->vertexnum == drain) && (cptr->gatenum == gate) &&
	(cptr->ptype == ptype) ) {

				// remove node
		CircuitGraph[source].vertexlist = cptr->nextvertex;
	}
	else {
		bptr = cptr;
		cptr = cptr->nextvertex;
		while ( (cptr->vertexnum != drain) || (cptr->gatenum != gate) ||
		(cptr->ptype != ptype) ) {

			bptr = cptr;
			cptr = cptr->nextvertex;
		}

		bptr->nextvertex = cptr->nextvertex;			// remove node
	}
	return cptr;						// return pointer to node
}


// is called by FindInverter when an inverter is known to exist
// the gnd and vdd edges has already been deleted by deletevdd & deletegnd

static void AssertInverter(UN_INT drain, UN_INT gate)
{
	TRANS_NODE *	v_ptr;
	TRANS_NODE *	tptr;									// trans list ptr
	INT_NODE *			inputlist;
	INT_NODE *			outputlist;
	OBJECT *		objptr;

													// make
input and output lists
	inputlist = InsertIntList( NULL, gate);
	outputlist = InsertIntList( NULL, drain);

					// now make the trans list from the nodes in the vertex list

	tptr = NULL;								// empty trans list init

	v_ptr = CircuitGraph[drain].vertexlist;			// set pointer to head
	while (v_ptr != NULL) {

		if ( (v_ptr->vertexnum == vdd) && (v_ptr->gatenum == gate) &&
		  (v_ptr->ptype) ) {

	// advance ptr to a node that won't be deleted from the vertex list

			v_ptr = v_ptr->nextvertex;

	// delete edge from circuit graph vertex list, add it to the trans list

			tptr = InsertTransList(tptr, DeleteEdge(drain, vdd, gate, 1) );
		}
		else if ( (v_ptr->vertexnum == gnd) && (v_ptr->gatenum == gate)
		&& (!(v_ptr->ptype)) ) {

			v_ptr = v_ptr->nextvertex;
```

```
                        tptr = InsertTransList(tptr, DeleteEdge(drain, gnd, gate, 0) );

                }
                else {                          // advance past edge to trans gate or something
                        v_ptr = v_ptr->nextvertex;
                }
        }
                // Create inverter object and set input and output object pointers

        objptr = CreateObject("inverter", 0, inputlist, outputlist, tptr);


        // add to output list
        CircuitGraph[drain].outputobjects = InsertOp_Node(CircuitGraph[drain].outputobjects , objptr);


        // addto inputslist
        CircuitGraph[gate].inputobjects = InsertOp_Node(CircuitGraph[gate].inputobjects , objptr);

                                                                                // assert the complent in comp
table
        AssertComplement(gate, drain);
        AssertComplement(drain, gate);

}



static TRANS_NODE * * BackGndPtr;


// deletes and frees the current node matching nodes

static void DeleteGnd(UN_INT curdrain, UN_INT curgate)
{
        TRANS_NODE *    p;
        TRANS_NODE *    v_ptr;
        int                     found;

        v_ptr = *BackGndPtr;                                            // reset to current position

        if ( (v_ptr == NULL) || (v_ptr->vertexnum < curdrain) ) {

                                // go back to beginning of gnd list and start search over

                // external ref!!!
                BackGndPtr = &(CircuitGraph[gnd].vertexlist);           // reset pointer;
                v_ptr = *BackGndPtr;
        }

        found = 0;
        while ( (v_ptr != NULL) && (!(found)) &&
                        (v_ptr->vertexnum >= curdrain) ) {

                found = (v_ptr->vertexnum == curdrain) && (v_ptr->gatenum == curgate)
                                && (!(v_ptr->ptype));

                if (!(found)) {
                        BackGndPtr = &(v_ptr->nextvertex);
                        v_ptr = v_ptr->nextvertex;
                }
        }
        while ( (v_ptr != NULL) && (v_ptr->vertexnum == curdrain) &&
                        (v_ptr->gatenum == curgate) && (!(v_ptr->ptype)) ) {

                p = v_ptr;
                *BackGndPtr = (**BackGndPtr).nextvertex;
                v_ptr = *BackGndPtr;

                free( p );                                              // release memory
        }
```

```
}


// returns true if an ntype edge between curdrain and ground with input
// curgate exists;  and deletes the trans !

static int GroundEdge(UN_INT curdrain, UN_INT curgate)
{
	TRANS_NODE *	v_ptr;
	int				found;

	v_ptr = CircuitGraph[curdrain].vertexlist;			// reset pointer

	found = 0;
	while ( (v_ptr != NULL) && !(found) ) {

		found = (v_ptr->vertexnum == gnd) && (v_ptr->gatenum == curgate)
				&& (!(v_ptr->ptype));

			v_ptr = v_ptr->nextvertex;
	}
	if (found) {
		DeleteGnd(curdrain, curgate);
	}
	return found;
}


static TRANS_NODE * * BackVddPtr;						// External Variable!

// may nexer be called with a NULL pointer!!!!!!!!!!!

static TRANS_NODE * NextVdd(TRANS_NODE * curptr)
{
	BackVddPtr = &(curptr->nextvertex);

	return *BackVddPtr;
}

// deletes and frees the current node

static TRANS_NODE * DeleteVdd( void )
{
	TRANS_NODE * p;

	p = *BackVddPtr;
	*BackVddPtr = (**BackVddPtr).nextvertex;

	free( p );							// release memory
//printf("freevdd");
	return *BackVddPtr;
}


extern void FindInverters ()
{
	UN_INT				drain;			// holds possible drain of inv
	UN_INT				gate;
	TRANS_NODE *		v_ptr;

int invcount =0;

	// 2 external refss!!!
	BackGndPtr = &(CircuitGraph[gnd].vertexlist);
	BackVddPtr = &(CircuitGraph[vdd].vertexlist);		// init back vdd pointer


	v_ptr = CircuitGraph[vdd].vertexlist;				// set pointer to head
```

```
	while (v_ptr != NULL) {

			drain = v_ptr->vertexnum;
			gate = v_ptr->gatenum;

			if ( (v_ptr->ptype) && (GroundEdge(drain, gate)) ) {

			// advance ptr to a good node that won't be deleted by AssertInverter

				while ( (v_ptr != NULL) && (v_ptr->vertexnum == drain) &&
						(v_ptr->gatenum == gate) && (v_ptr->ptype) ) {

					v_ptr = DeleteVdd();
				}

invcount++;
//printf("f i %i\n",invcount);

				AssertInverter(drain, gate);
				CircuitGraph[drain].nodetype = PULLUP;			// set nodetype

			}
			else {
				v_ptr = NextVdd(v_ptr);
			}
	}
// printf("found %i inverters\n",invcount);
}


// General Purposed Transister List Iterator with fast delete capabillity
// note Back Ptr must be initialized by calling program!!!

static TRANS_NODE * * BackPtr;


// may nexer be called with a NULL pointer!!!!!!!!!!!

static TRANS_NODE * NextTrans(TRANS_NODE * curptr)
{
	BackPtr = &(curptr->nextvertex);

	return *BackPtr;
}

// deletes and frees the current node returns pointer to node after
// the deleted node

static TRANS_NODE * DeleteCurrentTrans( void )
{
	TRANS_NODE * p;

	p = *BackPtr;
	*BackPtr = (**BackPtr).nextvertex;

	free( p );						// release memory
	return *BackPtr;
}


// deletes and frees 1/2 of the edges, deletes and makes trans
// list with other half

static TRANS_NODE * AssertTransGate( UN_INT source, UN_INT drain, UN_INT gate1, char gate1type, UN_INT gate2)
{
	TRANS_NODE *	v_ptr;
	TRANS_NODE *	tptr;						// trans list ptr
	INT_NODE *		inputlist;
```

```
	INT_NODE *			outputlist;
	OBJECT *		objptr;
	char				ptype;
	UN_INT				gate;

											// make
input and output lists
	inputlist = InsertIntList( NULL, gate1);
	inputlist = InsertIntList( inputlist, gate2);

	outputlist = InsertIntList( NULL, drain);
	outputlist = InsertIntList( outputlist, source);

				// now make the trans list from the nodes in the vertex list

	tptr = NULL;								// empty
trans list init

	// external ref!!!
	BackPtr = &(CircuitGraph[source].vertexlist);		// init back pointer

	v_ptr = CircuitGraph[source].vertexlist;		// set pointer to head

	while ( (v_ptr != NULL) && (v_ptr->vertexnum >= drain) ) {

		if (v_ptr->vertexnum == drain) {			// possible match

			if ( ((v_ptr->gatenum == gate1) && (v_ptr->ptype == gate1type)) ||
				((v_ptr->gatenum == gate2) && (v_ptr->ptype != gate1type)) ) {

				ptype = v_ptr->ptype;			// get the type and gate
				gate = v_ptr->gatenum;

	// delete and free current node and advance ptr to next node

				v_ptr = DeleteCurrentTrans();

	// delete edge from circuit graph vertex list, add it to the trans list

				tptr = InsertTransList( tptr, DeleteEdge(drain, source, gate, ptype) );
			}
			else {					// advance past current edge

				v_ptr = NextTrans(v_ptr);
			}
		}
		else {						// advance past current edge

			v_ptr = NextTrans(v_ptr);
		}
	}

		// Create trans gate object and set input and output object pointers

	objptr = CreateObject("transgate", 1, inputlist, outputlist, tptr);


	// add to outputs list
	CircuitGraph[drain].outputobjects = InsertOp_Node(CircuitGraph[drain].outputobjects , objptr);
	CircuitGraph[source].outputobjects = InsertOp_Node(CircuitGraph[source].outputobjects , objptr);


	// addto inputslist
	CircuitGraph[drain].inputobjects = InsertOp_Node(CircuitGraph[drain].inputobjects , objptr);
	CircuitGraph[source].inputobjects = InsertOp_Node(CircuitGraph[source].inputobjects , objptr);


	v_ptr = CircuitGraph[source].vertexlist;			// set pointer to head

							// advance to position after the last deletion
```

```
	while ( (v_ptr != NULL) && (v_ptr->vertexnum > drain) ) {
		v_ptr = v_ptr->nextvertex;
	}

	return v_ptr;
}



// this finds complement values that are directly asserted in the table
// and finds complement values from the fact that C a,b & C b,c & C c,d
// implies C a,d .... it is tricky recursive code that works on the
// comp table and uses these global variables

static UN_INT N2;							// the search value
static UN_INT N2NotFound;						// recursion control
static UN_INT Comp;							// recursive result
static INT_NODE * VisitedNodes;						// recursion control

// the value of level is the key to this routine.  odd levels imply
// complement values and even levels imply non complement or equality

static void SearchForN2(UN_INT n1, UN_INT level)
{
	INT_NODE *		iptr;

	if ( (N2NotFound) && (! (InIntList( VisitedNodes, n1 )) ) ) {

printf(" N2NotFound = True ");

		iptr = CompTable[n1];
		while ( (N2NotFound) && (iptr != NULL) ) {

			N2NotFound = (iptr->key != N2);
			iptr = iptr->next;
		}

		if ( !(N2NotFound) ) {						// we're done

			Comp = ((level % 2) == 1);
		}
		else {
										// check any
comp(n1,x) possiblities
										// unless n1 has been
visited before

			if (! (InIntList( VisitedNodes, n1 )) ) {

				VisitedNodes = InsertIntList(VisitedNodes, n1);

				iptr = CompTable[n1];
				while ((N2NotFound) && (iptr != NULL)) {

					SearchForN2(iptr->key, (level + 1));
					iptr = iptr->next;
				}
			}
		}
	}
}


static int Complement(UN_INT node1, UN_INT node2)
{
//printf(" Complement Called ");

	Comp =  InIntList(CompTable[node1], node2);
	if (! (Comp) ) {
//printf(" Comp inititialy false ");
```

```
                    if ( (CompTable[node1] != NULL) && (CompTable[node2] != NULL) ) {

                            VisitedNodes = NULL;
                            N2 = node2;
                            N2NotFound = 1;

                            SearchForN2(node1, 1);
                    }
                    else {                                      // comp was already set to zero anyway!!
                            Comp = 0;
                    }
            }
//if ( Comp ) { printf(" Comp is True "); } else { printf(" Comp is False "); }
            return ( Comp );
}



// step through the CircuitGraph vertex lists and check for transmission
// gates

extern void FindTransmissionGates ()
{
            UN_INT                          drain;                  // holds possible drain of transgate
            UN_INT                          source;
            UN_INT                          gate;
            TRANS_NODE      *v_ptr;
            TRANS_NODE              *       nextv;

int transmiscount =0;

            // 1 external ref!!!
            for (source = 1 ; source <= NodeCount ; source++) {

//printf("source = %u \n",source);

                    v_ptr = CircuitGraph[source].vertexlist;        // set pointer to head

                    while ( (v_ptr != NULL) && ((drain = v_ptr->vertexnum) > source) &&
                                    ( (nextv = (v_ptr->nextvertex)) != NULL ) ) {

                            if ( ( v_ptr->ptype != nextv->ptype ) &&
                                    ( drain == nextv->vertexnum ) &&
                                    ( Complement(v_ptr->gatenum, nextv->gatenum) ) ) {

                                    // Tramission Gate exists call AssertTransGate

                                    v_ptr = AssertTransGate(source, drain, v_ptr->gatenum, v_ptr->ptype, nextv->gatenum);
transmiscount++;
//printf("f t %i \n",transmiscount);

                            }
                            else {
//printf(".");
                                    v_ptr = v_ptr->nextvertex;
                            }
                    }
            }
//printf("found %i transmission gates\n",transmiscount);
}



static void FindPullUps()
{
int pcount=0;
            UN_INT  i;
            char    type1;
            TRANS_NODE *    vptr;
```

```
		for ( i = 0 ; i <= NodeCount ; i++) {

				vptr = CircuitGraph[i].vertexlist;
				if (vptr != NULL) {

						type1 = vptr->ptype;
						vptr = vptr->nextvertex;

						while ((vptr != NULL) && (vptr->ptype == type1)) {
								vptr = vptr->nextvertex;
						}
//printf(" %i ",i);
						if ((vptr != NULL) && (vptr->ptype != type1)) {

																		// a pull up node has
been found
pcount++;
//printf("%i, ",pcount);
								CircuitGraph[i].nodetype = PULLUP;
						}
				}
		}
//printf("%i pullup nodes found \n\n",pcount);
}

// At this point Classify nodes only recognizes pullups because this
// experiment was constrained to static CMOS, dynamic CMOS and pseudo-Nmos
// To expand the classify proceddure see Bhasin's thesis and Trick's paper

extern void ClassifyNodes()
{
		FindPullUps();
}



// these are intlists that are ordered ascending

// General Purposed Ordered Int List Iterator with fast insert & delete capabillity
// note IBack Ptr must be initialized by calling program!!!!

static INT_NODE * * IBackPtr;				// External Var!!!

// may nexer be called with a NULL pointer!!!!!!!!!!!

static INT_NODE * NextOInt(INT_NODE * curptr)
{
		IBackPtr = &(curptr->next);

		return *IBackPtr;
}

// deletes and frees the current node returns pointer to node after
// the deleted node

static INT_NODE * DeleteCurrentOInt( void )
{
		INT_NODE * p;

		p = *IBackPtr;
		*IBackPtr = (**IBackPtr).next;

		free( p );								// release memory
		return *IBackPtr;
}

static int InOIntList( INT_NODE * * adheadptr, UN_INT keyvalue)
{
		int		found;
		INT_NODE * curptr;
```

```
	found = 0;

	if ((*adheadptr) != NULL) {

		IBackPtr = adheadptr;					// init back pointer
		curptr = (*adheadptr);					// init current node

		while ((curptr != NULL) && (curptr->key < keyvalue)) {

			curptr = NextOInt(curptr);
		}
		if (curptr != NULL) {

			// then it points to a node that is >= keyvalue and the back
			// pointer is available for a quick insert or delete

			found = (curptr->key == keyvalue);
		}
	}
	return found;
}


static INT_NODE * CreateIntNode(UN_INT keyvalue)
{
	INT_NODE *	newptr;

	newptr = (INT_NODE *) malloc( sizeof(INT_NODE) );

	if (newptr == NULL) {				// failed allocation

		printf("failed integer list creation insufficient memory");
		exit( EXIT_FAILURE );
	}
	else {
		newptr->key = keyvalue;
	}
	return newptr;
}


// calling with pointer to a void pointer, and key
// returns a pointer to a one node int list

static INT_NODE * InsertOIntList( INT_NODE * * adheadptr, UN_INT keyvalue)
{
	INT_NODE *	newptr;

	if ((*adheadptr) == NULL) {			// empty list create one node list

		newptr = CreateIntNode(keyvalue);
		newptr->next = NULL;
		(*adheadptr) = newptr;			// external reference!!!!! tricky
	}
	else {							// look for node in list

		if ( !(InOIntList( adheadptr, keyvalue)) ) {

									// then insert it using the back
pointer

			newptr = CreateIntNode(keyvalue);	// tricky!!!!!!
			newptr->next = *IBackPtr;
			*IBackPtr = newptr;
		}
	}
	return *adheadptr;
}

// calling with void, key returns a pointer to a one node block pointer list
```

```
static BPL_NODE * InsertBp_Node(BPL_NODE * headptr, BLOCK * blkptr)
{
        BPL_NODE * newptr;

        newptr = (BPL_NODE *) malloc( sizeof(BPL_NODE) );

        if (newptr == NULL) {                           // failed allocation

                printf("failed block pointer list creation, insufficient memory");
                exit( EXIT_FAILURE );
        }
        else {                                          // initialize node values & insert at the front

                newptr->block = blkptr;
                newptr->nextblk = headptr;

        }
        return newptr;                                  // return pointer to front of list
}


// Creates a Block  which is an unrecognized subcicuit created durring the
// partitioning process it increments the global variable BlockCount and
// assigns the new value to this blocknum, this is done so that in other
// parts of the program we can tell if we are pointing to the same block
// because pointer comparisons sometimes donot work

static BLOCK * CreateBlock (char blocktype, INT_NODE * ilist,
                                        INT_NODE * olist, TRANS_NODE * tptr)

{
        BLOCK * newptr;

        newptr = (BLOCK *) malloc( sizeof(BLOCK) );

        if (newptr == NULL) {                           // failed allocation

                printf("failed block creation insufficient memory");
                exit( EXIT_FAILURE );
        }
        else {                                          // initialize node values
                BlockCount++;
                newptr->blocktype = blocktype;
                newptr->blocknum = BlockCount;
                newptr->inputlist = ilist;
                newptr->outputlist = olist;
                newptr->translist = tptr;
        }
        return newptr;                                  // return pointer to block
}


static char GetBlockType(INT_NODE * nodelist)
{
        UN_INT   pullups, inputs, externals,normal;
        INT_NODE *          iptr;

        char                blocktype;

        pullups = 0;
        inputs = 0;
        externals = 0;
normal=0;
        iptr = nodelist;
        while (iptr != NULL) {
                switch (CircuitGraph[iptr->key].nodetype) {

                        case PULLUP : pullups++; break;

                        case INPUT  : inputs++; break;
```

```
                    case EXTERNAL : externals++; break;

                    default : normal++; break;  // normal node
                }
                iptr = iptr->next;
        }
                                                                                    // now determine the block type

        if ( (pullups == 1) && (inputs == 0) && (externals == 0) ) {

                if ( InOIntList( &nodelist, vdd) ) {                    // load component
                        blocktype = LOAD;
//printf("L");
                }
                else if ( InOIntList( &nodelist, gnd) ) {       // driver component
                        blocktype = DRIVER;
//printf("D");
                }
                else {
        // pass component
printf("no gnd");
                        blocktype = PASS;
                }
        }
        else {                                                  // we have a pass type component
printf("more than 1 pullup");
                blocktype = PASS;
        }
        return blocktype;
}

// Searches for a block of type blocktype and either returns a pointer to
// a found block or creates a new block via a call to createblock


static BLOCK * SearchLorDBlock(UN_INT n_s, char blocktype)
{
        INT_NODE *          iptr;
        BPL_NODE *          bplptr;
        BLOCK *             blkptr;
        int                         found;

        bplptr = CircuitGraph[n_s].blockptr;
        found = 0;

        while ( (bplptr != NULL) && (! found) ) {

                if (bplptr->block->blocktype == blocktype) {
//printf(" Found ");
                        found = 1;
                }
                else {
                        bplptr = bplptr->nextblk;
                }
        }
        if (found) {
                blkptr = bplptr->block;
        }
        else {                                                                          // return null
                blkptr = NULL;
        }
        return blkptr;
}

// merge inputlists, outputlists and translists
// set passblk pointers of all nodes in blkptr2 outputlist to point to blkptr1
// discard *blkptr2

static void MergeBlocks(BLOCK * blkptr1, BLOCK * blkptr2)
{
```

```
	TRANS_NODE *	tptr, * tempptr;
	INT_NODE *			iptr;

	//  add the new transisters from block2 to the translist of block1

	tptr = blkptr2->translist;
	while (tptr != NULL) {

		blkptr1->inputlist = InsertOIntList( &(blkptr1->inputlist), tptr->gatenum );

		tempptr = tptr;
		tptr = tptr->nextvertex;
		blkptr1->translist = InsertTransList( blkptr1->translist, tempptr);
	}

	iptr = blkptr2->outputlist;
	while (iptr != NULL) {					// add externals, pullup, and input

		blkptr1->inputlist = InsertOIntList( &(blkptr1->inputlist), iptr->key );
		blkptr1->outputlist = InsertOIntList( &(blkptr1->outputlist), iptr->key );

		CircuitGraph[iptr->key].passblk = blkptr1;

		iptr = iptr->next;
	}

//!!! free(blkptr2); time consuming
}


static BLOCK * SearchForPass(INT_NODE * nodelist)
{
	BLOCK *		blkptr;

	blkptr = NULL;

	while (nodelist != NULL) {

		if ( (CircuitGraph[nodelist->key].nodetype >= EXTERNAL) &&
			(CircuitGraph[nodelist->key].passblk != NULL) ) {

			if (blkptr == NULL) {		// this is the first time a pass block
								// has been found set blkptr

				blkptr = CircuitGraph[nodelist->key].passblk;
			}
			else {					// merge blocks if neccessary

				if (blkptr->blocknum != CircuitGraph[nodelist->key].passblk->blocknum ) {
printf(" merging blocks ");
					MergeBlocks( blkptr, CircuitGraph[nodelist->key].passblk);
				}
			}
		}

		nodelist = nodelist->next;
	}

	return blkptr;
}


// this routine adds every external, input, or pullup node in nodelist
// to the input and output list of the block pointed to by blkptr
// it also sets the passblk pointers

static void UpDatePassBlock (BLOCK * blkptr, INT_NODE * nodelist)
{
	while (nodelist != NULL) {	// add externals, pullup, and input
```

```
			if (CircuitGraph[nodelist->key].nodetype >= EXTERNAL) {

				blkptr->inputlist = InsertOIntList( &(blkptr->inputlist), nodelist->key );
				blkptr->outputlist = InsertOIntList( &(blkptr->outputlist), nodelist->key );

				CircuitGraph[nodelist->key].passblk = blkptr;
			}
			nodelist = nodelist->next;
		}
}


static void  AssertBlock(UN_INT n_s, INT_NODE * nodelist, TRANS_NODE * pathlist)
{
		INT_NODE *		inputlist;					// ordered int lists
		INT_NODE *		outputlist;
		char			blocktype;
		TRANS_NODE * tptr, * tempptr;
		BLOCK *	blockptr;

		blocktype = GetBlockType(nodelist);

		if (blocktype == PASS) {

			blockptr = SearchForPass(nodelist);

			if (blockptr == NULL) {		// create new pass block and output list
										// create input and output lists
printf(" create new pass block ");
				inputlist = NULL;
				outputlist = NULL;

				tptr = pathlist;		// add all gate inputs to inputlist
				while (tptr != NULL) {
					inputlist = InsertOIntList( &inputlist, tptr->gatenum );
					tptr = tptr->nextvertex;
				}

				blockptr = CreateBlock (blocktype, inputlist, outputlist, pathlist);

										// add external, input and pullup nodes to
										// the input and output lists and
										// set their pass block pointers

				UpDatePassBlock( blockptr , nodelist);

			}
			else {						// add external, input and pullup nodes and
										// set their pass block pointers
printf(" update existing pass block ");
				UpDatePassBlock( blockptr , nodelist);

										//  add the new transisters to the translist and
										//  the gateinputs to the inputlist
				tptr = pathlist;
				while (tptr != NULL) {

					blockptr->inputlist = InsertOIntList( &(blockptr->inputlist), tptr->gatenum );

					tempptr = tptr;
					tptr = tptr->nextvertex;
					blockptr->translist = InsertTransList( blockptr->translist, tempptr);
				}
			}
		}
		else {							// load or driver

			blockptr = SearchLorDBlock(n_s, blocktype);

			if (blockptr == NULL) {		// create new block and output list
```

```
                                                                                    // create input and output lists
                        inputlist = NULL;
                        outputlist = NULL;

                        tptr = pathlist;
                        while (tptr != NULL) {
                                inputlist = InsertOIntList( &inputlist, tptr->gatenum );
                                tptr = tptr->nextvertex;
                        }

                        outputlist = InsertOIntList( &outputlist, n_s );

                        CircuitGraph[n_s].blockptr = InsertBp_Node( CircuitGraph[n_s].blockptr,
                                        CreateBlock (blocktype, inputlist, outputlist, pathlist));
                }
                else {                                  // just update the input list and the translist

                        tptr = pathlist;
                        while (tptr != NULL) {

                                blockptr->inputlist = InsertOIntList( &(blockptr->inputlist), tptr->gatenum );

                                tempptr = tptr;
                                tptr = tptr->nextvertex;
                                blockptr->translist = InsertTransList( blockptr->translist, tempptr);
                        }
                }
        }
}


// collects and asserts a block for a given n_s

static void PartitionComponent(UN_INT n_s)
{
        TRANS_NODE      * pathlist;                             // unordered trans list
        TRANS_NODE * tptr;                                              // points to transistor
        INT_NODE        * front;        // unordered int list
        INT_NODE        * fptr;
        INT_NODE        * nodelist;     // ordered int list
        UN_INT          n_f, n_i;

//int chaincount;

        pathlist = NULL;
        nodelist = NULL;                // initialize lists
        nodelist = InsertOIntList(&nodelist, n_s);
        front = NULL;

        tptr = CircuitGraph[n_s].vertexlist;

//chaincount = 0;
        do {
//chaincount++;
                n_f = tptr->vertexnum;
                n_i = tptr->sourcenum;  // this will equal n_s the first time

                // delete and free duplicate Ei edge

                free (DeleteEdge(n_f, n_i, tptr->gatenum,  tptr->ptype));

                // delete other Ei edge from circuit and add it to pathlist

                pathlist = InsertTransList(pathlist, DeleteEdge(n_i, n_f, tptr->gatenum,
                                                                        tptr->ptype) );

                if ( (n_f == vdd) || (n_f == gnd) || ( InOIntList( &nodelist, n_f) ) ||
                        (CircuitGraph[n_f].nodetype >= PULLUP) || (n_f == n_s) ) {
                                                        // do nothing
                }
```

```
			else {

				// n_f has not been visited before add it to front

				front = InsertIntList(front, n_f);
			}

			// if n_f not in Nodelist add n_f to node list

			if ( !(InOIntList( &nodelist, n_f)) ) {
				nodelist = InsertOIntList(&nodelist, n_f);
			}

			while ((front != NULL) && (CircuitGraph[front->key].vertexlist == NULL) ) {
				fptr = front;
				front = front->next;	// delete node and release memory
				free(fptr);
			}

			if (front != NULL) {				// get next n_f

				tptr = CircuitGraph[front->key].vertexlist;
			}
		} while (front != NULL);
//printf("chaincount = %i, ",chaincount);
		AssertBlock(n_s, nodelist, pathlist);
}


extern void PartitionCircuit()
{
		int		i;
//TRANS_NODE * t;

		for (i = NodeCount ; i >= 1 ; i--) {

			if ( (CircuitGraph[i].nodetype >= PULLUP) && (i != vdd) && (i != gnd) ) {

								// node number i is an input node or a pullup node

				while (CircuitGraph[i].vertexlist != NULL) {
											// transistors exist that need partitioning
//t = CircuitGraph[i].vertexlist;
//printf(" | %u %u %u ",t->sourcenum,t->vertexnum,t->gatenum);
//if (t->nextvertex == NULL) {
//		printf(" NULL ");
//}
						PartitionComponent(i);
				}
//printf("\n");
			}
		}
printf("\nCircuit Partitioned into %u blocks.\n\n",BlockCount);
}


// ####################### Node Degree List Routines

// returns a pointer to node if found, else it returns null

static ND_NODE * InNDList( ND_NODE * headptr, UN_INT nodenum)
{
		while ((headptr != NULL) && (headptr->nodenum != nodenum)) {

			headptr = headptr->next;
		}
		return headptr;
}

// returns all nodes to the free list
```

```
// We manage our own memory in this case to save time.
// The nodes of memory are stored in a list pointed to by the following
// variable

ND_NODE * NDFreeList = NULL;

void FreeNDNodes ( ND_NODE * headptr)
{
        ND_NODE * temp;

        while (headptr != NULL) {

                temp = headptr;
                headptr = headptr->next;

                temp->next = NDFreeList;
                NDFreeList = temp;
        }
}


// searches for node in list and then increments its degree by one
// if node isn't found then a node is created and inserted
// since the list may be changed the routine must return a pointer to the list

static ND_NODE * IncDegree( ND_NODE * headptr, UN_INT nodenum)
{

        ND_NODE *       newptr;

        newptr = InNDList(headptr, nodenum);            // search for node in list

        if (newptr == NULL) {           // node not in list

                if (NDFreeList != NULL) {

                        newptr = NDFreeList;
                        NDFreeList = NDFreeList->next;
                }
                else {
                        newptr = (ND_NODE *) malloc( sizeof(ND_NODE) );
                }

                if (newptr == NULL) {                   // failed allocation

                        printf("failed node degree list creation insufficient memory");
                        exit( EXIT_FAILURE );
                }
                else {  // initialize node values & insert at the front

                        newptr->nodenum = nodenum;
                        newptr->degree = 1;
                        newptr->next = headptr;
                        headptr = newptr;
                }
        }
        else {                                                  // increment node degree
                (newptr->degree)++;
        }
        return headptr;                                 // return pointer to front of list
}


// General Purposed Ordered Node Degree List Iterator with delete capabillity
// note NDBack Ptr must be initialized by calling program or all hell breaks lose

static ND_NODE * * NDBackPtr;

// may nexer be called with a NULL pointer!!!!!!!!!!!
```

```
static ND_NODE * NextND (ND_NODE * curptr)
{
        NDBackPtr = &(curptr->next);

        return *NDBackPtr;
}

// removes current node from the list & doesnt free it or anything

static void RemoveCurrent( void )
{
        *NDBackPtr = (**NDBackPtr).next;
}


// removes the node with the smallest degree from the list
// list maynot be empty !!!!!!!!!!!!!!!

static ND_NODE * RemoveSmallest( ND_NODE * * adheadptr)
{
        int             smallest;
        ND_NODE * curptr;

                // go through the list and find the smallest degree

        curptr = (*adheadptr);
        smallest = curptr->degree;                                  // init smallest
        while (curptr != NULL) {

                if (smallest > curptr->degree) {
                        smallest = curptr->degree;
                }
                curptr = curptr->next;
        }

        NDBackPtr = adheadptr;                                           // init back pointer
        curptr = (*adheadptr);                                      // init current node

        while (curptr->degree != smallest) {

                        curptr = NextND(curptr);
        }
        RemoveCurrent();

        return curptr;                          // return pointer to smallest degree
}


// Sorts an unordered degree list into a desending degree list

static ND_NODE * SortDegreeList( ND_NODE * oldlist)
{
        ND_NODE * nodeptr;
        ND_NODE * newlist;

        newlist = NULL;
        while (oldlist != NULL) {

                nodeptr = RemoveSmallest( &oldlist);

                nodeptr->next = newlist;                   // add smallest to front
                newlist = nodeptr;
        }
        return newlist;
}


// this creates the degree list for a reference circuit and numbers
// the transistors in the input reference trans list
```

```
static ND_NODE * BuildRefDegreeList( R_TRANS * tlist, UN_INT * numoftrans)
{
	ND_NODE *		newlist;
	UN_INT			ntrans;

	ntrans = 0;
	newlist = NULL;
	while (tlist != NULL) {

		ntrans++;
		tlist->tnum = ntrans;					// number the transistor

		newlist = IncDegree( newlist, tlist->drain);
		newlist = IncDegree( newlist, tlist->source);
		newlist = IncDegree( newlist, tlist->gate);

		tlist = tlist->nextt;
	}

	(*numoftrans) = ntrans;						// return the number of trans

	return newlist;
}

static ND_NODE * BuildDegreeList( TRANS_NODE * tlist, UN_INT * numoftrans)
{
	ND_NODE *		newlist;
	UN_INT	ntrans;

	ntrans = 0;
	newlist = NULL;
	while (tlist != NULL) {

		ntrans++;

		newlist = IncDegree( newlist, tlist->vertexnum);
		newlist = IncDegree( newlist, tlist->sourcenum);
		newlist = IncDegree( newlist, tlist->gatenum);

		tlist = tlist->nextvertex;
	}

	(*numoftrans) = ntrans;						// return the number of trans

	return newlist;
}


// compares two degree lists if list 1 is greater than list2
// a 1 is returned, if eqaul a zero,  if less than -1

static int DListCompare( ND_NODE * list1, ND_NODE * list2)
{
	int returnvalue;

	while ( (list1 != NULL) && (list2 != NULL) &&
				(list1->degree == list2->degree) ) {
//printf("cl degree = %i, rl degree = %i |",list1->degree,list2->degree);
		list1 = list1->next;
		list2 = list2->next;
	}
	if (list1 == NULL) {
		if (list2 == NULL) {
			returnvalue = 0;
		}
		else {
			returnvalue = -1;
		}
	}
```

```
	else if (list2 == NULL) {					// list1 != NULL
		returnvalue = 1;
	}
	else {
		if (list1->degree > list2->degree) {
			returnvalue = 1;
		}
		else {
			returnvalue = -1;
		}
	}
	return returnvalue;
}


// this routine access an extern list pointer variable
// and trys to find a reference circuit that matches the descriminators
// that describe the object to be recognized
// if no possibilities exist then null is returned

static REF_NODE * FindPossibleCircuit( char		objtype, UN_INT		numoftrans, UN_INT vdddegree,
										UN_INT
compsneeded, ND_NODE * dlist)
{
	REF_NODE * curptr;

	if (RefObjectList == NULL) {						// empty ref list
printf("empty ref list");
		curptr = NULL;
	}
	else {
		curptr = RefObjectList;

//		while ( (curptr != NULL) && (objtype != curptr->objtype) ) {
//printf("advancing on object type");
//			curptr = curptr->nextref;
//		}

// removed the object type criterion in the below searching restore it
// for pass block recognition if program is expanded !!!!!!


		while ( (curptr != NULL) &&	(numoftrans > curptr->numoftrans) ) {
//printf("advancing on num of trans");

			curptr = curptr->nextref;
		}
		while ( (curptr != NULL) &&	(numoftrans == curptr->numoftrans) &&
					(vdddegree > curptr->vdddegree) ) {
//printf("advancing on vdddegree");

			curptr = curptr->nextref;
		}
		while ( (curptr != NULL) &&	(numoftrans == curptr->numoftrans) &&
						(vdddegree == curptr->vdddegree) &&
					( (DListCompare(dlist, curptr->degreelist)) > 0 ) ) {
//printf("advancing on degreelist ");

			curptr = curptr->nextref;
		}

//printf("dlistcompare = %i ", DListCompare(dlist, curptr->degreelist) );

		if ( (curptr != NULL) && (objtype == curptr->objtype) &&
					(numoftrans == curptr->numoftrans) &&
						(vdddegree == curptr->vdddegree) &&
					( (DListCompare(dlist, curptr->degreelist)) == 0 ) ) {

			// donothing curptr points to a possible match
		}
		else {						// return null
```

```
//printf(" no descriminator match returning null ");
                                    curptr = NULL;
                        }
            }
            return curptr;
}


// this routine looks at the next ref circuit from the current ref circuit
// if they are equal a pointer to the next ref circuit is returned
// else null is returned

static REF_NODE * FindNextPossibleCircuit( REF_NODE * curptr)
{
            if ( (curptr->nextref != NULL) &&
                        (curptr->nextref->objtype == curptr->objtype) &&
                        (curptr->nextref->numoftrans == curptr->numoftrans) &&
                        (curptr->nextref->vdddegree == curptr->vdddegree) &&
                        ( (DListCompare(curptr->nextref->degreelist, curptr->degreelist)) == 0 ) ) {

                        curptr = curptr->nextref;
            }
            else {                                                    // return null
                        curptr = NULL;
            }
            return curptr;
}


// Tricky reference object list iterator with fast insert capability

static REF_NODE * * RefBackPtr;

// may nexer be called with a NULL pointer!!!!!!!!!!!

static REF_NODE * NextRef (REF_NODE * curptr)
{
            RefBackPtr = &(curptr->nextref);

            return *RefBackPtr;
}


// this routine access an extern list pointer variable

static void  InsertRefObject( REF_NODE * rptr)
{

            char        objtype;
            UN_INT   numoftrans, vdddegree, compsneeded;
            ND_NODE *           dlist;

            REF_NODE * curptr;

            if (RefObjectList == NULL) {                                                        // empty ref list

                        RefObjectList = rptr;
                        rptr->nextref = NULL;
            }
            else {

                        RefBackPtr = &RefObjectList;                                                        // init back pointer
                        curptr = RefObjectList;

                        objtype = rptr->objtype;
                        numoftrans = rptr->numoftrans;
                        vdddegree = rptr->vdddegree;
                        compsneeded = rptr->compsneeded;              // currently unused feature
```

```
			dlist = rptr->degreelist;

			while ( (curptr != NULL) && (objtype != curptr->objtype) ) {

					curptr = NextRef(curptr);
			}
			while ( (curptr != NULL) && (objtype == curptr->objtype) &&
						(numoftrans > curptr->numoftrans) ) {

					curptr = NextRef(curptr);
			}
			while ( (curptr != NULL) && (objtype == curptr->objtype) &&
						(numoftrans == curptr->numoftrans) &&
						(vdddegree > curptr->vdddegree) ) {

					curptr = NextRef(curptr);
			}
			while ( (curptr != NULL) && (objtype == curptr->objtype) &&
						(numoftrans == curptr->numoftrans) &&
								(vdddegree == curptr->vdddegree) &&
						( (DListCompare(dlist, curptr->degreelist)) > 0 ) ) {

					curptr = NextRef(curptr);
			}

			// curptr now points at NULL or at a node that should follow the
			// new reference node to be inserted so we can do the insert


			rptr->nextref = curptr;						// tricky stuff again!!!
			(*RefBackPtr) = rptr;
		}
}

// Creates and inserts referece object in the reference object list

static void CreateRefObject (char objname[12], UN_INT objnum, char objtype,
 UN_INT  numoftrans, UN_INT vdddegree, UN_INT compsneeded,
 ND_NODE * degreelist, R_TRANS * reftranslist)

{
		REF_NODE * newptr;

		newptr = (REF_NODE *) malloc( sizeof(REF_NODE) );

		if (newptr == NULL) {					// failed allocation

			printf("failed reference object creation insufficient memory");
			exit( EXIT_FAILURE );
		}
		else {								// initialize node values
			strcpy(newptr->objectname, objname);

// printf("new = %s  input = %s \n",newptr->objectname, objname);

			newptr->objnum = objnum;
			newptr->objtype = objtype;
			newptr->numoftrans = numoftrans;
			newptr->vdddegree = vdddegree;
			newptr->compsneeded = compsneeded;
			newptr->degreelist = degreelist;
			newptr->reftranslist = reftranslist;

										// insert
new reference object
			InsertRefObject( newptr);
		}
}
```

```
// Creates a refernce trans node

static R_TRANS * CreateRefTrans(UN_INT source, UN_INT drain, UN_INT gate, char ptype)
{
	R_TRANS * newptr;

	newptr = (R_TRANS *) malloc( sizeof(R_TRANS) );

	if (newptr == NULL) {					// failed allocation

		printf("failed reference transistor creation insufficient memory");
		exit( EXIT_FAILURE );
	}
	else {							// initialize node values
		newptr->drain = drain;
		newptr->source = source;
		newptr->gate = gate;
		newptr->ptype = ptype;
	}
	return newptr;
}

// calling with void headptr, returns a pointer to a one node ref trans list

static R_TRANS * InsertRefTrans( R_TRANS * headptr, UN_INT source,
								UN_INT drain, UN_INT gate, char ptype)
{
	R_TRANS *		newptr;

	newptr = CreateRefTrans(source, drain, gate, ptype);

	newptr->nextt = headptr;

	return newptr;						// return pointer to front of list
}


// takes one trans list and tacks is on to the tail of the other list

static TRANS_NODE * MergeTransList(TRANS_NODE * list1, TRANS_NODE * list2)
{
	TRANS_NODE * newlist;

	if (list1 != NULL) {					// goto end and add list2

		newlist = list1;
		do {
			list1 = list1->nextvertex;

		} while (list1->nextvertex != NULL);

		list1->nextvertex = list2;
	}
	else {							// return list2 since
list1==NULL
		newlist = list2;
	}
	return newlist;
}


static int GetRefVddDegree( ND_NODE * ndlist)
{
	int			degree;

	while ( (ndlist != NULL) && (ndlist->nodenum != REFVDD) ) {

		ndlist = ndlist->next;
	}
```

```
	if (ndlist != NULL) {					// then vdd was found

		degree = ndlist->degree;
	}
	else {
		degree = 0;
	}
	return degree;
}


static int GetVddDegree( ND_NODE * ndlist)
{
	int			degree;

	while ( (ndlist != NULL) && (ndlist->nodenum != vdd) ) {

		ndlist = ndlist->next;
	}

	if (ndlist != NULL) {					// then vdd was found

		degree = ndlist->degree;
	}
	else {
		degree = 0;
	}
	return degree;
}


// returns the a pointer to the ND_NODE that contains a node number equal to
// nodenum, else it returns null

static ND_NODE * GetNDNode( ND_NODE * ndlist, UN_INT nodenum)
{
	while ( (ndlist != NULL) && ( ndlist->nodenum != nodenum ) ) {

		ndlist = ndlist->next;
	}
	return ndlist;
}


// this binds a refnode to a circuit node

static void  Bind( ND_NODE * sref_ndptr, ND_NODE * s_ndptr )
{
	sref_ndptr->numofbindings++;				// inc number of bindings

	sref_ndptr->bound = s_ndptr->nodenum;		// mark ref node as bound

	s_ndptr->bound = sref_ndptr->nodenum;		// mark circuit node as bound
}


static void UnBind( ND_NODE * sref_ndptr, ND_NODE * s_ndptr )
{
	sref_ndptr->numofbindings--;	// decrement and check for zero

	if (sref_ndptr->numofbindings == 0) {

		sref_ndptr->bound = 0;	// unbind the variables
		s_ndptr->bound = 0;
	}
}


static void InitListsForTryMatch( REF_NODE * rcptr, TRANS_NODE * tlist,
								ND_NODE * ndlist)
{
```

```
        ND_NODE                         * refptr, * circptr;

        // initialize all transistors in tlist to be unbound

        while (tlist != NULL) {

                tlist->bound = 0;
                tlist = tlist->nextvertex;
        }

                // init all nodes to be unbound

        refptr = rcptr->degreelist;
        circptr = ndlist;

        while (refptr != NULL) {

                refptr->bound = 0;
                refptr->numofbindings = 0;
                refptr = refptr->next;

                circptr->bound = 0;
                circptr->numofbindings = 0;
                circptr = circptr->next;
        }

                // bind vdd and gnd  if they exist

        refptr = GetNDNode( rcptr->degreelist, REFVDD);
        if (refptr != NULL) {                                               // vdd inlist

                Bind(refptr, GetNDNode( ndlist, vdd) );
        }

        refptr = GetNDNode( rcptr->degreelist, REFGND);
        if (refptr != NULL) {                                               // gnd inlist

                Bind(refptr, GetNDNode( ndlist, gnd) );
        }


                // bind all nodenumbers with unique degree

//      refptr = rcptr->degreelist;
//      circptr = ndlist;

//      while (refptr != NULL) {

//              while  ((refptr != NULL) && (refptr->bound) {
//                      refptr = refptr->nextnd;
//              }
//              while (circptr->bound) {
//                      circptr = circptr->nextnd;
//              }

                // we are now pointing at two unbound nodes of equal degree
                // record the degree and advance to the next unbound node

//              degree = refptr->degree;

//              do {
//                      refptr = refptr->nextnd;
//              } while  (refptr->bound);

//              do {
//                      circptr = circptr->nextnd;
//              } while (circptr->bound)

//              if (refptr == NULL) || ((refptr != NULL) && (degree != refptr->degree)) ) {

//                      Bind(refptr, circptr);
```

```
//                      }


//          }
}


// At this point the two circuits have the same descriminating
// characteristics (numoftrans ect.) and trymatch trys to find a mapping
// from the ref circuit to the unknown circuit

static int TryMatch( REF_NODE * rcptr, TRANS_NODE * tlist, ND_NODE * ndlist)
{
        ND_NODE                 * sref_ndptr, * s_ndptr, * dref_ndptr, * d_ndptr,
                                        * gref_ndptr, * g_ndptr;
        TRANS_NODE *    tptr;
        R_TRANS *               refptr;             // the current reference edge that we are
                                                            // trying to match (bind)

        int                                     possiblematch;          // boolean loop control flag
        int         edgenotfound;  // boolean loop control flag
        int                                     nbcy;       // NoBindingConflictsYet flag
        UN_INT                  prevrefnum;


                                                            // initialize lists
        InitListsForTryMatch( rcptr, tlist, ndlist);

        // set refptr to first edge (transistor) in reference trans list

        refptr = rcptr->reftranslist;

        possiblematch = 1;

        tptr = tlist;

        while ( possiblematch && (refptr != NULL) ) {

                edgenotfound = 1;

                while ( (tptr != NULL) && (edgenotfound) ) {
//printf("looping ");
                        if ( (! (tptr->bound) ) &&          (tptr->ptype == refptr->ptype) ) {

                                // test for nodebinding conlficts
//printf("testing conflicts ");
                                nbcy = 1;                               // set the "NoBindingConflictYet" flag

                                sref_ndptr = GetNDNode( rcptr->degreelist, refptr->source);

                                if (sref_ndptr->bound) {
                                        // ref source is bound check to see if s or d of
                                        // current edge equals btorefs, else set conflict flag

                                        if ( (sref_ndptr->bound == tptr->vertexnum) ||
                                                (sref_ndptr->bound == tptr->sourcenum) ) {
//printf("bound source works ");
                                                sref_ndptr->numofbindings++; // inc # of bindings
                                                s_ndptr = NULL;                         // needed
for unbind
                                        }
                                        else {
//printf("bound source fails ");
                                                nbcy = 0;
                                        }
                                }
                                else {                  // ref source is unbound,  see if s or d of
                                                        // current trans is unbound and of like degree

                                        s_ndptr = GetNDNode( ndlist, tptr->sourcenum);
```

```
                                            if ( (! s_ndptr->bound) &&
                                                    (s_ndptr->degree == sref_ndptr->degree) ) {
//printf("binding source to source ");
                                                    Bind( sref_ndptr, s_ndptr );
                                            }
                                            else {
                                                    s_ndptr = GetNDNode( ndlist, tptr->vertexnum);

                                                    if ( (! s_ndptr->bound) &&
                                                            (s_ndptr->degree == sref_ndptr->degree) ) {
//printf("binding source to drain ");
                                                            Bind( sref_ndptr, s_ndptr);
                                                    }
                                                    else {
            // binding conflict
//printf("binding source fails ");

                                                            nbcy = 0;
                                                    }
                                            }
                                    }
                                    if ( nbcy) {

                                            // if we are here then the ref source has been bound and we
                                            // need to unbind it if a binding conflict arrises with the
                                            // drain
                                            dref_ndptr = GetNDNode( rcptr->degreelist, refptr->drain);

                                            if (dref_ndptr->bound) {
                                                    // ref drain is bound check to see if source or drain of current
                                                    // edge equals btorefd, else set conflict flag

                                                    if ( (dref_ndptr->bound == tptr->vertexnum) ||
                                                            (dref_ndptr->bound == tptr->sourcenum) ) {
//printf("bound drain works ");

            // inc num of bindings!!!
                                                            dref_ndptr->numofbindings++;
                                                            d_ndptr = NULL;                 // needed for unbinding
                                                    }
                                                    else {                                                  // binding
conflict
//printf("bound drain fails ");

                                                            nbcy = 0;
                                                    }
                                            }
                                            else {                  // ref drain is unbound,  see if s or d of cur trans
                                                                            // is unbound and of like degree

                                                    d_ndptr = GetNDNode( ndlist, tptr->sourcenum);

                                                    if ( (! d_ndptr->bound) &&
                                                            (d_ndptr->degree == dref_ndptr->degree) ) {

                                                            Bind( dref_ndptr, d_ndptr);
                                                    }
                                                    else {
                                                            d_ndptr = GetNDNode( ndlist, tptr->vertexnum);

                                                            if ( (! d_ndptr->bound) &&
                                                                    (d_ndptr->degree == dref_ndptr->degree) ) {

                                                                    Bind( dref_ndptr, d_ndptr);
                                                            }
                                                            else {
            // binding conflict
                                                                    nbcy = 0;
                                                            }
                                                    }
                                            }
                                            if (! nbcy) {           // there was a binding conflict so we must
                                                                                    // unbind the bound source
```

```
UnBind( sref_ndptr, s_ndptr);
}

else {
        // now both source and drain are bound if the gate has a
        // binding conflict we must unbind both the source and
        // the drain

        gref_ndptr = GetNDNode( rcptr->degreelist, refptr->gate);

        if (gref_ndptr->bound) {
                // ref gate is bound so check to see if gate of circuit
                // edge is bound to gate of reference edge

                if (gref_ndptr->bound == tptr->gatenum) {

                        gref_ndptr->numofbindings++;
//printf("!");
                }
                else {
        // binding conflict
                        nbcy = 0;
                }
        }
        else {                  // ref gate is unbound, see if gate of
                                        // cur trans is unbound and of like
degree

                g_ndptr = GetNDNode( ndlist, tptr->gatenum);

                if ( (! g_ndptr->bound) &&
                        (g_ndptr->degree == gref_ndptr->degree) ) {
//printf("!");
                        Bind( gref_ndptr, g_ndptr);
                }
                else {
        // binding conflict
                        nbcy = 0;
                }
        }
        if (! nbcy) {           // there was a binding conflict so we must
                                                // unbind the bound
source and drain

                UnBind( sref_ndptr, s_ndptr);
                UnBind( dref_ndptr, d_ndptr);
        }
}
}
if (! nbcy) {           // there were bind conflicts advance tptr
        tptr = tptr->nextvertex;
}
else {  // a good edge has been found and tptr points to it

        edgenotfound = 0;
}
}
else {
        tptr = tptr->nextvertex;
}

} // while edge not found loop end

if (! edgenotfound) {                   // an edge has been found
//printf("edge found advancing ");
        tptr->bound = 1;                        // mark circuit trans as bound

        // set ref trans edge pointer to point at newly bound edge

        refptr->edgeptr = tptr;
```

```
				refptr = refptr->nextt;				// advance the refptr

								// and restart search at the beginning of the list
				tptr = tlist;
			}
			else {	// the search failed we need to back up the refptr,
						// unbind that ref trans and and node associated with it
						// then set tptr equal to the next trans in the circuit and
						// do a goto
						// if refptr can't be backed up anymore then trymatch has
						// failed

				if ( refptr->tnum == 1 ) {			// refptr is pointing at the first trans in the list
//printf("big time failure");
						possiblematch = 0;				// trymatch has failed
				}


				else {
//printf("backtracking ");
						prevrefnum = refptr->tnum - 1;	// calc the number of previous trans

						refptr = rcptr->reftranslist;
						while (refptr->tnum != prevrefnum) {		// backup

							refptr = refptr->nextt;
						}

						tptr = refptr->edgeptr;
						tptr->bound = 0;							// unbind
trans


			// unbind s d & g

						UnBind( GetNDNode( rcptr->degreelist, refptr->source) ,
								GetNDNode( ndlist, tptr->sourcenum) );

						UnBind( GetNDNode( rcptr->degreelist, refptr->drain) ,
								GetNDNode( ndlist, tptr->vertexnum) );

						UnBind( GetNDNode( rcptr->degreelist, refptr->gate) ,
								GetNDNode( ndlist, tptr->gatenum) );

						tptr = tptr->nextvertex ;				// set tptr to next trans
				}
			}
		}
//printf("\n");
		return possiblematch;
}

static void FreeNodelist(ND_NODE * ndlist)
{
		ND_NODE * ndptr;

		while (ndlist != NULL) {

			ndptr = ndlist;
			ndlist = ndlist->next;

			free(ndptr);						// return memory
		}
}


static int NandCount = 0;

static void  RecogObject( BLOCK * loadptr, BLOCK * driverptr)
{
		TRANS_NODE *	tptr;
```

```
	ND_NODE *			nodedegreelist;

	UN_INT				numoftrans, vdddegree, compsneeded;

	REF_NODE *			rcptr;
	UN_INT				match;

// switch these lists to change the order in the reconizer for a test!!!!!!!!

			// create node degree list and get number of trans and vdddegree

	tptr = MergeTransList( driverptr->translist, loadptr->translist );

	nodedegreelist = BuildDegreeList( tptr, &numoftrans);
//printf("circuit numoftrans = %i ",numoftrans);

	nodedegreelist = SortDegreeList( nodedegreelist);

	vdddegree = GetVddDegree( nodedegreelist );
//printf("highest degree = %i ",nodedegreelist->degree);

	rcptr = FindPossibleCircuit(LOADDRIVER, numoftrans, vdddegree, compsneeded,
								nodedegreelist);
	match = 0;
	while ((! match) && (rcptr != NULL)) {
//printf("trying match ");
		match = TryMatch(rcptr, tptr, nodedegreelist);

		if (! match) {

			rcptr = FindNextPossibleCircuit( rcptr );

//if (rcptr != NULL) { printf(" Another possibility "); }
		}
		else {					// assert the load driver block
NandCount++;
			CreateObject (rcptr->objectname, rcptr->objnum,
					loadptr->inputlist, loadptr->outputlist, tptr);

//printf(" circuit recognized!! ");
		}
	}
	if (! match) {					// something new has been found
//printf(" match fail ");
			// create new ref ciruit and keep trak of it as a new circuit
	}
	// return memory both blocks and the node degree list

//	free(loadptr);				// these things slow things down considerably
//	free(driverptr);
FreeNDNodes( nodedegreelist );

}


extern void ReadRefCircuits(void)
{
	R_TRANS *			rtptr;
	ND_NODE *			nodedegreelist;
	UN_INT				numoftrans, vdddegree, compsneeded;
	REF_NODE *			rcptr;

	char		rname[12];
	char		buffer[80];
	UN_INT		indesc, objnum, rsource, rdrain, rgate, ptype;
	FILE		*fileptr;
	char		libfile[12];
										// read
reference circuits
	printf("\nPlease enter the library filename that you wish to use.\n\n");
	scanf( "%s", libfile);
```

```
        fileptr = fopen( libfile, "r");

        indesc = 0;
        while (      (fgets(buffer, 80, fileptr)) != NULL) {

        //           printf("%s", buffer);

                     if ((! indesc) && ((sscanf(buffer,"%s %d", &rname[0], &objnum)) == 2)) {

//                                printf( "%s %d \n", rname, objnum);

                                  indesc = 1;

                                  rtptr = NULL;                              // Null the ref trans list

                     }
                     else if ((indesc) && ((sscanf(buffer,"trans %d %d %d %d", &rsource, &rdrain, &rgate, &ptype) ) == 4)) {

//                                printf("trans %d %d %d %d \n", rsource, rdrain, rgate, ptype);

                                  rtptr = InsertRefTrans( rtptr, rsource, rdrain, rgate, ptype);

                     }
                     else {
                                  indesc = 0;

//                                getchar();

                                  // create node degree list and get number of trans and vdddegree

                                  nodedegreelist = BuildRefDegreeList( rtptr, &numoftrans);

                                  nodedegreelist = SortDegreeList( nodedegreelist);

                                  vdddegree = GetRefVddDegree( nodedegreelist );

                                  CreateRefObject (rname, objnum, LOADDRIVER, numoftrans, vdddegree,
                                                      compsneeded, nodedegreelist, rtptr);
                     }
        }
        fclose( fileptr);
}                                                                                        // end
refcircuit list creation


// Block list iterators with fast removal capability

static BPL_NODE * * BackBlockPtr;

// may nexer be called with a NULL pointer!!!!!!!!!!!

static BPL_NODE * NextBlock(BPL_NODE * curptr)
{
        BackBlockPtr = &(curptr->nextblk);

        return *BackBlockPtr;
}

// deletes the current node from the block pointer list
// and returns a pointer to the node following the current node

static BPL_NODE * DeleteBPNode( BPL_NODE * curptr )
{
        BPL_NODE *          p;

        p = *BackBlockPtr;

        *BackBlockPtr = (**BackBlockPtr).nextblk;
```

```
//          free(p);
            return *BackBlockPtr;
}



extern void LDBlocksRecognize(void)
{
            int                             i;
            int         loadfound;                          // load blocktype found
            int                             driverfound;                        // driver blocktype found

            BPL_NODE *          blkptr;
            BLOCK * loadptr;
            BLOCK * driverptr;

            for (i = NodeCount ; i >= 1 ; i--) {
//printf(":");
//printf("\n nodecountloop nodetype = |%i|",CircuitGraph[i].nodetype);
                        if (CircuitGraph[i].nodetype == PULLUP) {

                                    // node number is a pullup node look for load driver pair

//printf("Pullup ");
                                    BackBlockPtr = &(CircuitGraph[i].blockptr);             // set back ptr
                                    blkptr = CircuitGraph[i].blockptr;

                                    loadfound = 0;
                                    driverfound = 0;
//printf("pointers set up ");
                                    while ( (blkptr != NULL) && ((! loadfound) || (! driverfound)) ) {
//printf("searching for loaddriver ");
                                                if (blkptr->block->blocktype == LOAD) {
//printf(" foundload");
                                                            loadfound = 1;
                                                            loadptr = blkptr->block;                      // save pointer to block
                                                            blkptr = DeleteBPNode(blkptr);   // remove it from list
                                                }
                                                else if (blkptr->block->blocktype == DRIVER) {
//printf(" founddriver");
                                                            driverfound = 1;
                                                            driverptr = blkptr->block;
                                                            blkptr = DeleteBPNode(blkptr);
                                                }
                                                else {
//printf(" callingnext");
                                                            blkptr = NextBlock(blkptr);
                                                }
                                    }

                                    if (loadfound && driverfound) {
//printf("loaddriver at node %i \n",i);
                                                RecogObject( loadptr, driverptr);
//if (CircuitGraph[i].blockptr == NULL) printf("null block!");
                                    }

//printf("\n");
                        }
                        else {
//printf(" not pullup\n");
                        }
            }
}



static int BruteTryMatch( REF_NODE * rcptr, TRANS_NODE * tlist, ND_NODE * ndlist)
{
            ND_NODE                         * sref_ndptr, * s_ndptr, * dref_ndptr, * d_ndptr,
                                                        * gref_ndptr, * g_ndptr;
            TRANS_NODE *        tptr;
```

```
	R_TRANS *			refptr;			// the current reference edge that we are
										// trying to match (bind)

	int				possiblematch;		// boolean loop control flag
	int	edgenotfound;	// boolean loop control flag
	int				nbcy;			// NoBindingConflictsYet flag
	UN_INT			prevrefnum;


										// initialize lists
	InitListsForTryMatch( rcptr, tlist, ndlist);

	// set refptr to first edge (transistor) in reference trans list

	refptr = rcptr->reftranslist;

	possiblematch = 1;

	tptr = tlist;

	while ( possiblematch && (refptr != NULL) ) {

		edgenotfound = 1;

		while ( (tptr != NULL) && (edgenotfound) ) {
//printf("looping ");
			if ( (! (tptr->bound) ) &&	(tptr->ptype == refptr->ptype) ) {

				// test for nodebinding conlficts
//printf("testing conflicts ");
				nbcy = 1;					// set the "NoBindingConflictYet" flag

				sref_ndptr = GetNDNode( rcptr->degreelist, refptr->source);

				if (sref_ndptr->bound) {
					// ref source is bound check to see if s or d of
					// current edge equals btorefs, else set conflict flag

					if ( (sref_ndptr->bound == tptr->vertexnum) ||
						(sref_ndptr->bound == tptr->sourcenum) ) {
//printf("bound source works ");
						sref_ndptr->numofbindings++; // inc # of bindings
						s_ndptr = NULL;					// needed
for unbind
					}
					else {
//printf("bound source fails ");
						nbcy = 0;
					}
				}
				else {			// ref source is unbound, see if s or d of
								// current trans is unbound and of like degree

					s_ndptr = GetNDNode( ndlist, tptr->sourcenum);

//					if ( (! s_ndptr->bound) &&
//						(s_ndptr->degree == sref_ndptr->degree) ) {
if (! s_ndptr->bound) {

//printf("binding source to source ");
						Bind( sref_ndptr, s_ndptr );
					}
					else {
						s_ndptr = GetNDNode( ndlist, tptr->vertexnum);

//						if ( (! s_ndptr->bound) &&
//							(s_ndptr->degree == sref_ndptr->degree) ) {
if (! s_ndptr->bound) {

//printf("binding source to drain ");
							Bind( sref_ndptr, s_ndptr);
```

```
                                                }
                                                else {
            // binding conflict
//printf("binding source fails ");
                                                            nbcy = 0;
                                                }
                                    }
                        }
                        if ( nbcy) {

                                    // if we are here then the ref source has been bound and we
                                    // need to unbind it if a binding conflict arrises with the
                                    // drain
                                    dref_ndptr = GetNDNode( rcptr->degreelist, refptr->drain);

                                    if (dref_ndptr->bound) {
                                                // ref drain is bound check to see if source or drain of current
                                                // edge equals btorefd, else set conflict flag

                                                if ( (dref_ndptr->bound == tptr->vertexnum) ||
                                                            (dref_ndptr->bound == tptr->sourcenum) ) {
//printf("bound drain works ");

            // inc num of bindings!!!
                                                            dref_ndptr->numofbindings++;
                                                            d_ndptr = NULL;                  // needed for unbinding
                                                }
                                                else {                                              // binding
conflict
//printf("bound drain fails ");

                                                            nbcy = 0;
                                                }
                                    }
                                    else {                  // ref drain is unbound, see if s or d of cur trans
                                                                        // is unbound and of like degree

                                                d_ndptr = GetNDNode( ndlist, tptr->sourcenum);

//                                              if ( (! d_ndptr->bound) &&
//                                                          (d_ndptr->degree == dref_ndptr->degree) ) {
if (! d_ndptr->bound) {

                                                            Bind( dref_ndptr, d_ndptr);
                                                }
                                                else {
                                                            d_ndptr = GetNDNode( ndlist, tptr->vertexnum);

//                                                          if ( (! d_ndptr->bound) &&
//                                                                      (d_ndptr->degree == dref_ndptr->degree) ) {
if (! d_ndptr->bound) {

                                                                        Bind( dref_ndptr, d_ndptr);
                                                            }
                                                            else {
            // binding conflict
                                                                        nbcy = 0;
                                                            }
                                                }
                                    }
                                    if (! nbcy) {           // there was a binding conflict so we must
                                                                                    // unbind the bound source

                                                UnBind( sref_ndptr, s_ndptr);
                                    }

                                    else {
                                                // now both source and drain are bound if the gate has a
                                                // binding conflict we must unbind both the source and
                                                // the drain

                                                gref_ndptr = GetNDNode( rcptr->degreelist, refptr->gate);
```

```
                                        if (gref_ndptr->bound) {
                                                // ref gate is bound so check to see if gate of circuit
                                                // edge is bound to gate of reference edge

                                                if (gref_ndptr->bound == tptr->gatenum) {

                                                        gref_ndptr->numofbindings++;
//printf("!");
                                                }
                                                else {
            // binding conflict
                                                        nbcy = 0;
                                                }
                                        }
                                        else {                  // ref gate is unbound,  see if gate of
                                                                        // cur trans is unbound and of like
degree

                                                g_ndptr = GetNDNode( ndlist, tptr->gatenum);

//                                              if ( (! g_ndptr->bound) &&
//                                                      (g_ndptr->degree == gref_ndptr->degree) ) {
if (! g_ndptr->bound) {


//printf("!");
                                                        Bind( gref_ndptr, g_ndptr);
                                                }
                                                else {
            // binding conflict
                                                        nbcy = 0;
                                                }
                                        }
                                        if (! nbcy) {           // there was a binding conflict so we must
                                                                                // unbind the bound
source and drain
                                                UnBind( sref_ndptr, s_ndptr);
                                                UnBind( dref_ndptr, d_ndptr);
                                        }
                                }
                        }
                        if (! nbcy) {           // there were bind conflicts advance tptr
                                tptr = tptr->nextvertex;
                        }
                        else {  // a good edge has been found and tptr points to it

                                edgenotfound = 0;
                        }
                }
                else {
                        tptr = tptr->nextvertex;
                }

        }  // while edge not found loop end

        if (! edgenotfound) {                   // an edge has been found
//printf("edge found advancing ");
                tptr->bound = 1;                        // mark circuit trans as bound

                // set ref trans edge pointer to point at newly bound edge

                refptr->edgeptr = tptr;

                refptr = refptr->nextt;                 // advance the refptr

                                        // and restart search at the beginning of the list
                tptr = tlist;
        }
        else {  // the search failed we need to back up the refptr,
                        // unbind that ref trans and and node associated with it
```

```
                                        // then set tptr equal to the next trans in the circuit and
                                        // do a goto
                                        // if refptr can't be backed up anymore then trymatch has
                                        // failed

                                if ( refptr->tnum == 1 ) {              // refptr is pointing at the first trans in the list
//printf("big time failure");
                                        possiblematch = 0;                      // trymatch has failed
                                }


                                else {
//printf("backtracking ");
                                        prevrefnum = refptr->tnum - 1;   // calc the number of previous trans

                                        refptr = rcptr->reftranslist;
                                        while (refptr->tnum != prevrefnum) {            // backup

                                                refptr = refptr->nextt;
                                        }

                                        tptr = refptr->edgeptr;
                                        tptr->bound = 0;                                                                // unbind
trans


                        // unbind s d & g

                                        UnBind( GetNDNode( rcptr->degreelist, refptr->source) ,
                                                        GetNDNode( ndlist, tptr->sourcenum) );

                                        UnBind( GetNDNode( rcptr->degreelist, refptr->drain) ,
                                                        GetNDNode( ndlist, tptr->vertexnum) );

                                        UnBind( GetNDNode( rcptr->degreelist, refptr->gate) ,
                                                        GetNDNode( ndlist, tptr->gatenum) );

                                        tptr = tptr->nextvertex ;                               // set tptr to next trans
                                }
                        }
                }
//printf("\n");
                return possiblematch;
}


// brute force searcher for load driver gates

static void  BruteRecogObject( void )
{
        TRANS_NODE *     tptr;
        ND_NODE *                nodedegreelist;

        UN_INT                   i, numoftrans, vdddegree, compsneeded;

        REF_NODE *               rcptr;


        rcptr = RefObjectList;
        while (rcptr != NULL) {

                for (i = NodeCount ; i >= 1 ; i--) {

                        if ((CircuitGraph[i].blockptr != NULL) &&
                                (CircuitGraph[i].blockptr->block->blocktype == LOADDRIVER) ) {

                                tptr = CircuitGraph[i].blockptr->block->translist;

                                // create node degree list and get number of trans and vdddegree
```

```
                    nodedegreelist = BuildDegreeList( tptr, &numoftrans);

                    nodedegreelist = SortDegreeList( nodedegreelist);

                    vdddegree = GetVddDegree( nodedegreelist );


                    if ( BruteTryMatch(rcptr, tptr, nodedegreelist) ) {

                    if (numoftrans == rcptr->numoftrans) {
                                        // assert the load driver block
NandCount++;
                         CreateObject (rcptr->objectname, rcptr->objnum,
                              CircuitGraph[i].blockptr->block->inputlist,
                              CircuitGraph[i].blockptr->block->outputlist, tptr);


                         CircuitGraph[i].blockptr = NULL;

//printf(" circuit recognized!! ");
                    }
                    }

                    FreeNDNodes( nodedegreelist );

               } // end if

          } // end for

          rcptr = rcptr->nextref;

     } // end while
}



extern void BruteLDBlocksRecognize(void)
{
     int                 i;
     int       loadfound;               // load blocktype found
     int                 driverfound;             // driver blocktype found

     BPL_NODE *     blkptr;
     BLOCK * loadptr;
     BLOCK * driverptr;

     for (i = NodeCount ; i >= 1 ; i--) {

          if (CircuitGraph[i].nodetype == PULLUP) {

               BackBlockPtr = &(CircuitGraph[i].blockptr);          // set back ptr
               blkptr = CircuitGraph[i].blockptr;

               loadfound = 0;
               driverfound = 0;
               while ( (blkptr != NULL) && ((! loadfound) || (! driverfound)) ) {
                    if (blkptr->block->blocktype == LOAD) {
                         loadfound = 1;
                         loadptr = blkptr->block;              // save pointer to block
                         blkptr = DeleteBPNode(blkptr);   // remove it from list
                    }
                    else if (blkptr->block->blocktype == DRIVER) {
                         driverfound = 1;
                         driverptr = blkptr->block;
                         blkptr = DeleteBPNode(blkptr);
                    }
                    else {
                         blkptr = NextBlock(blkptr);
                    }
               }
               if (loadfound && driverfound) {
```

```
						// set up loaddriver block for brute serching

					loadptr->translist = MergeTransList( driverptr->translist, loadptr->translist );
					loadptr->blocktype = LOADDRIVER;

					CircuitGraph[i].blockptr = InsertBp_Node( CircuitGraph[i].blockptr, loadptr);
				}
			}
		}
		BruteRecogObject();
}
```