VERIFICATION OF CIRCUIT INTERCONNECTIVITY

by

R. L. Spickelmeier

Memorandum No. UCB/ERL M83/66

21 October 1983

COVER PAGE

VERIFICATION OF CIRCUIT INTERCONNECTIVITY

by

Rick L. Spickelmier

Memorandum No. UCB/ERL M83/66

21 October 1983

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Verification of Circuit Interconnectivity

by

Rick L. Spickelmier

October 1983

Electronics Research Laboratory

Cory Hall

University of California

Berkeley, California 94720

ACKNOWLEDGEMENTS

The author wishes to express his sincere appreciation to Prof. A.R. Newton for his advice, guidance and support in this project. The author would also like to thank Prof. D. O. Pederson for his advice and guidance in the preparation of this report. He also gratefully acknowledges the discusions with, and the support of, John Crawford, Dave Poulsen, Greg Sanguinetti, Ian Getreu and the Tektronix CAD and IC Design Groups. He especially would like to thank the Tektronix CAD and IC Design Groups as for acting as test sites. The author would also like to thank Peter Moore, Ken Keller and the Electrical Engineering CAD group at the University of California, Berkeley for discussions and their help in coding.

The author is especially grateful to Tektronix Incorporated and the Wheeler Fellowship for their funding of this project, and the Digital Equipment Company, Engineering Systems Group, for major contributions to the computing environment in which this research was carried out.

TABLE OF CONTENTS

CHAPTER 1

Introduction

As integrated circuits increase in size and complexity, conventional verification techniques, such as hand checking these circuits for errors, can take hours and sometimes days with no final guarantee of an error-free circuit. Circuits that have been visually checked by designers and layout specialists have been found to have missing contacts when entering the mask shop. Errors like this one are costly in terms of money and design time. Computer programs to verify the correctness of these circuits can reduce the checking time to minutes or even seconds and guarantee that the circuit is free of connectivity errors. Such programs compare the schematic netlist with a netlist extracted from the layout itself and indicate any discrepancies. Connectivity verification programs should find the most use in the following:

(1) Checking Circuits Laid Out by Hand

(2) Checking Circuit Design Aids

Netlist comparison programs can be used to verify that new or modified circuit design aids (extractors, routers) produce the proper result by comparing the output of the program with the *correct* result.

(3) Editing

After a layout/schematic/simulation file has been edited, a netlist comparison program can be used to compare the original file versus the edited file to see if the changes that the user intended to make were indeed what changes that were made.

(4) Parasitics

After an Integrated Circuit has been laid out, the circuit is sometimes extracted and re-simulated with detailed parasitics included. The extracted circuit contains circuit element and node names that have no meaning for the designer, so he ends up spending time trying to find name equivalences between the extracted circuit and his original simulation file. Netlist comparison programs can find those equivalences for the designer and post-processors can place the detailed parasitics in a simulation file that uses the designers original names.

1.1. The Problem

The problem addressed in this report is the comparison of two representations of a circuit (*i.e.* simulation file and layout) (1) to verify that the circuits are the same, or (2) to determine where the circuits differ. In doing the comparison, the program should take into account that certain pins on elements can be permuted with no change in the electrical or logical function of the network.

Various programs to meet the above requirements have emerged over the past few years, varying from in-house programs for a specific task to commercial programs for the more general problem (PDS, NCA[1]). Many in-house programs tend to solve a specific problem facing the company, such as comparing two MOSFET net lists, where the permutability (electrical equivalence) of the source and drain nodes is hard-wired into the program. Commercial programs, while sometimes being more general, are usually part

[1] PDS (Phoenix Data Systems) and NCA are the major commercial sources of netlist verifiers.

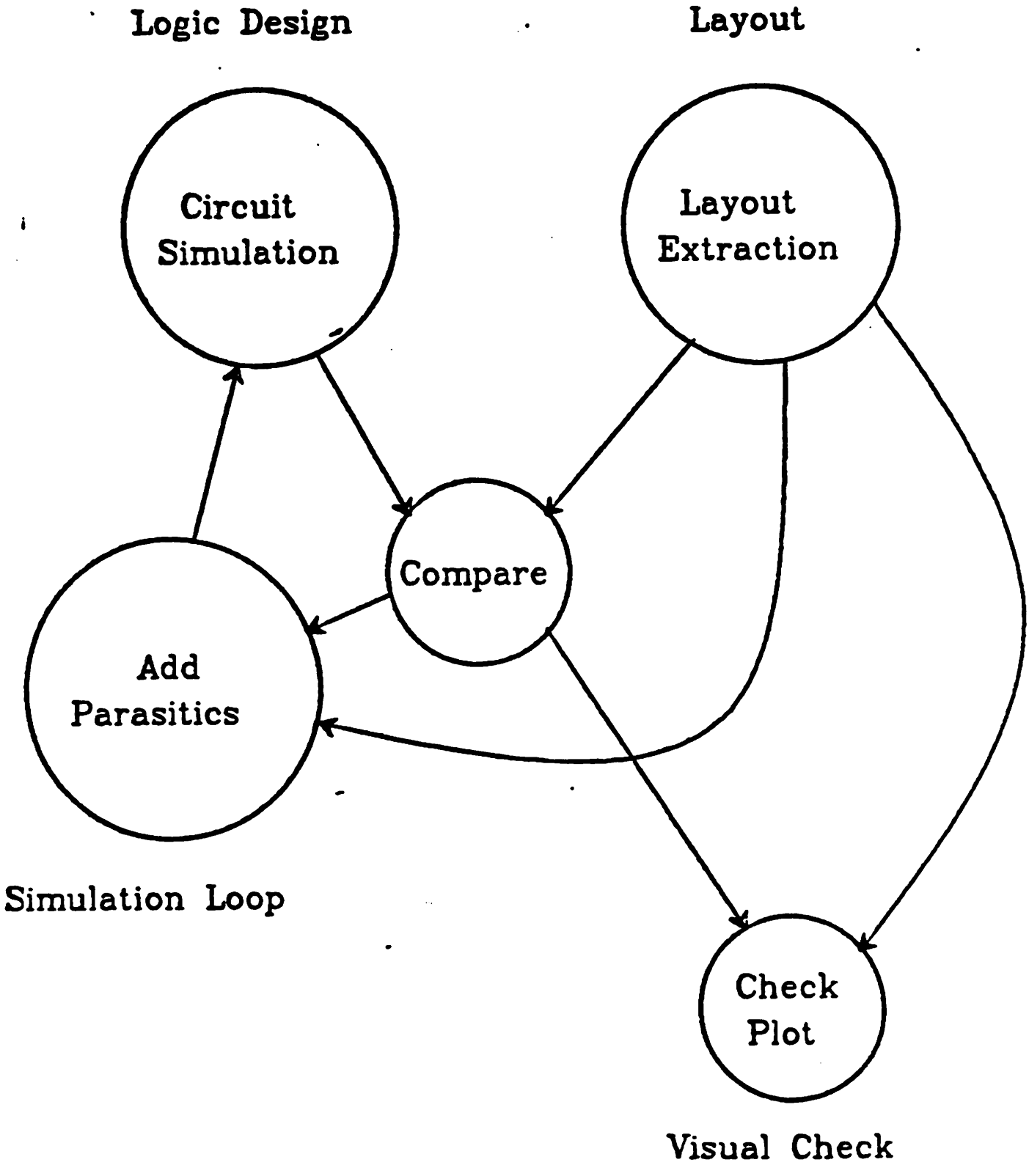


Figure 1.1. Typical Design System.

of a system of programs (input processors, DRC's, ERC's, database manager) that can not be broken up to be used separately, *i.e.* can't use the connectivity verifier without the input processor and database manager.

The program *Wombat*, described in this report, is designed to meet most of the basic requirements mentioned above. *Wombat* is written in the programming language C [Ker78a] and runs on a VAX under Berkeley 4.1 VAX/Virtual UNIX[2] and DEC VMS 3.0[3]. *Wombat* compares two netlists. One is usually based on the simulator input and the other one is often extracted from the layout. However, this is not necessarily the case and *Wombat* makes no assumptions about the number of pins or the technology of the circuit elements in the schematics. One can specify how the pins of individual elements permute and an initial correspondence between circuit elements and nets. *Wombat* generates a list of corresponding elements and nets and notes any differences.

The remaining part of this Chapter presents the various implementations of netlist verification and the problems encountered in netlist verification. The basic program flow and the algorithms implemented in *Wombat* are described in Chapter 2 and the data structures used in *Wombat* are presented in Chapter 3. Chapter 4 covers the various files used and generated by *Wombat*. The results of some example runs are presented in Chapter 5 and finally Chapter 6 summarizes *Wombat* and covers extensions/problems/solutions.

---

[2]UNIX is a trademark of Bell Labs.

[3]VAX and VMS are trademarks of the Digital Equipment Corporation.

1.2. APPRAISE

The Bell Labs APPRAISE program[All77a] is based on an iterative 2-level comparison hierarchy. On the first level, elements are compared and *bound*; on the next level, nets connected to *bound* elements are *bound*. The interesting features of this approach are its use of *signatures* and *thresholds* and its ability to dynamically change the items used in making up the *signature*.

1.2.1. Basic Flow of APPRAISE

repeat until *nothing left to bind*

repeat for each *element in the first circuit*

repeat for each *element in the second circuit that is of the same type as the first element*

calculate the distance between the two elements

end repeat *of second circuit elements*

decide whether to bind first circuit element to any of the second circuit elements

if *elements were bound, bind all nonpermutable pins.*

end repeat *of first circuit elements*

repeat until *no pins left to bind*

repeat for each *pin in the first circuit*

repeat for each *appropriate pin in the second circuit*

calculate the distance between the two pins

end repeat *of second circuit pins*

decide whether to bind first circuit pin to any of the second circuit pins

end repeat *of first circuit pins*

end repeat

end repeat

1.2.2. Calculating Distances

The element comparison is considered a pattern recognition problem. The comparison is based on the *distance* between *signatures* of elements. A *signature* is a set of *features* for an element. A *feature* is a *derived property* of the element (*e.g.* fanout). A *discriminant function* is used to calculate a *distance* between the same *feature* of two elements. The *distances* are then weighted and summed, and the result is called the *distance* between the *signatures* of the two elements. The *distances* between the *signature* of a single element in one circuit and the *signatures* of all the elements with the same *intrinsic properties* (*e.g.* cell type - NAND, NOR) on the other circuit are calculated and the two smallest *distances* are saved.

1.2.3. Deciding Whether to Bind

Deciding when to bind or when not to bind is one of the major features of this program. By introducing a *threshold*, binding is impeded until enough information is gained about an element and an amount of robustness is added (indifference to errors). The two smallest *signature distances* are then compared with a *threshold*, if both *distances* are larger than the *threshold*, no *binding* takes place. If one *distance* is smaller than the *threshold* and the other *distance* is larger, the two elements that were used to calculate the small *distance* are *bound*. If both *distances* are smaller than the *threshold*, and are *sufficiently separated*, the two elements with the smaller *distance* are *bound*, if both are not *sufficiently separated*, no *binding* takes place. *Sufficiently separated* means that the two *distances* differ by some fraction of the *threshold*; the fraction currently used is $\frac{1}{2}$.

1.3. LIVES

The NTT LIVES (Logic Interconnection VErification System) program[Miy81a] treats the two circuits to be compared as graphs and solves the problem based on graph isomorphism. Elements are considered vertices in the graph and interconnection wires are directed edges, with the direction being the direction of signal flow. The general method of solving the graph isomorphism problem for graphs with N vertices takes $N!$ operations. For circuits of 1000 elements, this is 4.0×10^{2567} operations and for circuits of only 50 elements, this is still 3.0×10^{64}. Therefore, to solve the graph isomorphism problem in a reasonable time, the problem must be broken down into smaller ones. This is done by partitioning the graph.

1.3.1. Partitioning

The LIVES partitioning algorithm is based on the three following methods:

(1) Partitioning based on distance

The program characterizes each vertex by the number of vertices N_1, N_2, N_3, ..., N_m that are distance 1,2,3,....,m from it. Distance is defined as the number of edges along the shortest path between two vertices.

(2) Partitioning based on distance from the terminals

Each vertex is characterized by its distance from each terminal. This method is used because certain parallel circuits, such as n-bit adders and registers, have many vertices that have the same characteristics by method (1).

(3) Partitioning based on labels

The two above partitioning methods depended only on topological information and it is useful to use the label information (type of gate - NAND, NOR) to further partition the circuit.

The above three methods when combined produce partitions that allow the graph isomorphism problem to be solved in a reasonable amount of computation time. Note that the items used in the partitioning algorithms are nothing more than a particular form of signature.

However, some assumptions are made for the comparison: (1) only a small set of elements are allowed (*inverter*, *nor*, *nand*, *and-or-inverter*, *nor latch*, *nand latch*, *transmission gate* and *wired-or*); (2) unidirectional logic flow; (3) total permutability of the inputs and total permutability of the outputs and (4) knowledge of where the terminals are. The program is however, a good solution for the specific problem.

1.3.2. Error Detection

If the verification result indicates that there is an error(s) in one of the circuits, the program tries to pin-point the error(s). First, the program divides the circuit graph into subgraphs; all vertices and edges that belong to all routes from a particular input to a particular output are considered a terminal-pair subgraph. Second, the equivalent terminal-pair subgraphs from the two circuits are checked for isomorphism. All equivalent terminal-pair subgraphs found to be non-isomorphic, are determined to be in error. For each subgraph in error, all vertices that also belong to correct subgraphs are removed. The remaining vertices and edges in the error sub-

graphs form the region of the circuit in error. This method works well except for clusters of errors, where most vertices belong to multiple error subgraphs and few belong to correct subgraphs. Most error detection methods fail when errors cluster (spurious errors can occur). In practice however, connectivity errors tend not to cluster in Integrated Circuit layouts.

1.4. Simulation

Another method of circuit verification is to simulate both circuits with a circuit/logic simulator and compare the outputs of the two simulators. This method is only as good as the input to the simulator (test vectors) are complete.

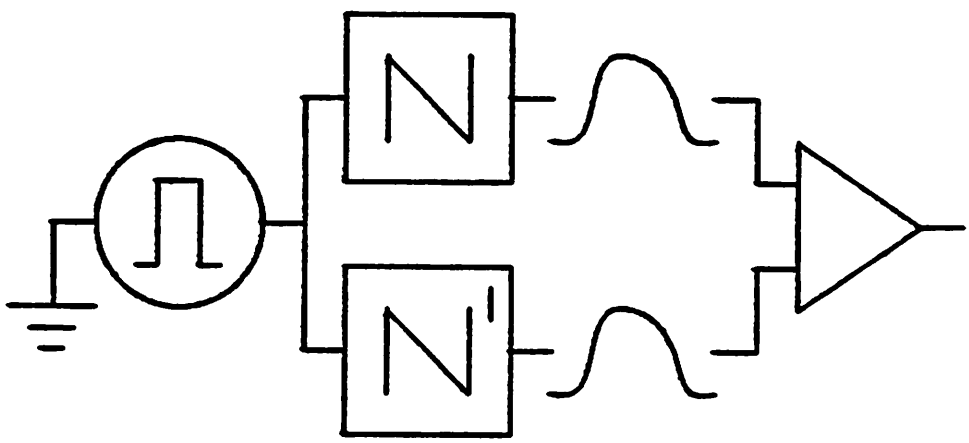


Figure 1.2 Verification by simulation

1.5. Problems in Netlist Verification

Both the LIVES and the APPRAISE programs suffer from common problems found in netlist verification. These problems are the treatment of hierarchy and the treatment of parallel paths.

1.5.1. Hierarchy and Pin Interchangeability

Most netlist verifiers output that the two circuits shown in Figure 1.2 are the same[4] That result is correct; the two inputs to the nand gate (b,c) are permuted in the second circuit.

Circuit 1

```
cella   nand2   a   b   c
cellb   xx      b
cellc   yy      c
```

Circuit 2

```
cella   nand2   a   c   b
cellb   xx      b
cellc   yy      c
```

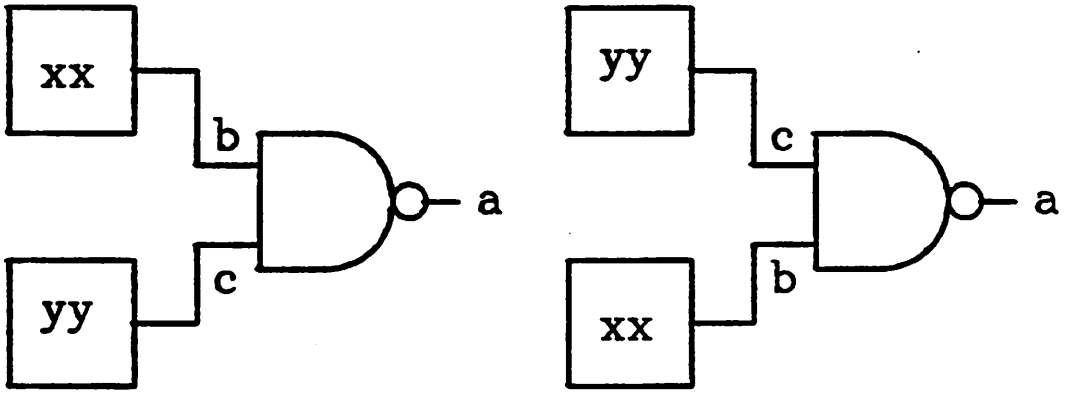


Figure 1.3. Non-expanded NAND gates

[4] Throughout this report, *Wombat* input format will be used to describe example circuits. The format is described in detail in Chapter 4 and Appendix A. In general, a circuit element is described by a statement of the form:

InstanceName CellType NetName NetName ...

Comments are prebeded by a semicolon (;).

However, if these two circuits are expanded to the transistor level, the two circuits in Figure 1.3 are produced. These circuits are logically and electrically the same, but different from a connectivity verifiers stand-point.

Circuit 1

```
; expanded nand gate
x001   dep   vdd   a   a
x002   enh   a     b   1
x003   enh   1     c   gnd

cellb  xx    b
cellc  yy    c
```

Circuit 2

```
; expanded nand gate
x001   dep   vdd   a   a
x002   enh   a     c   1
x003   enh   1     b   gnd

cellb  xx    b
cellc  yy    c
```

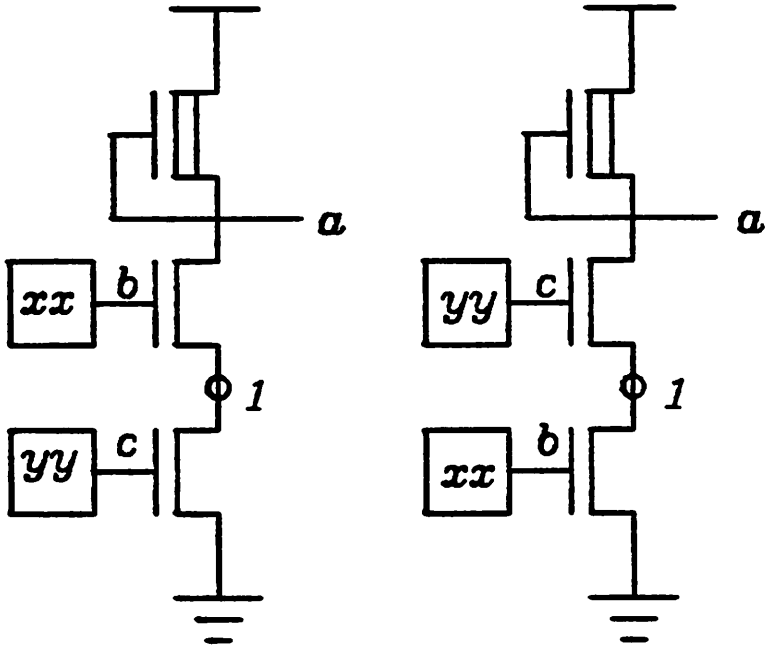


Figure 1.4. Expanded NAND gates

To handle the problem, the connectivity verifier must recognize higher-level cells out of the interconnection of lower-level ones or it must check the circuits in a hierarchical fashion. Checking a circuit in a hierarchical way requires that both of the circuits have the same hierarchy, which is not aways true. Layout designers tend to construct circuits using a different

hierarchy than the hierarchy used by the circuit designers. Layout designers create hierarchies for drawing simplicity and circuit designers occasionally create hierarchies for logical or electrical simulation simplicity. If the hierarchies happen to be the same, hierarchy can be exploited in the netlist verification process.

1.5.2. Parallel Paths

If a circuit contains parallel groups of elements, individual elements in a path always have exactly the same properties as one or more elements in the other parallel paths. None of the algorithms described above treat parallel paths efficiently. The only ways to handle such paths, are (1) to label elements in the parallel paths, or (2) to arbitrarily bind elements in the parallel paths.

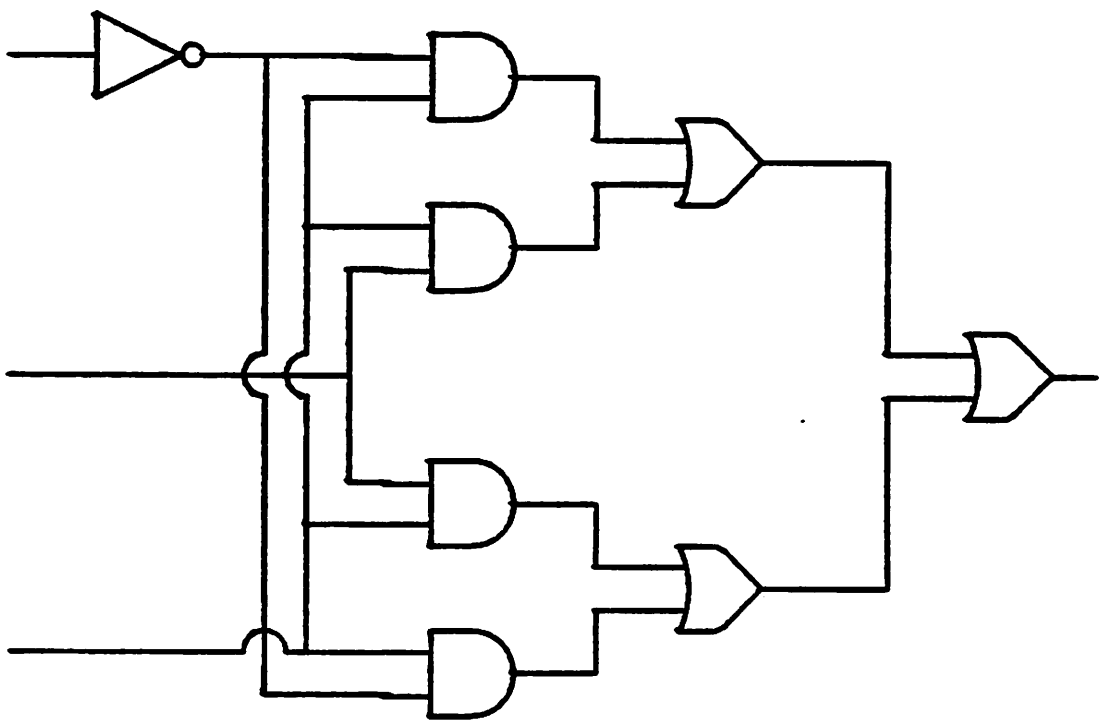


Figure 1.5. Parallel Paths

CHAPTER 2

Algorithm

The basic algorithms in *Wombat* are based on the signature comparison, and element and net binding algorithms in **APPRAISE**.[All77a] The discriminant functions in **APPRAISE** measure distance between signatures, while most of the discriminant functions in *Wombat* are ternary, that is the functions can return one of three values: *less than*, *equal* and *greater than*.

2.1. Basic Program Flow

read-in the data files and setup the data structures.

repeat {

calculate a signature for each element.

bind elements with the same unique signatures.

bind the nets of the elements bound on this iteration.

} **until** *(nothing left to bind* **or** *no more unique signatures can be found)*

The Wombat program is divided into three major sections: *readin*, *compare*, and *bind*

2.2. Readin

Wombat reads in two circuit files, and the following optional files: **Cell-Descriptor File**, **User-Binding File**, **Node-Alias File**. This section of the program takes up a large part of the CPU time in a run; up to 80% of the time for small circuits (see Chapter 6 for ways to reduce this time).

2.2.1. Cell-Descriptor File

The routines *yylex* and *yyparse*, readin and parse the **Cell-Descriptor File** (**CDF**[1]). Each cell definition in the file must have only one **cell** record, zero or more **pin** records, zero or one **ichng** record, and only one **endc** record. A file may have any number of cell definitions. The **cell** record specifies the cell name. The **pin** records specify the pin numbers and names. The pin number specifies the pins ordering in the cell definition used for simulation. The **ichng** record specifies how the pins may be allowed to permute; not all the pins must be specified in this record, only the ones necessary to specify all the allowed permutations. These three record types are used to build the permutability templates for each cell type. The permutability templates themselves are stored in a binary tree of templates.

Each time an element is read in from one of the netlist files, the permutability template tree is searched and a copy of the template for the cells type is placed in the elements data structure. If a permutability template did not exist for the elements type, one would be created that specified that no pins were permutable.

Chapter 6 presents ways to reduce the time spent in parsing large **CDF** files.

2.2.2. Circuit Files

As *Wombat* reads lines from the circuit files it builds the element and net data structures. The routine that does this is named *getelement*. As each element is read, an element structure is dynamically allocated and element information is placed in the structure. Now the routine *lookper* is called

[1]For more information on the **CDF** file format, refer to Appendix B.

to search the permutability template tree to see if a pin-permutability entry exists for the element type. If an entry exists, a local copy is allocated and a pointer is set to its location. If an entry does not exist, the routine *addper* is called and a non-permutable template is created for the element type. It is then placed in the permutability template tree, and as in the first case, a local copy is allocated. For the net data structure, as each net on the element is read in, the routine *addnet* is called to search the net tree to see if the net already exists; if it does, the element count of the net entry is incremented and the current element is placed on the element list for the net entry. If the net does not exist in the tree, a new net entry is created; and if necessary, the tree is rebalanced.

Ways to reduce the readin time for the circuit files will be presented in Chapter 6.

2.2.3. User-Binding File

The routine *user_bind* is used to readin and use user generated element and net binding information. As each line in the **user-binding** file is read in, the two element/net data structures are scanned for an occurrence of the items specified on the line. If both items are found and both are not bound, the items are bound. If either one is not found or if either one is bound to something other than the other one specified in the input line, an error is output. In finding nets in the first circuit, if a net is not found in the net tree, the node-alias list is searched to see if the name is an alias of a net in the net tree. Elements bound by user-binding are marked as *hard-bound*. In the comparison section of the program, if errors are found, sometimes bound elements must be unbound. However, elements marked as *hard-*

bound can not be unbound. This gives the user the option of being always right. This option can be overridden on the command line.

2.2.4. Node-Alias File

The routine *getalias* handles the readin and data structure creation for the **Node-Alias** file. As each node alias line is readin, the alias list is searched to see if the entry already exists, if it does not, an entry is allocated and placed at the end of the list with the information contained on the line. The information from the **node-alias** file is used only during the user-binding phase of program execution; this allows the user to use labels that he is familiar with (some of which may have been deleted from the first circuit input when dummy elements were deleted).

2.3. Comparison and Binding

The combination of comparison and binding is an iterative procedure. By making it iterative, ambiguous information (non-unique signature) is bypassed until more information is built up by net binding to resolve the ambiguities.

2.3.1. Signature Calculation

Signatures are composed of information about an element. Currently *Wombat* uses *fan-in*, *fan-out*, *element type* (NAND, NOR, Enhancement MOSFET, ALU), and *bound nets connected to the element*. *Wombat* does not combine the individual pieces of information about each element into a single entity, but leaves them separate and compares them individually in the signature comparison routine *compkey*. Note that on each iteration, more is

known about some of the nets connected to the elements and therefore a more precise signature for some of the elements can be used on the next iteration.

2.3.2. Signature Comparison

Wombat uses the routine *createtree* to create a tree of elements from the first circuit with the left and right branches determined by the elements signature. A counter is set to the number of elements with that signature. Then the routine *lookup* is called to compare elements in the second circuit against the various nodes in the tree (following the child pointers) until a match is made or a dead-end is reached (child pointer in the direction indicated by the signature comparison is NIL). A match is made when the signatures of the two elements are identical and the element count at that node is 1, i.e. the signature is unique. If the element at that node has already been bound, a connectivity error is reported and the elements are unbound.

The above is repeated using the remaining elements (elements not successfully matched or found to be in error in a previous iteration) until all elements are matched or found in error, or no change occurs in an iteration.

The *dynamically* changing signature comparison functions of APPRAISE were tried in *Wombat*, but made the program much too complex, and for the circuits tried, did not improve the performance of the program.

2.3.3. Latency

Wombat utilizes *latency (activity)* in the iterative compare and bind phase of execution. Since on any given iteration, new information is gained only for elements connected to nets bound on that iteration, only those ele-

ments are used in the next iteration. This results in a savings for loosely connected circuits and circuits that have few elements/nets bound on each iteration. However, for strongly connected circuits and circuits that bind quickly, the execution time has been found to increase due to the overhead of maintaining an *active elements* list. The program tries to use *latency* for as long as it helps and then turn if off. Use of *latency* can also be turned off on the command line.

The circuit shown in Figure 2.1 shows the best improvement that can be achieved by exploiting latency. On the first iteration only two elements look different than the others, the first one (fan-in of 0) and the last one (fan-out of 0). These two elements are bound to their counterparts in the other circuit and the four nets connected to them are also be bound to their counterparts. On the next iteration, only the two elements connected to the elements bound on this iteration have enough information to be bound; therefore, looking at more than just these two would be a waste of time. The first entry in Table 2.1[2] shows this sort of improvement for a 1000 element inverter chain.

Figure 2.1. Inverter Chain

[2] Throughout the report, all times and memory requirements are for a VAX 11/780 computer with a FPA and running the UNIX 4.1BSD operating system, unless otherwise noted.

elements	nets	cpu time (sec)	
		w/Latency	w/o Latency
1000	1001	45	470
1000	1003	59	490
8194	2826	323	440
2336	1374	88	210

Table 2.1. Latency

2.3.4. Error Detection

Errors occur when a signature in one circuit does not occur in the other circuit, or more than one element in one circuit has a signature that is unique in the other circuit.

2.4. Binding

Elements with the same unique signature are *bound* by the program and put on a list. At the end of each iteration, the routine *bind* is called to bind the nets of the elements on this list taking into account the pin-permutability information stored with the element.

2.5. Degenerate Cases

Circuits like the inverter chain shown in Figure 2.1 are considered *degenerate*. For an N element chain $\frac{N}{2}$ iterations are required to bind all elements and nets.

2.6. Memory Management

Since *Wombat* creates an entirely new comparison tree on every iteration, using a memory allocator that maintains a list of free and used memory

blocks would be inefficient, not only in terms of searching the list when allocating, but also in terms of freeing the data at the end of each iteration. Therefore *Wombat* allocates memory as one contiguous chunk and maintains pointers in to this block for the *beginning*, *end* and *current* locations in memory. When a chunk of memory is asked for, the *current* pointer is returned and is then advanced the size of the block asked for. Freeing memory is simply setting the *current* pointer to the *beginning*. If the *current* pointer passes the *end* pointer, the system is asked for more memory, which is assumed to be contiguous to the last block asked for. There are panic checks in the allocator (*malloc*) to guarantee that memory is contiguous.

CHAPTER 3

Data Structures

Each subsection of this Chapter contains a description of each *Wombat* function and the contents of the data structure and a listing of it from the program. It is assumed that the reader of this Chapter is familiar with the C programming language.

3.1. Net

The net data structures are composed of two AVL trees[1], one for each circuit (*first->net*, *second->net*). The trees are created as the circuits are read in. AVL trees were chosen because normally nodes read in from extracted circuits come in numerical order, which leads to highly degenerate trees. The AVL trees significantly reduce the lookup time over normal binary trees. The net data structure was also implemented as a hash table rather than AVL tree, but no significant improvement was noticed. As a net is read in, the net tree is searched to see if the net already exists, if it does the element count (*el_count*) is incremented and the current element is placed on the nets' element list (*el_list*). If the net does not exist in the tree, a new net entry is created, and if necessary, the tree is rebalanced.

[1]AVL trees are height balanced trees with a height mismatch of no more than 1.[Hor76a] In other words, the two subtrees at any node of the tree can not differ in height by more than 1.

```
/* "@(#)net.h9.43/21/83" */

struct nettype {
   STRING n_name;  /* name of the net */
   NET *n_left;    /* pointer to the left subtree */
   NET *n_right;   /* pointer to the right subtree */
   int n_bal;      /* avl tree balance factor */
   int n_el_count; /* number of elements connected to this net */
   ELLIST *n_el_list; /* list of elements connected to this net */
   NET *n_bound;   /* which net is it bound to in the other circuit */
};

struct net_node {
   NET *net;       /* pointer to net */
   NETLIST *next;  /* pointer to the next entry in the list */
};
```

Figure 3.1. Net Data Structure

3.2. Element

The element data structures are composed of two linked lists (*first->element*, *second->element*), one for each circuit representation. As each element is read in, an element structure is dynamically allocated and element information is placed in it. Now the pin-permutability template tree is searched to see if a permutability entry exists for the element type. If one exists, a local copy is allocated and a pointer in the element structure (*ptree*) is set to its location. If one does not exist, a non-permutable template is created for the element type and it is placed in the permutability template tree, and as in the first case, a local copy is allocated.

```
/* "@(#)element.h  9.23/16/83" */

struct eltype {
   STRING e_name;  /* name of the element */
   ELEMENT *e_next;        /* next element in the list */
   int e_fanout;       /* number of output pins tied to the element */
   int e_fanin;        /* number of input pins tied to the element */
   int e_fanbi;        /* number of bi-directional pins ties to the element */
   ELEMENT *e_bound;       /* corresponding element in the other rep */
   STRING e_type;    /* type of the element, i.e. nand3, inv, .... */
   LOGICAL e_hardbound;  /* element is hard bound */
   LOGICAL e_connect_error;        /* is the element connected incorrectly */
   LOGICAL e_active;/* is the element active (latency) */
   NODE *e_ptree;   /* pin permutability tree - netlist */
};

struct el_node {
   ELEMENT *el_element;   /* pointer to element */
   int el_checked;
   ELLIST *el_next;           /* pointer to the next entry in the list */
};
```

Figure 3.2. Element Data Structure

3.3. Element Type String Table

To save space, the element types for both circuits are stored in a string hash table. The table is 203 entries long, this size is based on running many cell names through the hashing function and coming up with as small of a size as possible, while keeping the number of collisions small. It is common for small circuits of 20 cell types to have zero or one collision and large circuits of over 70 cell types to have only two to five collisions (see Table 3.2). The hashing function is:

```
key = 0;
while (*string != ' ') {
    key = (key * 23 + *string++) % HASHTABLESIZE;
}
```

string	key
nand4	1
or	28
bob	89
inv	135
nand	139
and	145
alu4	150
nor	160
xor	172

Table 3.1. Hash Function Examples

number	collisons
11	0
20	1
56	5

Table 3.2. Number of Cell Types and Number of Collisions

3.4. Node-Aliases

The node-alias data structures make up the node-alias list. This list contains equivalence information for various nodes in the first circuit. This equivalence information is usually due to the deleting of *wired-or's*, *wired-and's* and *delay elements* from the first circuit.

```
/* "@(#)alias.h        9.23/16/83" */

/* node alias structure */
struct alias {
    STRING main_name;               /* name on the layout */
    ALIAS *next;                /* pointer to the next alias entry */
    struct othername *aliases;      /* list of aliases of main_name */
};

struct othername {
    STRING name;                /* net alias */
    struct othername *next;/* pointer to the next alias of main_name */
};

extern ALIAS *alias_list;       /* list of all aliases and names */
extern FILE *fpalias;                   /* alias file */
```

Figure 3.3. Node Alias Data Structure

3.5. Key

The key data structure contains the data used to compare elements. The information in this data structure is also called the signature.

```
/* "@(#)key.h          9.12/16/83" */

struct key {
    STRING string;  /* string key (cell type) */
    int fanout;
    int fanin;
    int fanbi;
    NODE *ptree;
};
```

Figure 3.4. Comparison Key Data Structure

3.6. Comparison Tree

The comparison tree data structure is composed of a single binary tree (*tree*). Each node in the tree represents a particular key and the element in the first circuit that has that key is pointed to. If there is more than one

element that matches the key, a count is kept.

```
/* "@(#)tree.h        9.33/24/83" */

struct tnode {
    KEY *key;          /* lookup key */
    TREE *left;        /* left pointer */
    TREE *right;       /* right pointer */
    int count;         /* number of elements containing the key */
    ELEMENT *el;       /* element for key */
};
```

Figure 3.5. Comparison Tree Data Structure

3.7. Permutability

The permutability data structures are composed of a global permutability template tree that contains pin permutability trees for each element type. Each node of the tree is made up of a *celltree* structure. The *celltree* structure contains the name of a cell type, a pointer to its premutability tree, and child pointers to more *celltree* structures. The permutability (*c_ptree*) tree is an m-way tree that is organized in a way that specifies the permutability and non-permutability of the pins that make up the element. Each node in the permutability tree is either a permutable (P), non-permutable (N) or leaf (L) node. Permutable and Non-Permutable nodes point to more m-way branches, leaf nodes point to *pin_entry* structures. The *pin_entry* structures contains the name and number of the pin (as in the CDF file) and a pointer to the net that it connects to.

```
/* "@(#)cdf.h 9.44/8/83" */

struct celltree {
        char *c_name;                   /* name of the cell type */
        struct node *c_ptree;           /* pointer to the permutability tree */
        struct celltree *c_left;    /* left child pointer */
        struct celltree *c_right;   /* right child pointer */
```

```
};

struct cellentry {
        char *name;          /* name of the cell type */
        struct node *ptree; /* pointer to the permutability tree */
};

struct pin_entry {
        char *name; /* name of the pin in the CDF file */
        int number; /* number of the pin in the CDF file */
        NET *net;    /* pointer to the net that the pin connects to */
        char p_type; /* type of pin (B I O) */
};

struct node {
        int no_type;                    /* type of node (P N L) */
        int checked;
        union {
                struct mway *mway;      /* ptr to more nodes (types P or N) */
                struct pin_entry pin;   /* pointer to a pin (type L) */
        } ptr;
};

struct mway {
        NODE *node;                     /* pointer to a node */
        struct mway *next;              /* next node */
};

struct pin_list {
        struct pin_entry p_entry;
        struct pin_list *next;
};


/* types of pins */

/* input pin */
#define INPUT 'I'

/* output pin */
#define OUTPUT 'O'

/* bidirectional pin */
#define BIDIRECTIONAL 'B'

/* feed through */
#define FEEDTHRU 'F'


/* types of nodes */

/* leaf node - a pin */
#define LEAF 'L'
```

```
/* permutable node - pins below this can permute */
#define PERM 'P'

/* non-permutable node - pins below this must stay in order */
#define NON 'N'

extern CELLTREE *cell_tree;          /* main permutability template tree */

extern FILE *fcdf;                   /* cdf file */
```

Figure 3.6. Permutability Tree Data Structure

CHAPTER 4

File Formats

4.1. Circuit Files

Circuit files contain a network description (specified by **-ns** and **-nx** flags) and are of the form:

<Instance Name> <Cell Type> {<Net Name>}+

Examples:

x001 nand2 out in1 in2

mdriver Enhancement drain gate source

This format is simple and programs to convert from other formats (SPICE[Vla81a] SPLICE[Sal83a] MOSSIM[Fit81a]) have been written in minutes.

4.2. User-Binding File

The **user-binding file** (specified by the **-ab** flag) contains the initial element and net equivalences, usually derived from labels on the layout. Entries in this file are of the form:

e|n <Name of Item in First Circuit> <Name of Item in Second Circuit>

Examples:

e mdriver m004

n output n0027

Lines Starting with a semicolon (;) are considered comments in all files.

4.3. Node-Alias File

The **node-alias file** (specified by the **-aa** flag) contains which node names in the first circuit are equivalent. The information from this file is used when a dummy element (*wired-or*, *wired-and*) is deleted and the user has specified a node in the user-binding file that has disappeared from the first circuit. Entries in this file are of the form:

<Node Name> {*<Aliases>*}+

Example: *A B C D*

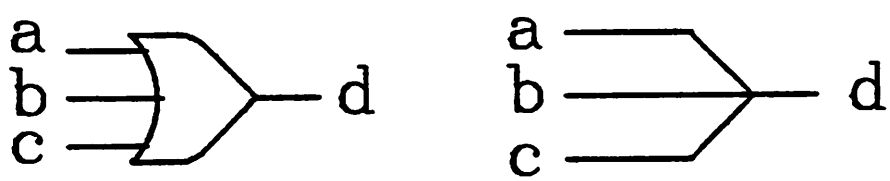


Figure 4.1. Node Alias Entry

4.4. Cell-Descriptor File

The **cell-descriptor file** (specified by the **-cd** flag) specifies how the pins of an element can permute. This particular file format is a simplified version of one developed at Tektronix for an internal IC/CAD system. See appendix B for a detailed description. Permutability entires are only a small part of the entire file; *Wombat* only uses the *cell*, *pin*, *ichng* and *endc* records when it creates the permutability tree.

The following are simplified examples of the *cell*, *pin*, *ichng* and *endc* records:

```
cell aoi
pin i (A 1)
pin i (B 2)
pin i (C 3)
pin i (D 4)
pin o (E 5)
ichng ((A B) (C D))
endc
```

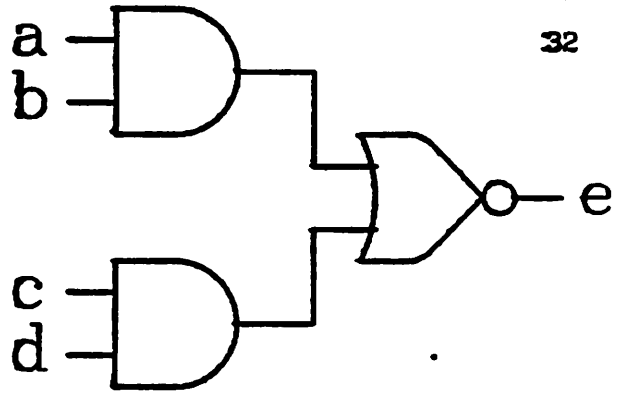


Figure 4.2. AND-OR-INVERT gate

```
cell nor4
pin i (A 1)
pin i (B 2)
pin i (C 3)
pin o (D 4)
ichng (A B C)
endc
```

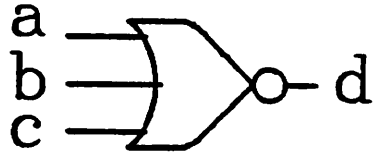


Figure 4.3. NOR gate

4.5. Correspondence File

The **correspondence file** (specified by the **-ac** flag) contains the element and net equivalences that the program has found, including any initial equivalences from the **user-binding file**. It is in the same format as the **user-binding file**.

4.6. Error Files

There are two types of error output files that the user can request (specified by the **-EM** and **-ED** flags). One of the the error files contains all all elements and nets from the first and second circuits that are either unbound

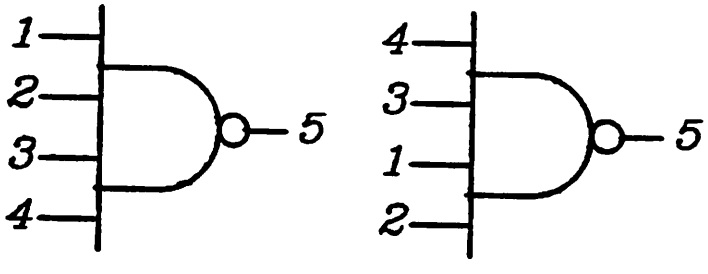
or in error. The other error file contains a list of all elements in the second circuit that are unbound or in error. The format of the second error file is the same as the circuit input files.

4.7. Pin-Permutation File

The **pin-permutation** file contains a list of the elements in the second circuit and how their pins were permuted from the first circuit (specified by the **-ap** flag).

It has the following format:

<InstanceName> <CellType> {<FirstCktPin>:<SecondCktPin>}+



bob and4 5:5 1:4 2:3 3:1 4:2

Figure 4.4. Pin Permutation Example

4.8. Output File

The output file (standard output) contains the following information:

- Initial element and net counts.
- Initial number of elements and nets bound by the user.
- Warning and error messages.
- Number of items bound and errors found on each iteration (verbose-mode).
- Total number of elements and nets bound.
- Total number of errors found in the circuits.
- Computer usage (user time, system time, memory used, page faults).
- Listing of elements and nets in error.

CHAPTER 5

Statistics

5.1. Results

This section summaries the performance of *Wombat* on circuits of varying size and complexity. Some of the examples are used to bring out certain points and others are used just because they happened to be around when the program was being developed.[1]

Table 5.1 shows the execution time and memory used by *Wombat* for a few example circuits. Note that the CPU time is approximately *O(elements)* when the circuit fits into physical memory. When the circuit is too big to fit in physical memory, the performance of *Wombat* degrades because the program psuedo-randomly accesses its data, reducing the effectiveness of page replacement strategies.

element	nets	iterations	memory (Mbytes)	cpu time (sec)		
				total	readin	compare
8194	2826	26	3.7	323	181	142
2336	1374	45	1.3	85	63	22
1088	385	24	0.4	30	21	9
1000	1001	500	0.5	53	22	28
127	284	4	0.1	6	5	1
76	117	3	0.05	3	2	1

Table 5.1.

[1] Unless otherwise stated, all statistics are for a VAX 11/780 with 7 Megabytes of physical memory running the Berkeley 4.2BSD UNIX operating system.

5.2. Circuits Used

This section details why the circuits in Table 5.1 were used. The first circuit is a transistor level digital circuit of 8194 elements; it is used because it just fit into the physical memory of our VAX.[1] The 2336 element circuit is a transistor level ALU and is used to show the effect of latency on execution time (see Table 2.1). The third circuit contains 1088 elements at the transistor level and is used as an example in a large number of projects from the Electronics Research Lab. The 1000 element circuit is a inverter chain and is used to show the effects of latency (see Table 2.1) and degeneracy. The final two circuits are gate arrays of approximately 100 gates and are included as examples of the types circuits that *Wombat* was originally designed to process. The original purpose of *Wombat* was to work in a gate array system where the number of elements would be in the 50 to 200 gate range.

[1] VAX 11/780 with 7 Megabytes of physical memory.

CHAPTER 6

Summary

6.1. Summary of Wombat

Wombat is a program for determining if two circuits are the same or different, and if different, where the differences are. It determines whether elements in the two circuits are the same by calculating signatures for every element and matching unique element signatures between the two circuits. The signature calculation and matching process is an iterative process. On each iteration, signatures are calculated and the elements with unique signatures are bound. At the end of each iteration, the nets connected to the elements bound on the iteration are bound; this gives new information that can be used in the next iterations signature calculation. This process ends when either there are no more elements left to bind or there are no more unique signatures. This program has been used successfully in an industrial design situation.

In the following sections of the report possible extensions to *Wombat* are suggested. These extensions deal with the interface the user sees and with the internal algorithms.

6.2. Interface to a Graphical Design System

Interface *Wombat* to the *Hawk* Viewport Manager and the *Squid* Database Manager[1]. With *Wombat* interfaced to *Hawk*, the user will be able to run

[1]The *Hawk* Viewport Manager and the *Squid* Database Manager are part of Ken Keller's Ph.D project under the direction of Professor Richard Newton.

Wombat until it is done, needs more input, or is interrupted. At this point the user could interactively scan the layout (using pan and zoom, with bound elements and nets highlighted) and graphically bind or unbind elements and nets. Then the user could continue execution of *Wombat*.

Interfacing *Wombat* to *Hawk* will have an added advantage, one of decreased readin time. The netlist information will be stored in a format that can be directly mapped into the *Wombat* data structures. This will save the very expensive readin and parsing step.

6.3. Hashing Rather Than Trees

Changing the comparison tree to a hash table could possibly decrease the memory requirements of the program and could speed up the comparison (a single calculation of the hashing function and table index versus a search of the tree). However, since no gain was made when the net trees were changed to hash tables, it is not clear that changing to a hash table would improve overall program performance.

6.4. Treatment of Hierarchy

If hierarchy were used in *Wombat*, at each level of the hierarchy a relatively small number of cells would have to be looked at and cells used multiple times would have to have their insides looked at only once. This would cause the execution time and memory usage to decrease. It is expected that this will be taken care of when *Wombat* is interfaced to *Hawk*.

6.5. True Signature Function

Using a true signature function (compressing the *key* data structure into a single integer) can decease the memory requirements of the program, and possibly speed up the element comparison (a single integer comparison and signature calculation rather than the N comparisons now being done). One disadvantage of this technique is that some method of handling collisions must be devised.

6.6. Other Signatures

Currently *Wombat* uses the features/properties *fanin*, *fanout*, *cell type* and *bound nets connected to the element* in its signature. There are many other features that could be used in a signature. The LIVES program[Miy81a] uses distances from vertices and distances from terminals in its signatures function. However, the more information that is used about distant elements and nets, the more likely that spurious errors can result early in the comparison process. Later, when elements have been localized, distance could be employed effectively.

6.7. Local verses Global Features

There are many factors to consider when choosing discriminant functions and items to be used in a signature (features). The features[2] used in the signature determine the type of discriminant function and the weighting of the function in relation to the other functions.

Features can be roughly broken up into two types, *Local* and *Global*. Local features consist of the set of features that can be used to describe an

[2] In this section, *features* refers to both derived and intrinsic properties.

element, without any knowledge of the rest of the circuit. Features that fit into this set are *cell type* and *number of pins*. The discriminant functions for local features are usually binary; that is, the output of the function is either TRUE (the two features are exactly the same) or FALSE (the two features are different, with no measure of the distance between them). Global features consist of the open set of features than can be used to describe an element, considering the rest of the circuit and how this particular element interacts with the circuit. Features that fit into this set are *fan-in*, *fan-out*, *types of the elements connected to the element* and *the number of elements that are N connections away from the element*. The discriminant functions for these types of features must not only return that two features differ, but by how much. When a discriminant function for global features returns that two features are not exactly the same, this does not mean that the two elements associated with the two features are different. Differing global features can be caused by errors in the circuit that are up to N connections away, if the features take into account the elements interactions with the circuit N connections away.

Since errors in one part of the circuit could cause errors in the feature calculations for elements in another part of the circuit, the output of the global feature discriminant functions must be weighted to take this into account. The weighting of global feature discriminant functions should be inversely proportional to the distance covered in the calculation. If a feature consists of the number of elements N connections away from the element, the weighting factor for the discriminant function should be $\frac{1}{N}$. By progressively decreasing the weighting factor as features use information farther and farther removed from the element, the area of spurious errors is limited

to the area around the element actually in error, rather than up to N connections away.

6.8. Demand Loading of Cell-Descriptor Entries

If a circuit contains only a fraction of the number of different cell types specified in the **cell-descriptor** file, a lot of time can be wasted in reading in, parsing and setting up data structures for unnecessary cell-descriptor entries. For example, in gate-array systems, the **cell-descriptor** file usually contains all of the cell-descriptor entries for the entire gate-array library, but only 15–20% of the entries are ever used in any given circuit. Since up to 80% of the entire run time is taken up in the readin phase for small circuits, and **cell-descriptor** file readin, parsing and data allocation can be a large fraction of this, this could result in significant savings. This problem could be solved by reading through the file and only storing what cells are in the file and the location in the file. Then in the permutability lookup routine, the program could search the tree, and if the entry exists but is not loaded, the program could *fault* on the entry and force it to be readin from the file and parsed.

6.9. Compiling and Dynamic Loading of the Cell-Descriptor Entries

The problem of parsing **CDF** files could be eliminated by writing C routines that create permutability data structures for each type of cell type. These C routines could be compiled and put into a library, and then dynamically loaded and executed when a permutability data structure needs to be created. These routines need not be written by the user, but could either be generated from the cell layout, or generated from the simple pin permutabil-

ity *tchng* record currently used.

6.10. Better Algorithm for Dealing with Pin Permutability Trees

The algorithms used for dealing with pin permutability are far too complex than should be; efficient and theoretically sound tree compare algorithms should be investigated.

6.11. Identical and Non-Identical Parallel Paths

Non-identical parallel paths are like identical parallel paths, but a small number of the elements differ from path to path. For circuits that contain non-identical parallel paths, signatures like the ones used in LIVES can be used to differentiate between the elements in the paths. But for circuits with identical parallel paths, none of the signature functions described in this report can uniquely define elements in the paths. The simple solution to this problem is to force the user to label some of the elements in parallel paths, but connectivity verifiers should be able to work with no initial information about correspondence between the two circuits wherever possible. Randomly binding the first elements in each one of the paths seems to be the best approach to the problem.

APPENDIX A

Wombat Manual Pages

This appendix contains the UNIX/VMS[1] manual pages for *Wombat*.

[1]UNIX is a Trademark of Bell Laboratories and VMS is a trademark of the Digital Equipment Corporation.

NAME

wombat – compare the connectivity of two networks
(*a program that eats, roots, and leaves*)

SYNOPSIS

wombat **–ns** *net1* **–nx** *net2* [**-cd** *cdf_file*] [**-ab** *b_file*] [**-e**] [**-o**] [**-aa** *a_file*] [**-ac** *c_file*] [**-ap** *p_file*] [**-EM** *error_file*] [**-ED** *error_file*] [**-v**] [**-w**] [**-r** *range*] [**-R**] [**-u**]

DEFINITION

Wombat (*Vombatus Ursinus*): a burrowing, nocturnal, herbivorous marsupial.

PREDATORS

Super-users

DESCRIPTION

–ns *net1* specifies a file containing a network description.

Format (formats are in Backus-Naur format where applicable):

<instance name> <cell type> {<net name>}+

Examples:

x001 buffer dataout datain clock
cell1 nand4 output input1 input2 input3 input4
inv1 inv out in

Note: any line beginning with a ';' is considered a comment

–nx *net2* specifies a file containing a network description.

–cd *cdf_file*

specifies the file containing the pin permutability information for the cell types in the network description. Cell types not specified in the file are assumed to have no permutable pins.

–ab *b_file* specifies the file containing net and element equivalences between the two networks. These elements are considered to be hard bound.

Format:

<type> <first rep> <second rep>

Where <type> is **e** for elements and **n** for nodes.

Examples:

e inverter e0045
n clock 004

Note: any line beginning with a ';' is considered a comment

–e tells the program to equate all elements with the same name. These elements are considered to be hard bound.

-o allows the user to over-ride hard binding (i.e. if the program determines that a user bound elements should not be bound, this option allows the program to unbind the elements.)

-aa *a_file* specifies the file which contains the node alias information. Used primarily for collapsing wired-OR's.

Format:

<name> {<alias>}+

Example:

nodea nodea1 nodea2

Note: any line beginning with a ';' is considered a comment

-ac *c_file* specifies the file generated by the program that lists all of the element and node correspondences between the two representations. Its format is the same as the '*b_file*'.

-ap *p_file* specifies the file generated by the program that lists the pins of each element permuted between the two representations

Format:

<element name> <element type> {<pins>}+

<pins> -> '<name1>:<name2> <unique>'

<name1> is the pin name in the first representation.
<name2> is the pin name in the second representation.
<unique> is '*' if the pin is not connected and ' ' if it is.

-v tells the program to produce lots of output.

-EM *error_file*
specifies the file generated by the program that contains the unbound and incorrectly connected elements.

-ED *error_file*
specifies the file generated by the program that contains the unbound and incorrectly connected elements and nets.

-w tells the program to suppress warning messages.

-r *range* sets the fanout (fanout, fanin and fanbi) comparison accuracy. Used for getting rid of non-essential error messages.

-R reverses the order of comparison.

-u supresses the use of activity.

DIAGNOSTICS

The program informs the user of differences between the connectivity of the two networks. The program also informs the user of bad CDF file, bad command line flags, no memory, bad or non-existent input files, etc.

Error messages are of the form:

TYPE: message (routine)

Where TYPE is either ERROR (fatal) or WARNING, message is the error message itself and routine is the routine which issued the message.

A 'kill -ALRM pid' on WOMBAT will cause the program to flush its output buffer and print the current program status.

KEEPER

Rick L Spickelmier
University of California at Berkeley

SEE ALSO

Spickelmier, Rick L. Verification of Circuit Interconnectivity
cdf(5)

LIMITS

Line length in all files must be less than or equal to 512 characters.
Word length in all files must be less than or equal to 32 characters.
The number of words per line in all files must be less than or equal to 128.

OTHER

WOMBAT works on the *inverse* Carver Mead principle:

- Regularity is bad
- Random logic is good

BUGS

The VMS version does not do runtime cpu and memory usage statistics properly.

APPENDIX B

CDF Manual Page

This appendix contains the UNIX/VMS[1] manual page for *CDF*.

[1]UNIX is a Trademark of Bell Laboratories and VMS is a trademark of the Digital Equipment Corporation.

NAME

cdf – cell descriptor file

SYNOPSIS

Provide pin type and permutability information for WOMBAT

DESCRIPTION

filecontents:	celldescr*
celldescr:	header datarecord* endrecord
header:	"cell" modelname
modelname:	STRING
datarecord:	pin | ichng
pin:	"pin" pintype pinname pinnumber
pintype:	"i"|"o"|"b"|"ip"|"op"|"bp"
pinname:	STRING
pinnumber:	NUMBER
ichng:	"ichng" group
group:	pinname+ | "(" group+ ")" | "[" group+ "]"
endrecord:	"endc"

Cell descriptions start with a cell header, have a set of data records and are ended with an endc record. A "pin" record (there are usually several of these) consists of the pintype, pinname, and pinnumber. The pintype is Input ("i"), Output ("o"), Bi-directional ("b"), Input Pad ("ip"), Output Pad ("op"), or Bi-directional Pad ("bp"). The pinnumber determines the pin order. The "ichng" record contains pin interchangeability information. Pins or groups of pins at a given level of nesting are (are not) interchangeable if enclosed by parentheses (brackets). For example, ([a b] [c d]) indicates that pins 'a' and 'b' are not interchangeable, pins 'c' and 'd' are not interchangeable, but the groups 'a b' and 'c d' are interchangeable.

AUTHORS

Tektronix MCE CAD group
Simplified by Rick Spickelmier (UC Berkeley)

SEE ALSO

Spickelmier, Rick L. Verification of Circuit Interconnectivity

PRODUCED BY

hand

APPENDIX C

Example Runs

This section gives two detailed example runs of *Wombat*, one with two circuit files that are equivalent and one with two circuit files that are different (*i.e.* there is an error in one of the circuits). All examples shown were generated by the UNIX version of *Wombat*.

Normal Run

wombat -v -ns ga.sim -nx ga.ext -ac ga.cor >& ga.out

Simulation File (*ga.sim*)

```
; example simulation input file for WOMBAT
cntr0 t15 n109 n109b hi n109b n111 n110
cntr1 t15 n108 n108b hi n105 n111 n110
cntr2 t15 n107 n107b hi n104 n111 n110
cntr3 t15 n106 n106b hi n102 n111 n110
xnor103 t06h g000001 n105 n108 g000002 n2
xnor102 t06h g000003 n104 n107 g000004 n103
xnor101 t06h g000005 n102 n106 n103 n107
or101 tor2h n103 n2 n108
inv101 tinv2h n2 n109b
or104 tor2h n124 n2 n108b
or105 tor3h n123 n109 n108b n107b
or106 t01h n125 g000006 n109 n106 n107 n108
gate101 t04h g000007 n127 n107 n108 n106b n101
mux101 otmux20h out1 n109 g000008 g000009
mux102 otmux20h out2 n125 n107 n101
mux103 otmux20h out3 n108 n109b n101
mux104 otmux20h out4 n106 n123 n101
mux105 t12h n121 g000010 n107 n106 n101
clock t15 n134 dmy1 n121 n134 n111 n124
sync t09 n133 dmy2 g000011 n127 g000012 n132
inv102 tinv1h n132 n111
in101 itbuf1h n101 mode
in102 itbuf2h n110 ldarrayb
in103 itbuf4h n0 dclk
in104 itbuf4h n1 dclk
in105 itbuf4h n111 dclk
out105 otbuf15h out5 n133
out106 otbuf15h out6 n134
chrcolor t15 n118 n118b n116 n118 n111 n123
opaque t15 n119 dmy3 n115 n119 n111 n123
blank0 t15 n12 dmy4 n114 n12 n111 n123
andor101 t05h g000013 n120 n112 n119 n112b n118b
or109 t01h n11 g000014 n112 n119 n113 n12
mux106 otmux20h out7 n137 n140 n101
mux107 t12h n13 g000015 n120 n112 n101
muxsel t09 n136 dmy5 g000016 n13 g000017 n111
ovrlay t09 n5b n5 g000018 n11 g000019 n111
blank1 t09 n137 dmy7 g000020 n12 g000021 n111
in106 it01 n113 g000022 cursor n112 n112b char
in107 itbuf1h n114 blank
in108 itbuf1h n115 opaque
in109 itbuf1h n116 chrcolor
out108 otbuf20h out8 n136
```

```
or110 tnor2h n138 n6 n137
gate102 t04h n139 g000023 n137 g000024 n6 g000025
test t15 n140 dmy8 n139 n138 n111 dat11
in209 itbuf2h n233 ovrlyblu
in216 itbuf2h n234 ovrlygrn
in208 itbuf1h n214 map00
in207 itbuf1h n213 map10
in206 itbuf1h n210 map01
in205 itbuf1h n209 map11
in204 itbuf1h n206 map02
in203 itbuf1h n205 map12
in202 itbuf1h n202 map03
in201 itbuf1h n201 map13
out204 otbuf15h dac0 n216
out203 otbuf15h dac1 n212
out202 otbuf15h dac2 n208
out201 otbuf15h dac3 n204
flop201 t09 n203 dmy9 g000026 n201 g000027 n0
mux201 t12h dat3 g000028 n203 n202 n4
flop202 t15 n204 dmy10 n233 dat3 n0 n5
flop203 t09 n207 dmy11 g000029 n205 g000030 n0
mux202 t12h dat2 g000031 n207 n206 n4
flop204 t15 n208 dmy12 n233 dat2 n0 n5
flop205 t09 n211 dmy13 g000032 n209 g000033 n0
mux203 t12h dat1 g000034 n211 n210 n4
flop206 t15 n212 dmy14 n233 dat1 n0 n5
flop207 t09 n215 dmy15 g000035 n213 g000036 n0
mux204 t12h dat0 g000037 n215 n214 n4
flop208 t15 n216 dmy16 n233 dat0 n0 n5
in215 it06 n230 map04 g000038 n229 map14 g000039
in214 it06 n226 map05 g000040 n225 map15 g000041
in213 itbuf1h n222 map06
in212 itbuf1h n221 map16
in211 itbuf1h n218 map07
in210 itbuf1h n217 map17
out208 otbuf15h dac4 n232
out207 otbuf15h dac5 n228
out206 otbuf15h dac6 n224
out205 otbuf15h dac7 n220
flop209 t09 n219 dmy17 g000042 n217 g000043 n0
mux205 t12h dat7 g000044 n219 n218 n4
flop210 t15 n220 dmy18 n234 dat7 n1 n5
flop211 t09 n223 dmy19 g000045 n221 g000046 n0
mux206 t12h dat6 g000047 n223 n222 n4
flop212 t15 n224 dmy20 n234 dat6 n1 n5
flop213 t09 n227 dmy21 g000048 n225 g000049 n0
mux207 t12h dat5 g000050 n227 n226 n4
flop214 t15 n228 dmy22 n234 dat5 n1 n5
flop215 t09 n231 dmy23 g000051 n229 g000052 n0
mux208 t12h dat4 g000053 n231 n230 n4
flop216 t15 n232 dmy24 n234 dat4 n1 n5
in305 itbuf2h n6 ovrlyred
in304 it06 n314 map08 g000054 n313 map18 g000055
```

```
in303 it06 n310 map09 g000056 n309 map19 g000057
in302 it06 n306 map010 g000058 n305 map110 g000059
in301 it06 n302 map011 g000060 n301 map111 g000061
out304 otbuf15h dac8 n316
out303 otbuf15h dac9 n312
out302 otbuf15h dac10 n308
out301 otbuf15h dac11 n304
flop301 t09 n303 dmy25 g000062 n301 g000063 n1
mux307 t12h dat11 g000064 n302 n303 n4b
flop302 t15 n304 dmy26 dat11 n6 n1 n5b
flop303 t09 n307 dmy27 g000065 n305 g000066 n1
mux308 t12h dat10 g000067 n306 n307 n4b
flop304 t15 n308 dmy28 dat10 n6 n1 n5b
flop305 t09 n311 dmy29 g000068 n309 g000069 n1
mux309 t12h dat9 g000070 n310 n311 n4b
flop306 t15 n312 dmy30 dat9 n6 n1 n5b
flop307 t09 n315 dmy31 g000071 n313 g000072 n1
mux310 t12h dat8 g000073 n314 n315 n4b
flop308 t15 n316 dmy32 dat8 n6 n1 n5b
in306 itbuf1h n319 maprdb
in307 it06 n317 maprdb ovrlygrn n318 maprdb ovrlyblu
mux302 t12h n4b n4 n6 n2 n319
mux306 t11 n323 dat0 dat4 dat8 n5b hi n318 n317
mux305 t11 n322 dat1 dat5 dat9 n11 hi n318 n317
mux304 t11 n321 dat2 dat6 dat10 n12 hi n318 n317
mux303 t11 n320 dat3 dat7 dat11 n13 hi n318 n317
out305 otbuf15h data3 n320
out306 otbuf15h data2 n321
out307 otbuf15h data1 n322
out308 otbuf15h data0 n323
inv301 tinv1h hi g000074
```

Extracted File (*ga.ext*)

```
x0 t05h 0 1 2 3 4 5
x1 t15 6 7 8 6 9 10
x2 t09 11 12 13 6 14 9
x3 t15 15 5 16 15 9 10
x4 t15 17 18 19 17 9 20
x5 t15 21 22 23 22 9 24
x6 t15 25 26 23 27 9 24
x7 t15 28 29 23 30 9 24
x8 t15 31 32 23 33 9 24
x9 t09 34 35 36 37 38 39
x10 t15 40 41 42 43 39 44
x11 t09 45 46 47 48 49 39
x12 t15 50 51 42 52 39 44
x13 t09 53 54 55 56 57 39
x14 t15 58 59 42 60 39 44
x15 t09 61 62 63 64 65 39
x16 t15 66 67 42 68 39 44
x17 t09 69 70 71 72 73 39
x18 t15 74 75 76 77 78 44
x19 t09 79 80 81 82 83 39
x20 t15 84 85 76 86 78 44
x21 t09 87 88 89 90 91 39
x22 t15 92 93 76 94 78 44
x23 t09 95 96 97 98 99 39
x24 t15 100 101 76 102 78 44
x25 t09 103 104 105 106 107 78
x26 t15 108 109 110 111 78 112
x27 t09 113 114 115 116 117 78
x28 t15 118 119 120 111 78 112
x29 t09 121 122 123 124 125 78
x30 t15 126 127 128 111 78 112
x31 t09 129 130 131 132 133 78
x32 t15 134 135 136 111 78 112
x33 t04h 137 138 28 25 32 139
x34 t04h 140 141 11 142 111 143
x35 itbuf1h 139 144
x36 itbuf2h 24 145
x37 itbuf4h 39 146
x38 itbuf4h 78 146
x39 itbuf4h 9 146
x40 it01 147 148 149 2 4 150
x41 itbuf1h 8 151
x42 itbuf1h 152 153
x43 itbuf1h 16 154
x44 itbuf1h 37 155
x45 itbuf1h 156 157
x46 itbuf1h 48 158
x47 itbuf1h 159 160
x48 itbuf1h 56 161
x49 itbuf1h 162 163
x50 itbuf1h 64 164
x51 itbuf1h 165 166
x52 itbuf2h 42 167
x53 itbuf1h 72 168
x54 itbuf1h 169 170
x55 itbuf1h 82 171
x56 itbuf1h 172 173
x57 it06 174 175 176 90 177 178
x58 it06 179 180 181 98 182 183
x59 itbuf2h 76 184
x60 it06 185 186 187 106 188 189
x61 it06 190 191 192 116 193 194
x62 it06 195 196 197 124 198 199
x63 it06 200 201 202 132 203 204
x64 itbuf2h 111 205
x65 itbuf1h 206 207
x66 it06 208 207 184 209 207 167
x67 tinv2h 210 22
x68 tinv1h 211 9
x69 tinv1h 23 212
x70 otmux20h 213 21 214 215
x71 otmux20h 216 217 28 139
x72 otmux20h 218 25 22 139
x73 otmux20h 219 31 10 139
x74 t12h 19 220 28 31 139
x75 otmux20h 221 11 222 139
x76 t12h 223 224 1 2 139
x77 t12h 43 225 34 156 226
x78 t12h 52 227 45 159 226
x79 t12h 60 228 53 162 226
x80 t12h 68 229 61 165 226
x81 t12h 77 230 69 169 226
x82 t12h 86 231 79 172 226
x83 t12h 94 232 87 174 226
x84 t12h 102 233 95 179 226
x85 t12h 234 226 111 210 206
x86 t11 235 43 77 110 223 23 209 208
x87 t11 236 52 86 120 6 23 209 208
x88 t11 237 60 94 128 238 23 209 208
x89 t11 239 68 102 136 112 23 209 208
x90 t12h 110 240 185 103 234
x91 t12h 120 241 190 113 234
x92 t12h 128 242 195 121 234
x93 t12h 136 243 200 129 234
x94 t09 244 245 246 223 247 9
x95 t15 3 248 152 3 9 10
x96 tor2h 249 210 25
x97 tor2h 20 210 26
x98 tor3h 10 21 26 29
x99 t01h 217 250 21 31 28 25
```

```
x100 t01h 238 251 2 3 147 6
x101 tnor2h 252 111 11
x102 otbuf15h 253 254
x103 otbuf15h 255 17
x104 otbuf20h 256 244
x105 otbuf15h 257 40
x106 otbuf15h 258 50
x107 otbuf15h 259 58
x108 otbuf15h 260 66
x109 otbuf15h 261 74
x110 otbuf15h 262 84
x111 otbuf15h 263 92
x112 otbuf15h 264 100
x113 otbuf15h 265 108
x114 otbuf15h 266 118
x115 otbuf15h 267 126
x116 otbuf15h 268 134
x117 otbuf15h 269 235
x118 otbuf15h 270 236
x119 otbuf15h 271 237
x120 otbuf15h 272 239
x121 t09 112 44 273 238 274 9
x122 t09 254 275 276 138 277 211
x123 t15 222 278 140 252 9 110
x124 t06h 279 33 31 249 28
x125 t06h 280 30 28 281 249
x126 t06h 282 27 25 283 210
```

Output File (*ga.out*)

```
verbose mode set
simulation file: ga.sim
extracted file: ga.ext
cell and node correspondence file: ga.cor

WOMBAT 8.1 run for ga.sim ga.ext on Tue Dec 14 17:50:01 1982


reading in the first network file
reading in the second network file

number of elements in first file = 127
number of elements in second file = 127
number of nets in first file = 284
number of nets in second file = 284
number of initially bound elements = 0 (  0.00%)
number of initially bound nets = 0 (  0.00%)
setting up the fans

comparison loop
using activity

iteration count = 1
bin count = 61
number of bound elements = 51 ( 40.16%)
incremental bound elements = 51
number of bound nets = 122 ( 42.96%)
incremental bound nets = 122
connectivity error count = 0

iteration count = 2
bin count = 38
number of bound elements = 88 ( 69.29%)
incremental bound elements = 37
number of bound nets = 197 ( 69.37%)
incremental bound nets = 75
connectivity error count = 0

iteration count = 3
bin count = 33
number of bound elements = 121 ( 95.28%)
incremental bound elements = 33
number of bound nets = 278 ( 97.89%)
incremental bound nets = 81
connectivity error count = 0

iteration count = 4
bin count = 6
number of bound elements = 127 (100.00%)
incremental bound elements = 6
```

number of bound nets = 284 (100.00%)
incremental bound nets = 8
connectivity error count = 0

number of iterations = 4

total number of elements = 127
total number of bound elements = 127 (100.00%)
total number of nets = 284
total number of bound nets = 284 (100.00%)

connectivity error count = 0

hard binding breaks = 0
hard binding skips = 0

number of collisions = 1
maximum bin count = 61

maximum allocated memory = 106764

maximum resident set size = 302

total number of major page faults = 6
total number of minor page faults = 31
total number of swaps = 0

total real time = 6 seconds
total user time = 5.25 seconds
total sys time = 0.65 seconds
system utilization ((utime+stime)/real_time) = 0.9833
parse user time = 0.00 seconds
parse sys time = 0.00 seconds
element readin and setup user time = 4.43 seconds
element readin and setup sys time = 0.37 seconds
user bind and node alias user time = 0.00 seconds
user bind and node alias sys time = 0.00 seconds
create user time = 0.17 seconds
create sys time = 0.07 seconds
lookup user time = 0.37 seconds
lookup system time = 0.02 seconds

Element/Net Correspondence File (*ga.cor*)

```
; element and node correspondence
; file for ga.sim ga.ext
; written by WOMBAT 8.1 on
; Tue Dec 14 17:50:07 1982
e inv301 x69
e out308 x120
e out307 x119
e out306 x118
e out305 x117
e mux303 x86
e mux304 x87
e mux305 x88
e mux306 x89
c mux302 x85
e in307 x66
e in306 x65
e flop308 x32
e mux310 x93
e flop307 x31
e flop306 x30
e mux309 x92
e flop305 x29
e flop304 x28
e mux308 x91
e flop303 x27
e flop302 x26
e mux307 x90
e flop301 x25
e out301 x113
e out302 x114
e out303 x115
e out304 x116
e in301 x60
e in302 x61
e in303 x62
e in304 x63
e in305 x64
e flop216 x24
e mux208 x84
e flop215 x23
e flop214 x22
e mux207 x83
e flop213 x21
e flop212 x20
e mux206 x82
e flop211 x19
e flop210 x18
e mux205 x81
e flop209 x17
e out205 x109
e out206 x110
e out207 x111
e out208 x112
e in210 x53
e in211 x54
e in212 x55
e in213 x56
e in214 x57
e in215 x58
e flop208 x16
e mux204 x80
e flop207 x15
e flop206 x14
c mux203 x79
e flop205 x13
e flop204 x12
e mux202 x78
e flop203 x11
e flop202 x10
e mux201 x77
e flop201 x9
e out201 x105
e out202 x106
e out203 x107
e out204 x108
e in201 x44
e in202 x45
e in203 x46
e in204 x47
e in205 x48
e in206 x49
e in207 x50
e in208 x51
e in216 x59
e in209 x52
e test x123
e gate102 x34
e or110 x101
e out108 x104
e in109 x43
e in108 x42
e in107 x41
e in106 x40
e blank1 x2
e ovrlay x121
e muxsel x94
e mux107 x76
e mux106 x75
e or109 x100
e andor101 x0
```

e blank0 x1
e opaque x95
e chrcolor x3
e out106 x103
e out105 x102
e in105 x39
e in104 x38
e in103 x37
e in102 x36
e in101 x35
e inv102 x88
e sync x122
e clock x4
e mux105 x74
e mux104 x73
e mux103 x72
e mux102 x71
e mux101 x70
e gate101 x33
e or106 x99
e or105 x98
e or104 x97
e inv101 x87
e or101 x96
e xnor101 x124
e xnor102 x125
e xnor103 x126
e cntr3 x8
e cntr2 x7
e cntr1 x6
e cntr0 x5
n blank 151
n char 150
n chrcolor 154
n cursor 149
n dac0 260
n dac1 259
n dac10 266
n dac11 265
n dac2 258
n dac3 257
n dac4 264
n dac5 263
n dac6 262
n dac7 261
n dac8 268
n dac9 267
n dat0 68
n dat1 60
n dat10 120
n dat11 110
n dat2 52
n dat3 43
n dat4 102
n dat5 94
n dat6 86
n dat7 77
n dat8 136
n dat9 128
n data0 272
n data1 271
n data2 270
n data3 269
n dclk 146
n dmy1 18
n dmy10 41
n dmy11 46
n dmy12 51
n dmy13 54
n dmy14 59
n dmy15 62
n dmy16 67
n dmy17 70
n dmy18 75
n dmy19 80
n dmy2 275
n dmy20 85
n dmy21 88
n dmy22 93
n dmy23 96
n dmy24 101
n dmy25 104
n dmy26 109
n dmy27 114
n dmy28 119
n dmy29 122
n dmy3 248
n dmy30 127
n dmy31 130
n dmy32 135
n dmy4 7
n dmy5 245
n dmy7 12
n dmy8 278
n dmy9 35
n g000001 282
n g000002 283
n g000003 280
n g000004 281
n g000005 279
n g000006 250
n g000007 137
n g000008 214
n g000009 215
n g000010 220
n g000011 276

n g000012 277
n g000013 0
n g000014 251
n g000015 224
n g000016 246
n g000017 247
n g000018 273
n g000019 274
n g000020 13
n g000021 14
n g000022 148
n g000023 141
n g000024 142
n g000025 143
n g000026 36
n g000027 38
n g000028 225
n g000029 47
n g000030 49
n g000031 227
n g000032 55
n g000033 57
n g000034 228
n g000035 63
n g000036 65
n g000037 229
n g000038 181
n g000039 183
n g000040 176
n g000041 178
n g000042 71
n g000043 73
n g000044 230
n g000045 81
n g000046 83
n g000047 231
n g000048 89
n g000049 91
n g000050 232
n g000051 97
n g000052 99
n g000053 233
n g000054 202
n g000055 204
n g000056 197
n g000057 199
n g000058 192
n g000059 194
n g000060 187
n g000061 189
n g000062 105
n g000063 107
n g000064 240
n g000065 115
n g000066 117
n g000067 241
n g000068 123
n g000069 125
n g000070 242
n g000071 131
n g000072 133
n g000073 243
n g000074 212
n hi 23
n ldarrayb 145
n map00 166
n map01 163
n map010 191
n map011 186
n map02 160
n map03 157
n map04 180
n map05 175
n map06 173
n map07 170
n map08 201
n map09 196
n map10 164
n map11 161
n map110 193
n map111 188
n map12 158
n map13 155
n map14 182
n map15 177
n map16 171
n map17 168
n map18 203
n map19 198
n maprdb 207
n mode 144
n n0 39
n n1 78
n n101 139
n n102 33
n n103 249
n n104 30
n n105 27
n n106 31
n n106b 32
n n107 28
n n107b 29
n n108 25
n n108b 26
n n109 21
n n109b 22

n n11 238
n n110 24
n n111 9
n n112 2
n n112b 4
n n113 147
n n114 8
n n115 152
n n116 16
n n118 15
n n118b 5
n n119 3
n n12 6
n n120 1
n n121 19
n n123 10
n n124 20
n n125 217
n n127 138
n n13 223
n n132 211
n n133 254
n n134 17
n n136 244
n n137 11
n n138 252
n n139 140
n n140 222
n n2 210
n n201 37
n n202 156
n n203 34
n n204 40
n n205 48
n n206 159
n n207 45
n n208 50
n n209 56
n n210 162
n n211 53
n n212 58
n n213 64
n n214 165
n n215 61
n n216 66
n n217 72
n n218 169
n n219 69
n n220 74
n n221 82
n n222 172
n n223 79
n n224 84
n n225 90
n n226 174
n n227 87
n n228 92
n n229 98
n n230 179
n n231 95
n n232 100
n n233 42
n n234 76
n n301 106
n n302 185
n n303 103
n n304 108
n n305 116
n n306 190
n n307 113
n n308 118
n n309 124
n n310 195
n n311 121
n n312 126
n n313 132
n n314 200
n n315 129
n n316 134
n n317 208
n n318 209
n n319 206
n n320 235
n n321 236
n n322 237
n n323 239
n n4 226
n n4b 234
n n5 44
n n5b 112
n n6 111
n opaque 153
n out1 213
n out2 216
n out3 218
n out4 219
n out5 253
n out6 255
n out7 221
n out8 256
n ovrlyblu 167
n ovrlygrn 184
n ovrlyred 205

Run with Errors

The following run simulates a missing contact on the layout. Node 28 (n107 in the simulation file) has been split into nodes 28 and 999 (28 changed to 999 on element x99). Note: all elements marked as connected incorrectly are the only ones connected to node 28. Also, warning message at beginning.

wombat -v -ns ga.sim -nx err.ext -ac err.cor >& err.out

Extracted File (*err.ext*)

x0 t05h 0 1 2 3 4 5
x1 t15 6 7 8 6 9 10
x2 t09 11 12 13 6 14 9
x3 t15 15 5 16 15 9 10
x4 t15 17 18 19 17 9 20
x5 t15 21 22 23 22 9 24
x6 t15 25 26 23 27 9 24
x7 t15 28 29 23 30 9 24
x8 t15 31 32 23 33 9 24
x9 t09 34 35 36 37 38 39
x10 t15 40 41 42 43 39 44
x11 t09 45 46 47 48 49 39
x12 t15 50 51 42 52 39 44
x13 t09 53 54 55 56 57 39
x14 t15 58 59 42 60 39 44
x15 t09 61 62 63 64 65 39
x16 t15 66 67 42 68 39 44
x17 t09 69 70 71 72 73 39
x18 t15 74 75 76 77 78 44
x19 t09 79 80 81 82 83 39
x20 t15 84 85 76 86 78 44
x21 t09 87 88 89 90 91 39
x22 t15 92 93 76 94 78 44
x23 t09 95 96 97 98 99 39
x24 t15 100 101 76 102 78 44
x25 t09 103 104 105 106 107 78
x26 t15 108 109 110 111 78 112
x27 t09 113 114 115 116 117 78
x28 t15 118 119 120 111 78 112
x29 t09 121 122 123 124 125 78
x30 t15 126 127 128 111 78 112
x31 t09 129 130 131 132 133 78
x32 t15 134 135 136 111 78 112
x33 t04h 137 138 28 25 32 139
x34 t04h 140 141 11 142 111 143
x35 itbuf1h 139 144
x36 itbuf2h 24 145
x37 itbuf4h 39 146
x38 itbuf4h 78 146
x39 itbuf4h 9 146
x40 it01 147 148 149 2 4 150
x41 itbuf1h 8 151
x42 itbuf1h 152 153
x43 itbuf1h 16 154
x44 itbuf1h 37 155
x45 itbuf1h 156 157
x46 itbuf1h 48 158
x47 itbuf1h 159 160
x48 itbuf1h 56 161
x49 itbuf1h 162 163
x50 itbuf1h 64 164
x51 itbuf1h 165 166
x52 itbuf2h 42 167
x53 itbuf1h 72 168
x54 itbuf1h 169 170
x55 itbuf1h 82 171
x56 itbuf1h 172 173
x57 it06 174 175 176 90 177 178
x58 it06 179 180 181 98 182 183
x59 itbuf2h 76 184
x60 it06 185 186 187 106 188 189
x61 it06 190 191 192 116 193 194
x62 it06 195 196 197 124 198 199
x63 it06 200 201 202 132 203 204
x64 itbuf2h 111 205
x65 itbuf1h 206 207
x66 it06 208 207 184 209 207 167
x67 tinv2h 210 22
x68 tinv1h 211 9
x69 tinv1h 23 212

x70 otmux20h 213 21 214 215
x71 otmux20h 216 217 28 139
x72 otmux20h 218 25 22 139
x73 otmux20h 219 31 10 139
x74 t12h 19 220 28 31 139
x75 otmux20h 221 11 222 139
x76 t12h 223 224 1 2 139
x77 t12h 43 225 34 156 226
x78 t12h 52 227 45 159 226
x79 t12h 60 228 53 162 226
x80 t12h 68 229 61 165 226
x81 t12h 77 230 69 169 226
x82 t12h 86 231 79 172 226
x83 t12h 94 232 87 174 226
x84 t12h 102 233 95 179 226
x85 t12h 234 226 111 210 206
x86 t11 235 43 77 110 223 23 209 208
x87 t11 236 52 86 120 6 23 209 208
x88 t11 237 60 94 128 238 23 209 208
x89 t11 239 68 102 136 112 23 209 208
x90 t12h 110 240 185 103 234
x91 t12h 120 241 190 113 234
x92 t12h 128 242 195 121 234
x93 t12h 136 243 200 129 234
x94 t09 244 245 246 223 247 9
x95 t15 3 248 152 3 9 10
x96 tor2h 249 210 25
x97 tor2h 20 210 26
x98 tor3h 10 21 26 29
x99 t01h 217 250 21 31 999 25
x100 t01h 238 251 2 3 147 6
x101 tnor2h 252 111 11
x102 otbuf15h 253 254
x103 otbuf15h 255 17
x104 otbuf20h 256 244
x105 otbuf15h 257 40
x106 otbuf15h 258 50
x107 otbuf15h 259 58
x108 otbuf15h 260 66
x109 otbuf15h 261 74
x110 otbuf15h 262 84
x111 otbuf15h 263 92
x112 otbuf15h 264 100
x113 otbuf15h 265 108
x114 otbuf15h 266 118
x115 otbuf15h 267 126
x116 otbuf15h 268 134
x117 otbuf15h 269 235
x118 otbuf15h 270 236
x119 otbuf15h 271 237
x120 otbuf15h 272 239
x121 t09 112 44 273 238 274 9
x122 t09 254 275 276 138 277 211
x123 t15 222 278 140 252 9 110
x124 t06h 279 33 31 249 28
x125 t06h 280 30 28 281 249
x126 t06h 282 27 25 283 210

Output File (*err.out*)

```
verbose mode set
simulation file: ga.sim
extracted file: err.ext
cell and node correspondence file: err.cor

WOMBAT 8.1 run for ga.sim err.ext on Tue Dec 14 17:53:12 1982


reading in the first network file
reading in the second network file
WARNING: net counts differ (main)

number of elements in first file = 127
number of elements in second file = 127
number of nets in first file = 284
number of nets in second file = 285
number of initially bound elements = 0 (  0.00%)
number of initially bound nets = 0 (  0.00%)
setting up the fans

comparison loop
using activity
connectivity error - element (2) x125 : t06h
connectivity error - element (2) x124 : t06h
connectivity error - element (2) x99 : t01h
connectivity error - element (2) x74 : t12h
connectivity error - element (2) x71 : otmux20h
connectivity error - element (2) x33 : t04h
connectivity error - element (2) x7 : t15

iteration count = 1
bin count = 61
number of bound elements = 44 ( 34.65%)
incremental bound elements = 44
number of bound nets = 112 ( 39.44%)
incremental bound nets = 112
connectivity error count = 7

iteration count = 2
bin count = 45
number of bound elements = 81 ( 63.78%)
incremental bound elements = 37
number of bound nets = 187 ( 65.85%)
incremental bound nets = 75
connectivity error count = 7

iteration count = 3
bin count = 33
number of bound elements = 114 ( 89.76%)
incremental bound elements = 33
```

number of bound nets = 268 (94.37%)
incremental bound nets = 81
connectivity error count = 7

iteration count = 4
bin count = 6
number of bound elements = 120 (94.49%)
incremental bound elements = 6
number of bound nets = 274 (96.48%)
incremental bound nets = 6
connectivity error count = 7

number of iterations = 4

total number of elements = 127
total number of bound elements = 120 (94.49%)
total number of nets = 284
total number of bound nets = 274 (96.48%)

connectivity error count = 7

hard binding breaks = 0
hard binding skips = 0

number of collisions = 1
maximum bin count = 61

maximum allocated memory = 106712

maximum resident set size = 302

total number of major page faults = 6
total number of minor page faults = 31
total number of swaps = 0

total real time = 8 seconds
total user time = 5.15 seconds
total sys time = 0.67 seconds
system utilization ((utime+stime)/real_time) = 0.7271
parse user time = 0.00 seconds
parse sys time = 0.00 seconds
element readin and setup user time = 4.37 seconds
element readin and setup sys time = 0.42 seconds
user bind and node alias user time = 0.00 seconds
user bind and node alias sys time = 0.00 seconds
create user time = 0.15 seconds
create sys time = 0.08 seconds
lookup user time = 0.35 seconds
lookup system time = 0.02 seconds

first element list

mux105 t12h : not bound
mux102 otmux20h : not bound
gate101 t04h : not bound
or106 t01h : not bound
xnor101 t06h : not bound
xnor102 t06h : not bound
cntr2 t15 : not bound

first net list

g000003 : not bound
g000004 : not bound
g000005 : not bound
g000006 : not bound
g000007 : not bound
g000010 : not bound
n104 : not bound
n107 : not bound
n125 : not bound
out2 : not bound

second element list

x125 t06h : connected incorrectly
x124 t06h : connected incorrectly
x99 t01h : connected incorrectly
x74 t12h : connected incorrectly
x71 otmux20h : connected incorrectly
x33 t04h : connected incorrectly
x7 t15 : connected incorrectly

second net list

137 : not bound
216 : not bound
217 : not bound
220 : not bound
250 : not bound
279 : not bound
28 : not bound
280 : not bound
281 : not bound
30 : not bound
999 : not bound

APPENDIX D

Program Listing

Wombat consists of approximately 4000 lines of code and comments in the C programming language.[Ker78a] The code consists of 63 routines; with 16 for pin-permutability. The program runs under UNIX 4.1BSD and VMS 3.0[1]

For a program listing/tape[2] or a copy of this report contact Pamela Bostlemann at following address:

Pamela Bostlemann
c/o Industrial Liaison Program
Electronics Research Lab
499 Cory Hall
University of California at Berkeley
Berkeley, CA 94720

[1]UNIX is a Trademark of Bell Laboratories and VMS is a Trademark of DEC.

[2]Be sure to specify the operating system you wish the program run on (UNIX or VMS).

APPENDIX E

Glossary of Terms

This appendix contains definitions of terms used in this report.

AVL trees

AVL trees are height balanced trees with a height mismatch of no more than 1. In other words, the two subtrees at any node of the tree can not differ in height by more than 1.

bound

when two items (nets, elements) are found to be equivalent, the items are bound. Two items are equivalent when each one has the same unique signature.

CDF see **cell-descriptor file.**

cell-descriptor file

a file that among other things, contains information on how the pins on an element can be allowed to permute with no change in the electrical or logical function of the element. See Appendix B.

circuit

a collection of elements connected together by nets.

degenerate circuit

an N element circuit that takes $\frac{N}{2}$ iterations to bind all elements. An

inverter chain is an example of this.

discriminant function

a function used to calculate the distance between two features.

distance

a measure of the difference between two signatures or features.

DRC design rule checker. Used to verify that spacing requirements on an Integrated Circuit layout are met.

ERC electrical rule checker. Used to verify that electrical rules on an Integrated Circuit layout are met.

element

a black box that is connected to other black boxes via nets and pins. In the context of this report an element is an instance of a cell (NAND Gate, MOSFET, ALU).

feature

a derived property of an item (*i.e.* fanout, elements/nets connected to it). As opposed to intrinsic properties (cell type).

hard-bound

element or net bound by the user (as opposed to bound by the program). Usually from labels found on the layout. This information is specified in the **user-binding file**.

second graph.

net a means of connecting elements together. A collection of pins.

non-identical parallel paths

two or more groups of elements that are almost the same.

permutable

electrical/logical equivalence. Examples of this are the source and drain equivalence of MOSFETS, and the equivalence of the inputs to a NAND gate.

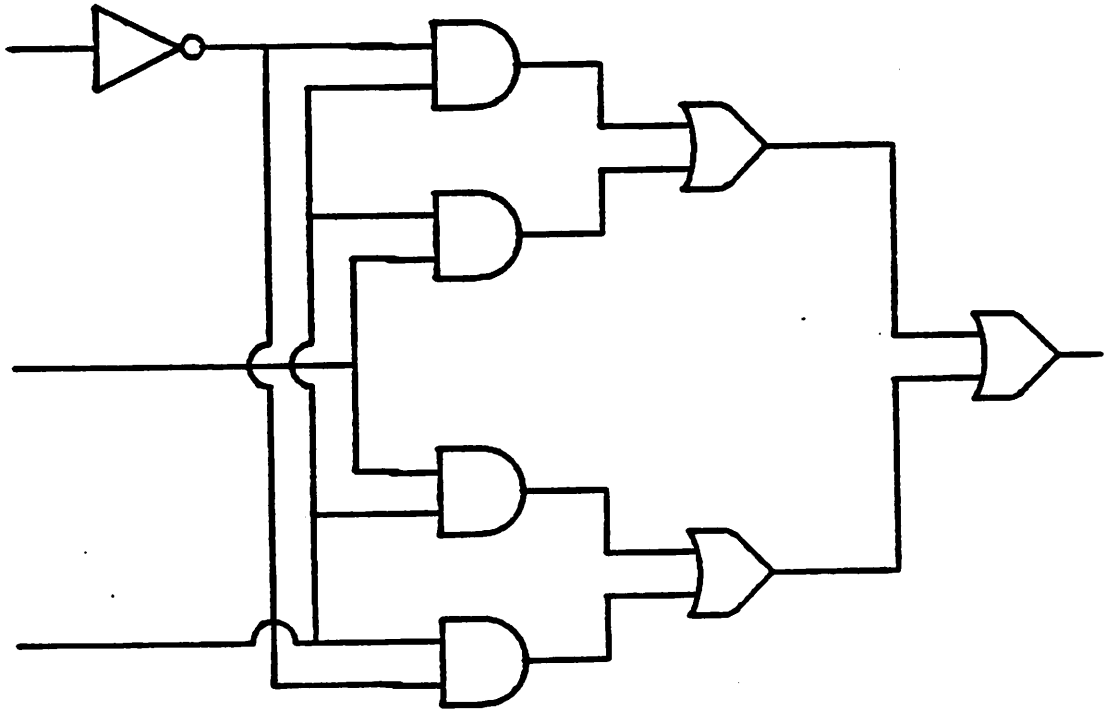
pin the place where an element connects to other elements through nets. The inputs and outputs of the element.

signature

a set of features for an element; the information used in comparisons with other elements (fanin, fanout, type). Possibly in condensed form.

identical parallel paths

two or more groups of elements that are totally identical.



Identical Parallel Paths

intrinsic Property

a property of an item that stays the same no matter where the item is in the circuit, *e.g.* cell type, number of inputs.

isomorphism

two graphs are **isomorophic** if there is a one-to-one correspondence between the vertices of the two graphs such that for every edge in that connects two vertices in the first graph, there is a corresponding edge in the second graph that connects the two corresponding vertices in the

Bibliography

References

Ker78a.

B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

All77a.

R.M. Allgair and D.S. Evans, "A Comprehensive Approach to a Connectivity Audit, or a Fruitful Comparison of Apples and Oranges," *The Proceedings of the Design Automation Conference* 33 pp. 312-321 (1977).

Miy81a.

N. Miyahara, T. Watanabe, M. Endo, and S. Yamada, "A New CAD System for Automatic Logic Interconnection Verification," *The Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 114-117 (1981).

Hor76a.

Ellis Horowitz and Sartaj Sahni, *Fundamentals of Data Structures*, Computer Science Press, Potomac, Maryland (1976).

Vla81a.

A. Vladimirescu, K. Zhang, A.R. Newton, D.O. Pederson, and A. Sangiovanni-Vincentelli, *SPICE Version 2G User's Guide*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, Cailfornia (August 1981).

Sal83a.

R.A. Saleh, J.E. Kleckner, and A.R. Newton, *SPLICE Version 1.6 User's Guide*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California (April 1983).

Fit81a.

Dan Fitzpatrick, *MEXTRA: A Manhattann Circuit Extractor*, Computer Science Departmemnt, University of California, Berkeley, California (May 1981).