ANL/DIS/TM-43

Assigning Functional Meaning to Digital Circuits

by S.T. Eckmann and G.H. Chisholm

Decision and Information Sciences Division,
Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439

July 1997

Work sponsored by the United States Department of Defense

[illegible]

CONTENTS

ACKNOWLEDGMENTS v

ABSTRACT 1

1 REPORT OBJECTIVE, SCOPE, AND ORGANIZATION 1

2 BACKGROUND 2

2.1 Reverse Engineering 2
2.2 Reverse Engineering Assistant 2
2.3 Example Reverse Engineering Problem 2

3 CANONICAL REPRESENTATIONS 6

3.1 Representing Circuits as Logical Formulas 6
3.2 `exor/and` Canonical Form 7
3.3 Binary Decision Diagrams 11

4 CANONICALIZATION ISSUES 15

4.1 Input Order for `exor/and` Representation 15
4.2 Input Order for Binary Decision Diagram Representation 17

5 CONCLUSIONS AND RECOMMENDATIONS 24

6 BIBLIOGRAPHY 25

APPENDIX A: Rewriting 2-Bit Adder Outputs to Canonical Form 27

APPENDIX B: Equivalence of Figure 9 and Figure 10 35

APPENDIX C: Equivalence of Figure 2 and Figure 4 53

TABLES

1 Truth Table for 1-Bit Full Adder 3

2 Logical Functions of Two Variables 7

3 Signatures for Hypothetical Function 20

4 4-Bit ALU Suspects Sets 23

FIGURES

1 Schematic of 1-Bit Adder 3

2 Schematic of 2-Bit Adder, Gate-Level View 4

3 Schematic of 2-Bit Adder, Structural View 4

4 Schematic of 2-Bit Adder, `exor/and` Implementation 5

5 Schematic of 1-Bit Adder, Functional Form 8

6 Schematic of `exor/and` Canonical Forms for 2-Bit Adder 10

7 Schematic of Binary Decision Tree for `cout` 12

8 Schematic of Binary Decision Diagram for `cout` for Input Order of `a0 < b0 < a1 < b1 < cin` 14

9 Schematic of `exor/and` for `cout` for Input Order of `a0 < b0 < a1 < b1 < cin` 15

10 Schematic of `exor/and` for `cout` for Input Order of `i0 < i1 < i2 < i3 < i4` 16

11 Schematic of Binary Decision Diagram for `cout` for Input Order of `a0 < b0 < cin < a1 < b1` 17

ACKNOWLEDGMENTS

We wish to offer our thanks to the following individuals for their help: Luther Martin, for bringing an interesting problem to our attention and providing guidance in our research and development efforts; Ken Dritz, Bob Veroff, and Anthony Wojcik, for providing insight and useful criticism; the participants in a brainstorming and review meeting held in Colorado Springs — David Castañon, Joanne DeGroat, Dick Kemmerer, and Brian Smith — for suggesting the idea of applying test vectors; and Travis Doom, for suggesting the use of suspect sets in refining the application of test vectors.

ASSIGNING FUNCTIONAL MEANING TO DIGITAL CIRCUITS

by

S.T. Eckmann and G.H. Chisholm

ABSTRACT

During computer-aided design, the problem of how to determine the logical function of a digital circuit arises in many contexts. For example, assigning functional meaning to a circuit is a fundamental operation in both reverse engineering and implementation validation. This report describes such a determination by discussing how a higher-level functional representation is constructed from a detailed circuit description (i.e., a gate-level netlist, which is a list of logic gates and their interconnections). The approach used involves transforming parts of the netlist into a functional representation and then manipulating this representation. Two types of functional representations are described: (1) a mathematical representation based on the logical operators "`exor`" and "`and`" and (2) a directed acyclic graph representation based on binary decision trees. Each representation provides a canonical form of the logical function being implemented (i.e., a form that is independent of implementation details). Such forms, however, have a well-known problem associated with the ordering of inputs: for each order, a unique form exists. A solution to this problem is given for both representations. Experimental results that demonstrate the use of these representations in the process of assigning functional meaning to a circuit are provided. The report also identifies and discusses issues critical to the performance required of this fundamental operation.

1 REPORT OBJECTIVE, SCOPE, AND ORGANIZATION

The primary objective of this report is to identify promising directions for near-term efforts in assigning functional meaning to digital circuits. The scope of the report includes techniques, issues, and possible solutions. The broader topics of reverse engineering, implementation verification, and functional matching are beyond the scope of this report. Information presented in Section 2 is the basis for the discussion and supports the recommendations. Section 3 illustrates two candidate representations, and Section 4 presents a high-level algorithm for reverse engineering that is based on these forms. Section 5 illustrates and discusses experimental results from using the two representations discussed in Section 3. Finally, Section 6 offers some recommendations. Throughout this document, logical expressions and formulas, computer code segments, and other technical terms appear in a different font from that used for the rest of the text.

2 BACKGROUND

2.1 REVERSE ENGINEERING

Reverse engineering is the construction of a higher-level functional representation of an implementation, designed to facilitate understanding of a system. Such a representation is developed by partitioning a detailed circuit description and assigning functional meaning to elements of that partition.

2.2 REVERSE ENGINEERING ASSISTANT

The Reverse Engineering Assistant (REA) is a tool set being researched and developed at Argonne National Laboratory (ANL) to help analysts in the reverse engineering of digital circuits. We expect the inputs to be (1) a gate-level netlist[1] representing some circuit of interest and (2) one or more component "libraries" that contain circuits representing components that might be embedded in the given circuit. The objective is to find embedded components.

A fundamental operation of the REA will be to determine the logical function of a digital circuit. In particular, the REA must provide an efficient means of determining whether some subnet of a given netlist implements some function in the cell library.

2.3 EXAMPLE REVERSE ENGINEERING PROBLEM

Consider a simple instance of a reverse engineering problem. We are given a library containing the function "1-bit full adder." Table 1 is a truth table for this function, and Figure 1 depicts a `nand`-gate implementation of this function. We are also given the circuit depicted in Figure 2, a 2-bit full-adder built from two of the 1-bit full adders in Figure 1. The signal labeled `s8` connects the "carry-out" bit of the first 1-bit adder to the "carry-in" bit of the second. We would like the REA to determine the functionality of the two subcircuits (i.e., identify a pair of embedded 1-bit full adders) and yield the circuit shown in Figure 3.[2]

[1] A netlist is a list of nodes and interconnections (signals or nets) between them. A gate-level netlist is a netlist in which each node is a logic gate.

[2] In more general terms, we want the REA to find all instances of all library components and return to the analyst a "reduced" (or abstracted) circuit, in which all matched gates have been replaced by the components of which they are constituents.

TABLE 1 Truth Table for 1-Bit Full Adder

Input			Output	
a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

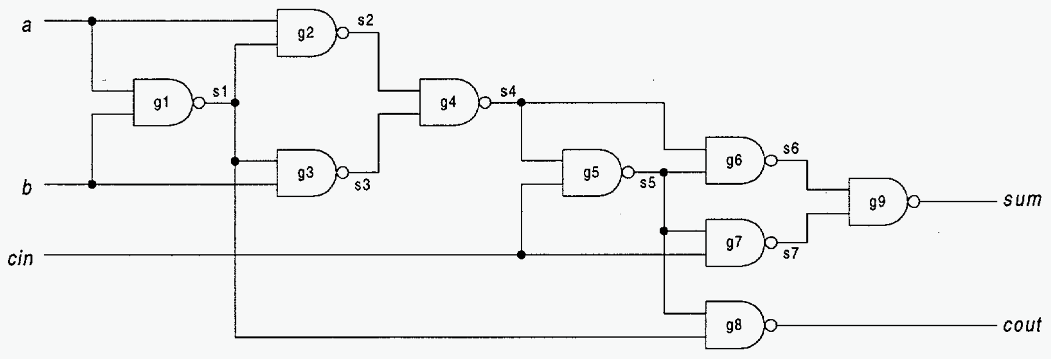

FIGURE 1 Schematic of 1-Bit Adder

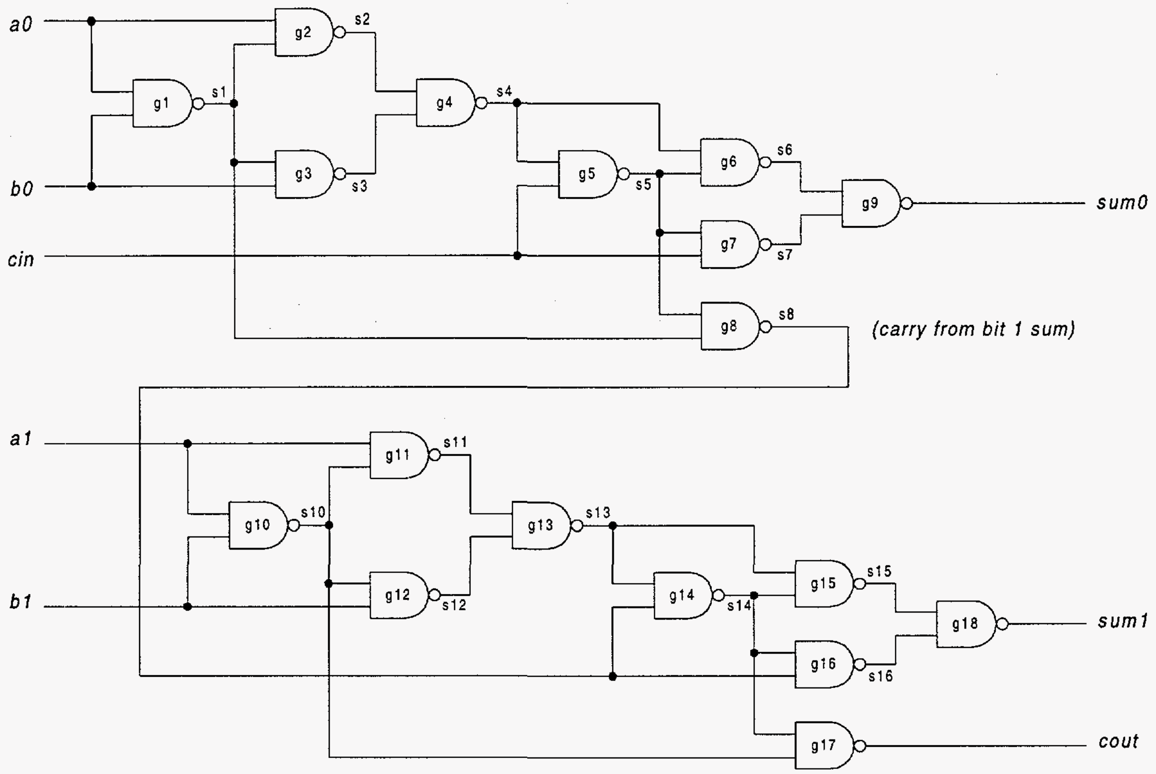

FIGURE 2 Schematic of 2-Bit Adder, Gate-Level View

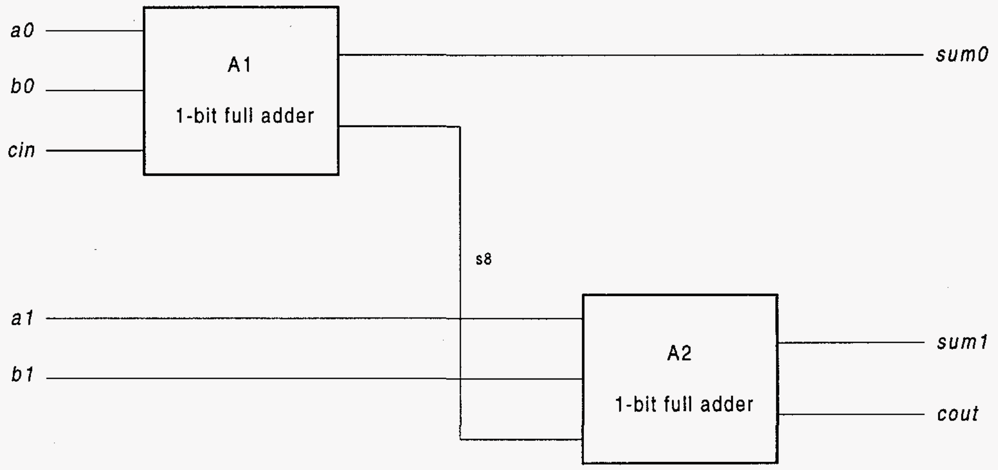

FIGURE 3 Schematic of 2-Bit Adder, Structural View

Perhaps the most obvious matching technique is to compare the library component and the candidate subnet structurally. Structural matching is inherently syntax matching. For example, the REA includes an implementation of the SubGemini subgraph isomorphism algorithm (Ohlrich et al. 1993) as its structural matching tool. However, any logic function may be realized in silicon in numerous ways. For example, Figure 4 shows another implementation of the 2-bit full adder; it has `exor` (exclusive-or) and `and` gates. This 2-bit adder is functionally equivalent to the circuit shown in Figure 2 but not structurally equivalent.

Circuits may include both unusual implementations or intentionally obfuscated implementations. This potential diversity of implementations limits the applicability of structural matching because a component library cannot include all possible implementations. The REA needs a more general, semantic technique to handle circuits for which syntactic matching fails. In this report, "semantic" matching is referred to as "functional" matching to emphasize that the intent is to compare circuit functions instead of circuit structures.

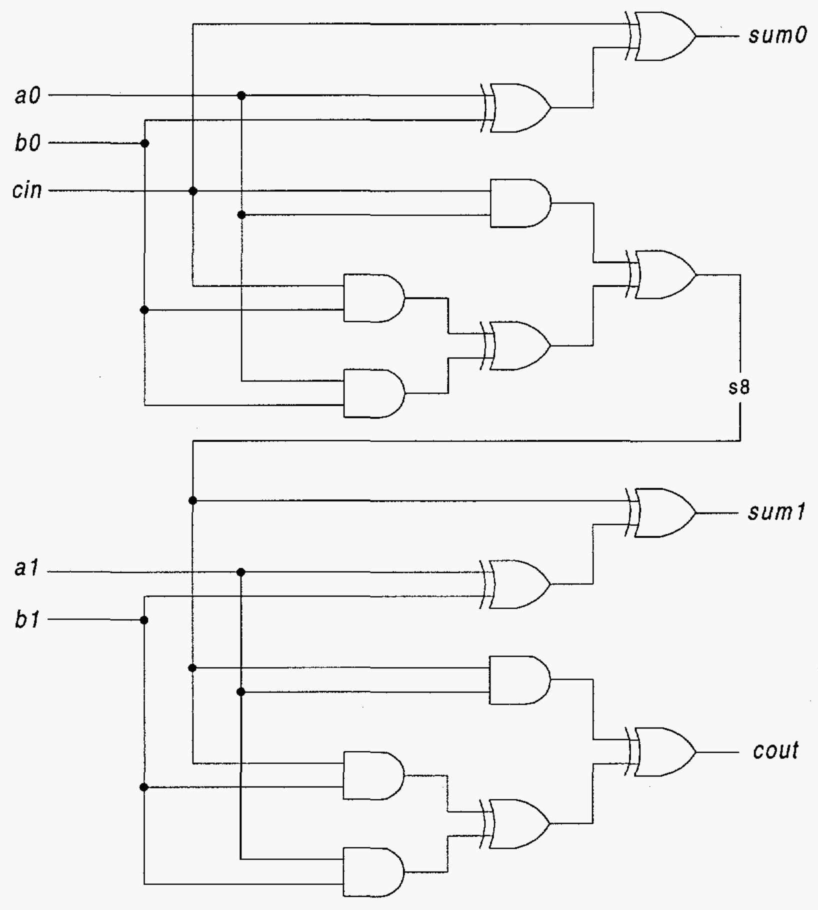

FIGURE 4 Schematic of 2-Bit Adder, `exor/and` Implementation

3 CANONICAL REPRESENTATIONS

One approach to functional matching is to convert circuits into logical formulas, transform the formulas into canonical form, and compare these forms with those in a library (all library elements were previously converted to this canonical form). In this context, a canonical form is one to which all equivalent forms can be transformed via a well-defined algorithm, such that any two instances of a given function will be identical. In other words, we are guaranteed that functionally identical circuits are mapped to equivalent logical formulas. Given a canonical form, functional matching becomes a simple two-step procedure: (1) convert the subject circuits (e.g., subnet and cell) to canonical form and (2) test the resulting formulas for equality.

3.1 REPRESENTING CIRCUITS AS LOGICAL FORMULAS

Shannon (1938) first postulated a mathematical theory of switching circuits based on a two-valued Boolean algebra. Thus, we can derive the behavior of digital circuits from a gate-level description and express this behavior as logical formulas in propositional logic. The set of switching functions used to represent a 2-bit switching function will have 2^2 possible combinations of these variables and 2^{2n} different switching functions (Table 2).

Any logical function can be expressed in terms of sets of operations (e.g., $\{\wedge, \vee, \neg\}$), and there are several canonical and functionally complete sets of operations. However, our selection of a set of operations for our application will be constrained by our use of an automated theorem-proving program, OTTER (Organized Techniques for Theorem Proving and Effective Research). Such a program provides an environment for rapidly constructing and executing experiments on representations of digital circuits. Specifically, we will be using rewrite rules to simplify and canonicalize expressions that represent digital circuits. The set of rules must be constructed such that:

- The order of applying the rules does not affect the final outcome and
- The application of the rules ends with a unique expression.

If we admit simplification rules within our set of rules, we will be restricted to the use of the $\{\oplus, \wedge\}$ set of operations for expression of logical functions. This restriction results from the fact that a finite, complete set of rewrite rules exists for the canonicalization and simplification of expressions constructed with the $\{\oplus, \wedge\}$ set of operations.

TABLE 2 Logical Functions of Two Variables[a]

Operation	A B 0 0	A B 0 1	A B 1 0	A B 1 1
Logical 0	0	0	0	0
$A \wedge B$	0	0	0	1
$A \wedge \neg B$	0	0	1	0
A	0	0	1	1
$\neg A \wedge B$	0	1	0	0
B	0	1	0	1
$A \oplus B$	0	1	1	0
$A \vee B$	0	1	1	1
$\neg(A \vee B)$	1	0	0	0
$A \leftrightarrow B$	1	0	0	1
$\neg B$	1	0	1	0
$B \rightarrow A$	1	0	1	1
$\neg A$	1	1	0	0
$A \rightarrow B$	1	1	0	1
$\neg(A \wedge B)$	1	1	1	0
Logical 1	1	1	1	1

[a] $\neg$ = not; $\wedge$ = and; $\oplus$ = exclusive or; $\rightarrow$ = implication; and $\vee$ = or.

3.2 `exor/and` CANONICAL FORM

Logical formulas are readily represented as expressions. For example, the 1-bit full adder depicted in Figure 1 may be represented as shown in Figure 5. In other words, the circuit's two outputs are defined in terms of the logical functions of its inputs. Figure 5 illustrates an expression of the 1-bit adder in a logic framework.

OTTER is a resolution-style automated reasoning program that operates on statements (called clauses) written in first-order logic with equality (McCune 1994; Wos et al. 1992). OTTER uses inference for deducing new clauses and demodulators for simplifying and rewriting clauses. OTTER is used here to illustrate the `exor/and` canonical form.

In OTTER notation, the canonicalizing set of rewrite rules for `{exor, and}` is as follows:

```
exor(0,x)=x.
exor(x,0)=x.
exor(x,x)=0.
```

```
cout = nand2(nand2(nand2(nand2(a,nand2(a,b)),
                                        nand2(nand2(a,b),b)),cin),
                    nand2(a,b))).

sum = nand2(nand2(nand2(nand2(a,nand2(a,b)),nand2(nand2(a,b),b)),
                          nand2(nand2(nand2(a,nand2(a,b)),
                                        nand2(nand2(a,b),b)),
                                cin)),
                   nand2(nand2(nand2(nand2(a,nand2(a,b)),
                                        nand2(nand2(a,b),b)),
                                cin),
                          cin))).
```

FIGURE 5 Schematic of 1-Bit Adder, Functional Form

```
exor(x,exor(x,y))=y.
exor(x,y)=exor(y,x).
exor(y,exor(x,z))=exor(x,exor(y,z)).
and(0,x)=0.
and(x,0)=0.
and(1,x)=x.
and(x,1)=x.
and(x,x)=x.
and(x,and(x,y))=and(x,y).
and(x,y)=and(y,x).
and(y,and(x,z))=and(x,and(y,z)).

and(x,exor(y,z))=exor(and(x,y),and(x,z)).
```

The variables x, y, and z represent arbitrary logical formulas. The logical values false and true are represented by 0 and 1, respectively. The first six rules apply to formulas that contain only exor, the next eight apply to formulas that contain only and, and the last one applies to formulas that contain both. For example, the first rule says any formula of the form exor(0, x) should be replaced by the formula represented by the variable x.

This set of rewrite rules and a procedure for applying them are all that is needed to canonicalize formulas that contain only exor and and. To canonicalize formulas that contain other logical operators, such as or, nand, etc., we must first translate such formulas into exor/and form, then apply the canonicalization rewrite rules. The translation can be done with another set of rewrite rules. For example, the following set of rules translates all operations in the 8-bit arithmetic logic unit (ALU) implementations with which we have been experimenting:

```
buf1(x) = x.

  inv(x) = exor(x,1).

  and2(x1,x2) = and(x1,x2).
  and3(x1,x2,x3) = and(x1,and(x2,x3)).
  and4(x1,x2,x3,x4) = and(x1,and(x2,and(x3,x4))).
  and5(x1,x2,x3,x4,x5) = and(x1,and(x2,and(x3,and(x4,x5)))).
  and8(x1,x2,x3,x4,x5,x6,x7,x8) =
   and(and4(x1,x2,x3,x4),and4(x5,x6,x7,x8)).

  or2(x1,x2) = inv(and(inv(x1),inv(x2))).
  or3(x1,x2,x3) = inv(and3(inv(x1),inv(x2),inv(x3))).
  or4(x1,x2,x3,x4) = inv(and4(inv(x1),inv(x2),inv(x3),inv(x4))).

  exor2(x1,x2) = exor(x1,x2).

  xnor2(x1,x2) = inv(exor(x1,x2)).

  nor2(x1,x2) = and(inv(x1),inv(x2)).
  nor3(x1,x2,x3) = and3(inv(x1),inv(x2),inv(x3)).
  nor4(x1,x2,x3,x4) = and4(inv(x1),inv(x2),inv(x3),inv(x4)).
  nor8(x1,x2,x3,x4,x5,x6,x7,x8) =
   and8(inv(x1),inv(x2),inv(x3),inv(x4),

inv(x5),inv(x6),inv(x7),inv(x8)).

nand2(x1,x2) = exor(and(x1,x2),1).
  nand3(x1,x2,x3) = exor(and3(x1,x2,x3),1).
  nand4(x1,x2,x3,x4) = exor(and4(x1,x2,x3,x4),1).
  nand5(x1,x2,x3,x4,x5) = exor(and5(x1,x2,x3,x4,x5),1).
```

As an example, consider the canonicalization of a 2-bit adder. The first step in canonicalizing a circuit given as a netlist is to express the circuit's outputs in a functional form. The implementation in Figure 2 has three outputs, which, by composing gates, can be expressed functionally as follows:

```
sum0 =
 nand2(nand2(nand2(nand2(a0,nand2(a0,b0)),nand2(nand2(a0,b0),b0)),
                  nand2(nand2(nand2(a0,nand2(a0,b0)),
                              nand2(nand2(a0,b0),b0)),
                        cin)),
            nand2(nand2(nand2(nand2(a0,nand2(a0,b0)),
                              nand2(nand2(a0,b0),b0)),
                        cin),
                  cin))).

sum1 =
 nand2(nand2(nand2(nand2(a1,nand2(a1,b1)),nand2(nand2(a1,b1),b1)),
                  nand2(nand2(nand2(a1,nand2(a1,b1)),
                              nand2(nand2(a1,b1),b1)),
                        s8)),
                  nand2(nand2(nand2(nand2(a1,nand2(a1,b1)),
                                   nand2(nand2(a1,b1),b1)),
                              s8),
                     s8))).

    cout = nand2(nand2(nand2(nand2(a1,nand2(a1,b1)),
                              nand2(nand2(a1,b1),b1)),s8),
                  nand2(a1,b1))).
```

where

```
    s8 = nand2(nand2(nand2(nand2(a0,nand2(a0,b0)),
                             nand2(nand2(a0,b0),b0)),cin),
                 nand2(a0,b0))).
```

The adder's three output ports – `sum0`, `sum1`, and `cout` – are defined in terms of logical functions on its input ports – `a0`, `b0`, `a1`, `b1`, and `cin`. Signal `s8` connects the carry-out bit of the first 1-bit adder to the carry-in bit of the second. It is used in the formulas as a shorthand.

When the adder's inputs are ordered `a0 < b0 < a1 < b1 < cin`, the resulting `exor/and` canonical forms for the three outputs are as shown in Figure 6.

The OTTER ATP program was selected as a tool to manipulate the derived logical formulas. One consequence of this selection is the necessity of maintaining a canonical form of

```
    sum0 = exor(a0,exor(b0,cin))).

    sum1 =
     exor(a1,exor(b1,exor(and(a0,b0),exor(and(a0,cin),and(b0,cin)))))).

    cout = exor(and(a0,and(b0,a1)),
                     exor(and(a0,and(b0,b1)),
                         exor(and(a0,and(a1,cin)),
                             exor(and(a0,and(b1,cin)),
                                 exor(and(b0,and(a1,cin)),
                                     exor(and(b0,and(b1,cin)),
                                         and(a1,b1)))))))).
```

FIGURE 6 Schematic of `exor/and` Canonical Forms for 2-Bit Adder

these formulas during processing by the software. Specifically, we must ensure that any arbitrary expression used as input produces a canonical form.[3]

The above results were obtained by using OTTER and the rewrite rules presented above. Details are provided in Appendix A. Because `exor/and` is a canonical form, any other implementation of these three output functions will reduce to the same three formulas, if the same order is used for the adder inputs. Section 4.1 discusses the dependence of canonical forms on input order. A canonical representation supports one's intuition about the functionality of the circuit but removes any structural information that may be applied in the discovery of subcircuit functionality. For example, the structurally different 2-bit adders depicted in Figures 2 and 4 implement the functionality of a full 2-bit adder.

3.3 BINARY DECISION DIAGRAMS

Binary decision diagrams (BDDs) were introduced and proven to be a canonical form in Bryant (1986). Bryant (1992) is a good survey. Three basic definitions from Brace et al. (1990) are provided here:

- *Binary decision diagram* is a directed acyclic graph (DAG). The graph has two sink nodes labeled `0` and `1` representing the Boolean functions `0` and `1`. Each nonsink node is labeled with a Boolean variable `v` and has two out-edges labeled `1` (or `then`) and `0` (or `else`). Each nonsink node represents the Boolean function corresponding to its `1` edge if `v = 1` or to its `0` edge if `v = 0`.

- *Ordered binary decision diagram* (OBDD) is a BDD with the constraint that the input variables are ordered and that every source-to-sink path in the OBDD "visits" the input variables in ascending order.

[3] Here is a more precise discussion of the OTTER constraints. Given a set of expressions (e.g., all well-former formulas restricted to operators {`exor, and`}, variable symbols, and propositional symbols), consider an equivalence relation (in this case, logical equivalence) that partitions the set of formulas.

In the broadest sense, a canonical form is any member of a subset distinguished from the original set of expressions. Each of these distinguished expressions represents the equivalence class to which it belongs. (Note all commutative and associative variants of a formula in conjunctive normal form [CNF] are canonical forms; for example, {`and, or, not`} expressions with no notion of lexical order.)

We strive for:

- A unique form such that the equivalence class has exactly one member in the unique form and
- A finite procedure P that takes an arbitrary expression E as input and produces its unique form P(E) as output.

Then two expressions, A and B, are equivalent if, and only if, P(A) is identical to P(B).

- *Reduced ordered binary decision diagram* (ROBDD) is an OBDD in which each node represents a distinct logic function.

Many digital design tools use BDDs internally (e.g., see *Abstract Hardware Limited* [1996]). Most applications of BDDs use ROBDDs exclusively. Because our reverse engineering application will use only ROBDDs, too, BDD as used in the rest of this report means ROBDD.

Consider as an example a construction of the BDD for the 2-bit adder's `cout` output port. Again we start with this function:

```
cout = nand2(nand2(nand2(nand2(a1,nand2(a1,b1)),
                         nand2(nand2(a1,b1),b1)),s8),
             nand2(a1,b1))).
```

where

```
s8 = nand2(nand2(nand2(nand2(a0,nand2(a0,b0)),
                        nand2(nand2(a0,b0),b0)),cin),
            nand2(a0,b0))).
```

Also, again we use the input order `a0` < `b0` < `a1` < `b1` < `cin`. Conceptually, a BDD can be built by starting with the complete binary decision tree for the function of interest. For the cout function, the tree is shown in Figure 7.

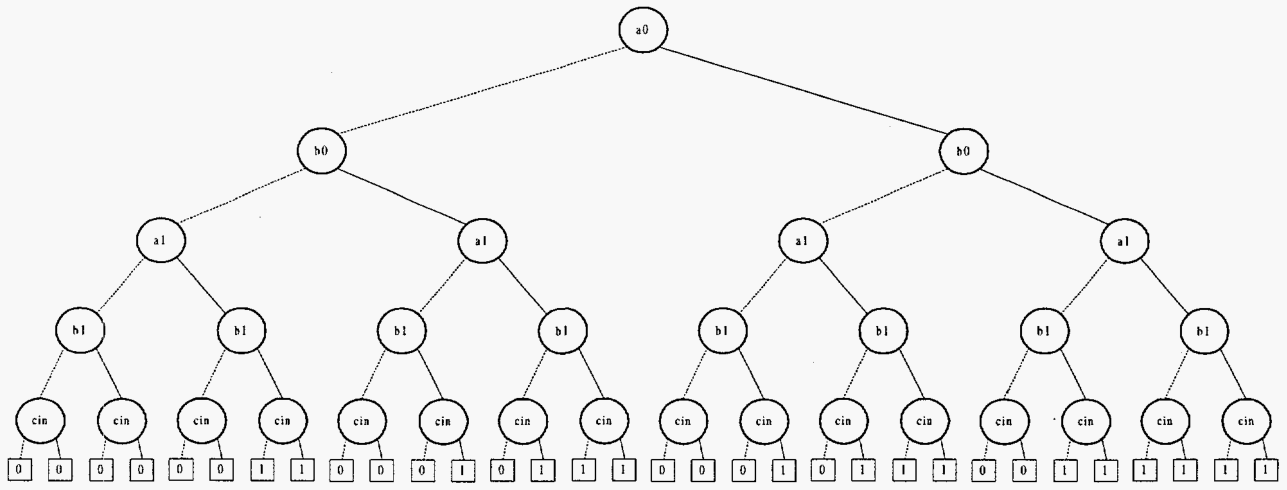

FIGURE 7 Schematic of Binary Decision Tree for `cout`

Each left out-edge is implicitly labeled `0`, and each right out-edge is implicitly labeled `1`. This tree is almost an OBDD. All that needs to be done is to replace the multiple leaf nodes with a single node labeled `1` and another labeled `0`, and to redirect all edges appropriately; the resulting DAG will be an OBDD. The next step in a manual construction is to eliminate redundancy, yielding an ROBDD. For example, the five `cin` nodes with both out-edges going to `0` could be collapsed into a single node. There is a simple recursive procedure for reducing the OBDD into an ROBDD that is guaranteed to yield a canonical form. For the `cout` output of the 2-bit full adder, the BDD is shown in Figure 8 in graphical form.

BDDs are used extensively in the digital design community because (1) they are typically a much more compact representation than any logical canonical form (e.g., `exor/and`) and (2) very efficient implementations are available. The three BDD packages we evaluated, which were from Carnegie-Mellon University and the University of California (Berkeley and Santa Barbara), are based on the implementation described in Brace et al. (1990). Key features include (1) hash table implementations, which typically provide linear time construction (the intermediate steps in the manual construction described above are eliminated) and constant time comparison operations; (2) libraries of operations for building and manipulating BDDs; and (3) access to BDDs only through a BDD manager that hides implementation details (hash tables, etc.) from the application.

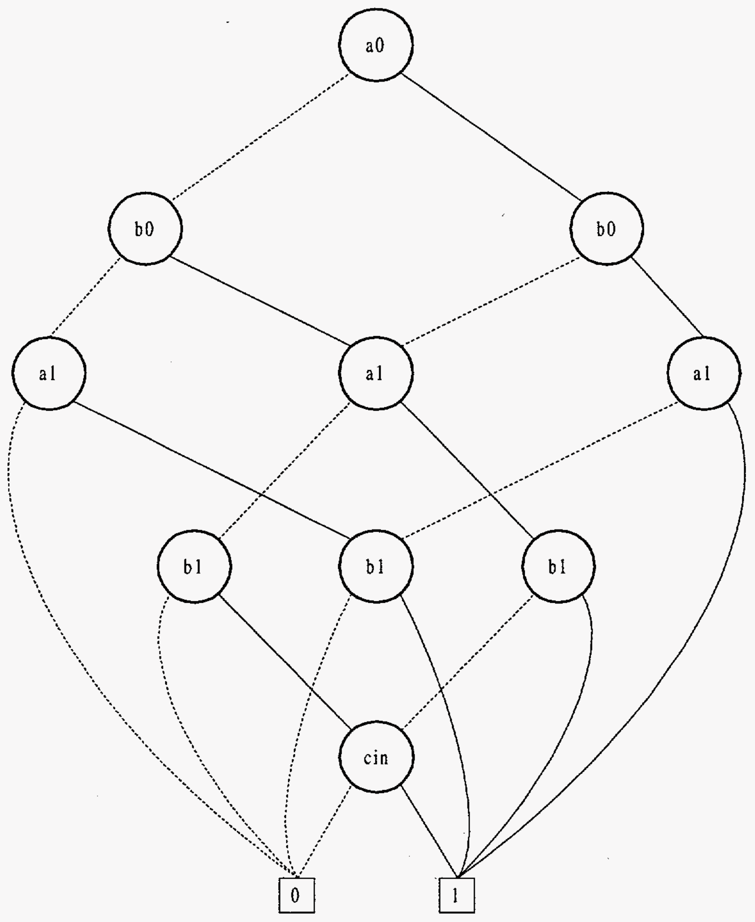

FIGURE 8 Schematic of Binary Decision Diagram for `cout` for Input Order of `a0 < b0 < a1 < b1 < cin`

4 CANONICALIZATION ISSUES

4.1 INPUT ORDER FOR exor/and REPRESENTATION

Since both the `exor/and` form and BDD form are known to be canonical, it is not surprising that all implementations of the 2-bit adder were canonicalized to the same form in our experiments. What may be surprising is that this is true only for a given order of input signal names.

Recall that our goal is to demonstrate functional equivalence between an unknown circuit and a library circuit. The netlist for the unknown circuit provides no information about how to order the input signals; i.e., input signals are assigned names that may result in their random ordering. Figure 9 depicts the cout function for a 2-bit full adder with the input signals of `a0 < b0 < a1 < b1 < cin`. This represents a library circuit for which we have full knowledge about input signal names and order. Figure 10 depicts an unknown circuit with input signals of `i0 < i1 < i2 < i3 < i4`. The expressions for both circuits (Figures 9 and 10) are in canonical forms with respect to the order of their input signals. However, simply using rewrite rules and demodulation fails to demonstrate that the two circuits are equivalent.

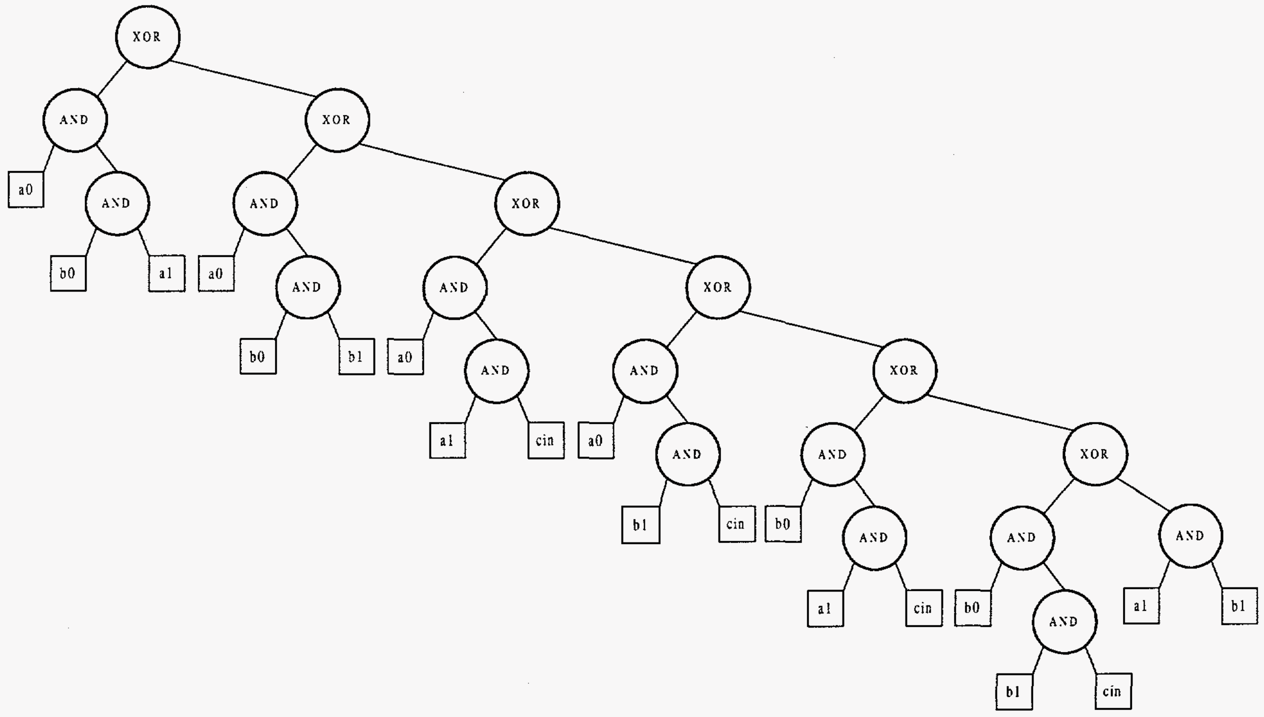

FIGURE 9 Schematic of `exor/and` for `cout` for Input Order of `a0 < b0 < a1 < b1 < cin`

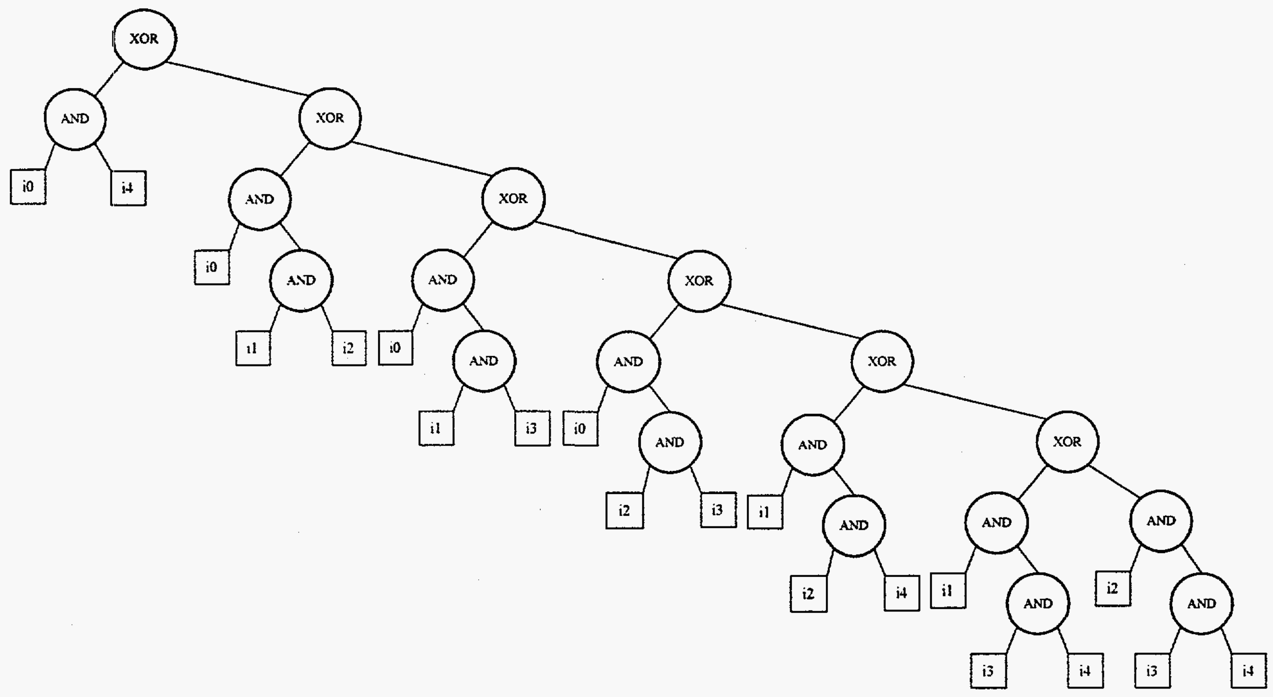

FIGURE 10 Schematic of `exor/and` for `cout` for Input Order of `i0 < i1 < i2 < i3 < i4`

Canonical forms are defined with respect to certain parameters; if you change the parameters, a different canonical form results. When OTTER is used to transform and canonicalize expressions, the set of rewrite rules may include rules that depend on lexicographic order. Such rules exist in the set being applied in the following example:

```
exor(x,y)=exor(y,x).                      % Commutativity of exor.
and(x,y)=and(y,x).                        % Commutativity of and.
exor(y,exor(x,z))=exor(x,exor(y,z)).      % Rotation of exor.
and(y,and(x,z))=and(x,and(y,z)).          % Rotation of and.
```

These rules do not change the syntactical form of the terms to which they apply, and they raise the possibility of endless application/reapplication to the same terms (i.e., nontermination). To guarantee termination, OTTER imposes a lexical order on constants and function symbols. The lexical order of constants (used to name input signals) and function symbols (e.g., `exor` and `and`) can be specified in a "lex" list. This dependence on the order of input signal names in OTTER's lex list poses a problem for trying to demonstrate the equivalence between the functionally identical circuits depicted in Figures 9 and 10. Appendix B includes OTTER data sets that were used to demonstrate such equivalence. The approach used for these demonstrations is independent of OTTER's lex list and shows how the equivalence between a circuit with unknown inputs and a library circuit was determined.

The essence of this approach is to consider the library circuit as a specification of a function. In an OTTER representation of a specification, variables represent inputs. The unknown circuit is represented by constants and function symbols, as it was before, except that we represent the circuit in netlist form. In the previous examples, OTTER applied rewrite rules to manipulate the circuit description. For this example, we use different inference rules (i.e., hyperresolution and paramodulation; see Wos et al. 1992).

4.2 INPUT ORDER FOR BINARY DECISION DIAGRAM REPRESENTATION

The BDDs for those two input orders also differ, but a better illustration results from comparing Figure 8 with Figure 11, which shows the BDD for the 2-bit adder's `cout` output with input order of `a0, b0, cin, a1, b1.`

In most contexts in which canonical forms are used to compare functions, the variables (inputs) are known for both circuits, so variable ordering is of minor importance. In the context of reverse engineering, however, we do not know the inputs for one of the functions; when we

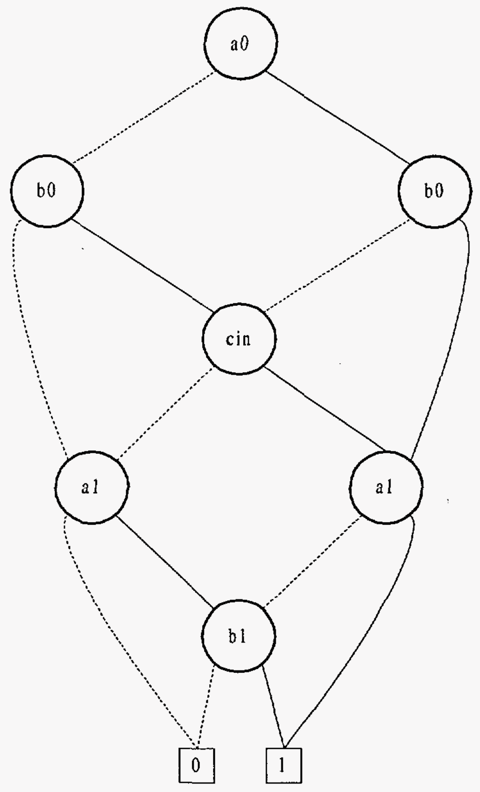

FIGURE 11 Schematic of Binary Decision Diagram for `cout` for Input Order of `a0 < b0 < cin < a1 < b1`

partition a netlist, we get a set of inputs, but no clues about what they might represent. Matching input order is therefore a problem unique to this application. There are no algorithms for choosing identical orders; it must be done heuristically. Therefore, for each candidate subcircuit that we wish to test, we may have to enumerate all possible input orders. A circuit with `m` inputs has `m!` possible input orders. Exhaustive enumeration of input permutations is therefore not feasible for subcircuits with more than about 10 distinct inputs. We can generalize from the `exor/and` and BDD examples that all canonical forms suffer from this sensitivity to input order. The rest of this section discusses how the input order sensitivity problem is handled by our two chosen canonical forms.

Appendix B and Section C.2 of Appendix C illustrate an approach to using OTTER to determine the functionality of a circuit, irrespective of the signal names assigned to the inputs. This approach is based on describing the function (e.g., a 2-bit adder) as an expression for which the input signals are represented by variables. The circuit under study is input as a netlist. OTTER manipulates these inputs and determines whether the netlist is an instance of the specification. (See the appendixes for details.)

We are investigating signature testing as being the most promising method for avoiding the enumeration of all input permutations for BDDs. The idea is to apply input vectors to a candidate subcircuit and check whether its output signature matches that of a library component. If not, the candidate cannot possibly be an instance of the library component. Only when a candidate passes the signature tests is it considered a candidate for detailed (i.e., input permutation) testing.

The specific input test vectors with which we have experimented for signature testing are constant vectors and unit vectors:

- `c1` – all `1` bits,
- `c0` – all `0` bits,
- `u1` – single `1` bit (any single input; all other inputs are `0`), and
- `u0` – single `0` bit (any single input; all other inputs are `1`).

The signature function associated with these input vectors is the number of `1` bits in the set of library module outputs. A candidate subcircuit is tested for feasibility as follows:

1. If the candidate signature for the c_1 test vector $\neq$ signature(c_1), then it fails.

2. If the candidate signature for the c_0 test vector $\neq$ signature(c_0), then it fails.

3. For each of the n u_1 test vectors v_i (for a circuit with n inputs), if none of the u_1 test vectors applied to the candidate has a signature of signature(v_i), then it fails.

4. For each of the n u_0 test vectors v_i (for a circuit with n inputs), if none of the u_0 test vectors applied to the candidate has a signature of signature(v_i), then it fails.

A candidate subcircuit that passes these steps is a feasible match, but this algorithm does not eliminate the potential need to check all input permutations to determine whether a feasible match is an actual match. A refinement of steps 3 and 4 keeps track of which candidate inputs could possibly match which module inputs. The sets suspects(i), one for each of the library module's inputs, identify candidate inputs that could possibly match input i. The sets nonsuspects(i) identify candidate inputs that cannot possibly match input i and will therefore be excluded from subsequent tests of input i. Both sets are initialized to Empty. The set of input permutations that must be tested is built from the suspects sets instead of an exhaustive enumeration of all permutations of the candidate's inputs. The refined steps 3 and 4 look like this:

```
3. For each of the n  $u_1$ test vectors $v_i$  (for a circuit with n inputs):
       Initialize suspects(i) and nonsuspects(i) to Empty
       For each candidate input j  {
           If candidate signature for $v_j$  matches signature($v_i$)  then
               add j to suspects(i)
           Else
               add j to nonsuspects(i)
       }
       If suspects(i) is Empty then fail

4. For each of the n  $u_0$ test vectors  $v_i$ (for a circuit with n inputs):
       For each candidate input j {
           If j is already in nonsuspects(i) then
               skip j
           Else if candidate signature for $v_j$ does not match signature($v_i$)  then
               add j to nonsuspects(i), and remove j from suspects(i)
       }
       If suspects(i) is Empty then fail
```

As the inner loop in step 3 is repeatedly executed, each candidate input j is added either to suspects(i) or to nonsuspects(i). At the end of step 3 (and throughout step 4), it must be the case that, for each module input i, suspects(i) $\cup$ nonsuspects(i) = { candidate inputs }, and also suspects(i) and nonsuspects(i) are disjoint. If at the end of step 3 each set suspects(i) contains at least one element (meaning there is at least one candidate input that might match module input i), then step 4 is performed. The inner loop for step 4 differs from that for step 3 because the suspects and nonsuspects sets have now been populated.

As an example, consider application of the refined signature testing algorithm to the hypothetical function partially described by Table 3, which lists input vectors, outputs, and signatures (but is not a complete truth table).

Suppose we are considering a candidate subcircuit that is an instance of this function. All of the signature tests obviously will succeed, in the sense that the candidate will be identified as a feasible match. The interesting part of the algorithm will thus be the construction of suspects sets. Further suppose that the candidate's inputs are named `s1, s2, s3,` and `s4` and that the correct mapping between module inputs and candidate inputs is as follows:

$$[h \rightarrow s2,\ i \rightarrow s3,\ j \rightarrow s1,\ k \rightarrow s4].$$

The first iteration of the outer loop of step 3 sets `h = 1`, and `i = j = k = 0`. The signature for this test vector, per the appropriate row in Table 3, is 1. The inner loop of step 3 tests with each of the four u_1 input vectors applied to the candidate's inputs. The first sets `s1 = 1`, and `s2 = s3 = s4 = 0`. This corresponds to `j = 1`, and `h = i = k = 0` (based on the known name mapping), which yields a signature of `2` (the implementation actually computes the signature from the candidate's output BDDs, since it does not know the correct mapping between module inputs and candidate inputs). Therefore `s1` cannot match `h`, so we add `s1` to `nonsuspects(h)`.

TABLE 3 Signatures for Hypothetical Function

	Inputs				Outputs			
Type	**h**	**i**	**j**	**k**	**p**	**q**	**r**	**Sig**
c_1	1	1	1	1	1	0	1	2
c_0	0	0	0	0	0	1	0	1
	1	0	0	0	1	0	0	1
u_1	0	1	0	0	0	0	1	1
	0	0	1	0	1	1	0	2
	0	0	0	1	0	1	1	2
	0	1	1	1	0	0	0	0
u_0	1	0	1	1	1	1	1	3
	1	1	0	1	0	0	1	1
	1	1	1	0	0	0	1	1

The second iteration of the inner loop of step 3 sets s2 = 1, and s1 = s3 = s4 = 0. This corresponds to h = 1, and i = j = k = 0, which has a signature of 1. Therefore s2 could match h, so we add s2 to suspects(h). After the third and fourth iterations of the inner loop, we will have:

```
suspects(h)     =  {s2, s3} and
nonsuspects(h)  =  {s1, s4}.
```

The other three iterations of the outer loop of step 3 are similar, and after step 3 finishes (testing with u_1-type vectors), the suspects and nonsuspects sets will be:

```
suspects(h)     =  {s2, s3}
nonsuspects(h)  =  {s1, s4}
suspects(i)     =  {s2, s3}
nonsuspects(i)  =  {s1, s4}
suspects(j)     =  {s1, s4}
nonsuspects(j)  =  {s2, s3}
suspects(k)     =  {s1, s4}, and
nonsuspects(k)  =  {s2, s3}
```

which says that inputs h and i are indistinguishable on the basis of u_1-type vectors, but they are different from j and k. Similarly, j and k are indistinguishable on the basis of u_1-type vectors, but they are different from h and i.

The first iteration of the outer loop of step 4 sets h = 0, and i = j = k = 1. The signature for this test vector, per the appropriate row in Table 3, is 0. The inner loop of step 3 tests with each of the four u_1 input vectors applied to the candidate's inputs. The first iteration skips s1 because it is already in nonsuspects(h).

The second iteration of the inner loop of step 4 sets s2 = 0, and s1 = s3 = s4 = 1 (s2 is not in nonsuspects(h), so this iteration is not skipped). This corresponds to h = 0, and i = j = k = 1, which has a signature of 0. Therefore s2 is still a possible match for h, so nothing is done.

The third iteration of the inner loop of step 4 sets s3 = 0, and s1 = s2 = s4 = 1 (s3 is not in nonsuspects(h), so this iteration is not skipped). This corresponds to i = 0, and h = j = k = 1, which has a signature of 3. Therefore s3 is no longer a possible match for h, and it is moved from suspects(h) to nonsuspects(h).

The fourth iteration of the inner loop of step 4 is skipped because `s4` is already in `nonsuspects` (h). So we have:

```
suspects(h)      = {s2} and
nonsuspects(h)   = {s1, s3, s4}.
```

The other three iterations of the outer loop of step 4 are similar, and after step 4 finishes (testing with u_0-type vectors), the suspects sets will be:

```
suspects(h)  =  {s2},
suspects(i)  =  {s3},
suspects(j)  =  {s1, s4}, and
suspects(k)  =  {s1, s4}
```

because inputs `h` and `i` have different u_0 signatures, but inputs `j` and `k` still cannot be distinguished. Therefore, in this example, there are 4 (i.e., $1 \times 1 \times 2 \times 2$) permutations to try instead of 24 (4!). Although this smaller number is not a significant improvement, results to date do suggest that this signature testing scheme with the `suspects` sets refinement is very precise, in the sense that the `suspects` sets are typically much smaller than the entire set of inputs. For example, in a test with a 4-bit ALU, which has 14 inputs, the number of permutations to be examined was reduced from 87,178,291,200 (14!) to 8,640. The actual `suspects` sets for that test case are shown in Table 4. The `nonsuspects` sets `nonsuspects(i)` are just `{all candidate inputs}` minus `suspects(i)`. For example, since

```
{all candidate inputs} = {  S__3, S__2, S__0, B__7, B__6, B__5,
                            B__4, A__4, S__1, NET6, M, A__7, A__6,
                            A__5 }
```

and

```
suspects(S3) = { S__3 },
```

we can deduce that

```
nonsuspects(S3) = {      S__2, S__0, B__7, B__6, B__5, B__4,
                   A__4, S__1, NET6, M, A__7, A__6, A__5 }
```

TABLE 4 4-Bit ALU Suspects Sets

suspects[S3]	S__3					
suspects[S2]	S__2	S__0	B__7	B__6	B__5	B__4
suspects[S1]	S__1	NET6				
suspects[S0]	S__2	S__0	B__7	B__6	B__5	B__4
suspects[A3]	A__7	A__6	A__5			
suspects[A2]	A__7	A__6	A__5			
suspects[A1]	A__7	A__6	A__5			
suspects[A0]	A__4					
suspects[B3]	S__2	S__0	B__7	B__6	B__5	B__4
suspects[B2]	S__2	S__0	B__7	B__6	B__5	B__4
suspects[B1]	S__2	S__0	B__7	B__6	B__5	B__4
suspects[B0]	S__2	S__0	B__7	B__6	B__5	B__4
suspects[M]	M					
suspects[CN]	S__1	NET6				

Steps 3 and 4 apply two different "families" of test vectors. Additional families of test vectors (e.g., the family of vectors with two 1 bits) could be applied, the expectation being that each family will reduce the size of some `suspects` sets. Taken to the limit, this approach would effectively apply a complete truth table to the candidate's inputs to determine whether it matches the library module; however, the simplest two families (u_1 and u_0) may be sufficient. In the worst case, `suspects(i)` will still contain the entire set of candidate inputs for all module inputs `i`, but this should happen only for modules with completely symmetric inputs (e.g., a parity circuit).

It seems reasonable that the precision of signature testing will increase for components that have more inputs and outputs. This result is exactly what is needed: as the number of permutations increases factorially, we want greater assurance that only very likely candidates make it past that stage of comparison.

5 CONCLUSIONS AND RECOMMENDATIONS

The advantage of using a canonical form is that it gives us positive information relatively efficiently on whether two circuits perform the same logical function, because given an order for inputs, all implementations of a function will reduce to the same canonical form.

When the `exor/and` representation is manipulated by an automated reasoning program, a unique canonical form of all implementations of a function is obtained. This unique form implies that, irrespective of input order, the functional equivalence of two circuits is demonstrable. However, experimentation with the specification and implementation of a 4-bit ALU indicates that the `exor/and` representation suffers from a combinatorial "explosion" problem: The files associated with the higher order functions in the ALU are too large to process. For this reason, we decided to adopt the BDD representation as the canonical representation for the remainder of our investigations on assigning functional meaning to circuits.

Signature testing is a mechanism that eliminates the need to test all permutations when BDDs are being applied to demonstrate the functional equivalence of two circuits.

We are currently experimenting with both the `exor/and` and BDD canonical representations. At this time, it appears that structural matching is the best method for finding large components with standard implementations, while functional matching is best used for finding small components with nonstandard implementations.

6 BIBLIOGRAPHY

Brace, K.S., R.L. Rudell, and R.E. Bryant, 1990, "Efficient Implementation of a BDD Package," pp. 40–45 in *Proceedings of the 27th ACM/IEEE Design Automation Conference.*

Bryant, R.E., 1986, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers* C-35(6):677–691, August.

Bryant, R.E., 1992, *Symbolic Manipulation with Ordered Binary Decision Diagrams*, Report CMU-CS-92-160, URL: ftp://reports.adm.cs.cmu.edu/usr/anon/1992/CMU-CS-92-160.ps.

Knuth, D., and P. Bendix, 1970, "Simple Word Problems in Universal Algebras," pp. 263–297 in *Computational Problems in Abstract Algebras,* J. Leech (editor), Pergamon Press, Oxford, England.

McCune, W.W., 1994, *Otter 3.0 Reference Manual and Guide*, Report ANL-94/6, Argonne National Laboratory, Argonne, Ill., URL: ftp://info.mcs.anl.gov/pub/Otter/Papers/otter3_manual.ps.gz.

Mailhot, F., 1991, "Technology Mapping for VLSI Circuits Exploiting Boolean Properties and Operations," Ph.D. dissertation, Department of Electrical Engineering, Stanford University.

Ohlrich, M., C. Ebeling, E. Ginting, and L. Sather, 1993, "SubGemini: Identifying Subcircuits Using a Fast Subgraph is Omorphism Algorithm," pp. 31–37 in *Proceedings of the 30th ACM/IEEE Design Automation Conference.*

Shannon, C.E., 1938, "A Symbolic Analysis or Relay and Switching Circuits," *Transactions of the American Institute of Electrical Engineers* 57:713–723.

Wos, L., R. Overbeek, E. Lusk, and J. Boyle, 1992, *Automated Reasoning*, 2nd ed., McGraw-Hill.

APPENDIX A:

REWRITING 2-BIT ADDER OUTPUTS TO CANONICAL FORM

One way to obtain canonical forms is to use a “rewrite rule” system. The OTTER automated reasoning program excels at this sort of task. For example, consider canonicalizing the 2-bit adder. We start by separating the gates of the netlist into internal and output gates: output gates provide the circuit’s externally visible outputs; internal gates provide all the rest. The output gates of the 2-bit adder are therefore those that generate the carry-out bit and the two sum bits (i.e., `sum0` and `sum1`). The objective of this first step is to transform the netlist into a set of functional representations of the circuit’s outputs. In the following OTTER input file, internal signal `s8` is also specified as an output gate to match the presentation in Section 3:

```
set(demod_inf).
set(pretty_print).

   % internal gates
list(demodulators).
    EQUAL(s1,  nand(a0,b0)).
    EQUAL(s2,  nand(a0,s1)).
    EQUAL(s3,  nand(s1,b0)).
    EQUAL(s4,  nand(s2,s3)).
    EQUAL(s5,  nand(s4,cin)).
    EQUAL(s6,  nand(s4,s5)).
    EQUAL(s7,  nand(s5,cin)).
    EQUAL(s10, nand(a1,b1)).
    EQUAL(s11, nand(a1,s10)).
    EQUAL(s12, nand(s10,b1)).
    EQUAL(s13, nand(s11,s12)).
    EQUAL(s14, nand(s13,s8)).
    EQUAL(s15, nand(s13,s14)).
    EQUAL(s16, nand(s14,s8)).
end_of_list.

   % output gates
list(sos).
    OUTPUT(sum0, nand(s6,s7)).
    OUTPUT(s8,   nand(s5,s1)).
    OUTPUT(cout, nand(s14,s10)).
    OUTPUT(sum1, nand(s15,s16)).
end_of_list.
```

If we feed the above lines to OTTER, we get something like this:

```
----- Otter 3.0.4, August 1995 -----
The job was started by eckmann on eckmann.home, Tue Apr 22 19:58:44 1997
The command was "otter".

set(demod_inf).

list(demodulators).
1 [] EQUAL(s1,nand(a0,b0)).
2 [] EQUAL(s2,nand(a0,s1)).
3 [] EQUAL(s3,nand(s1,b0)).
4 [] EQUAL(s4,nand(s2,s3)).
5 [] EQUAL(s5,nand(s4,cin)).
6 [] EQUAL(s6,nand(s4,s5)).
7 [] EQUAL(s7,nand(s5,cin)).
8 [] EQUAL(s10,nand(a1,b1)).
9 [] EQUAL(s11,nand(a1,s10)).
10 [] EQUAL(s12,nand(s10,b1)).
11 [] EQUAL(s13,nand(s11,s12)).
12 [] EQUAL(s14,nand(s13,s8)).
13 [] EQUAL(s15,nand(s13,s14)).
14 [] EQUAL(s16,nand(s14,s8)).
end_of_list.

list(sos).
15 [] OUTPUT(sum0,nand(s6,s7)).
16 [] OUTPUT(s8,nand(s5,s1)).
17 [] OUTPUT(cout,nand(s14,s10)).
18 [] OUTPUT(sum1,nand(s15,s16)).
end_of_list.

======= end of input processing =======

=========== start of search ===========

given clause #1: (wt=5) 15 []
OUTPUT(sum0,nand(s6,s7)).
** KEPT (pick-wt=43): 19
[15,demod,6,4,2,1,3,1,5,4,2,1,3,1,7,5,4,2,1,3,1]
OUTPUT(
    sum0,
    nand(
        nand(
            nand(nand(a0,nand(a0,b0)),nand(nand(a0,b0),b0)),
            nand(nand(nand(a0,nand(a0,b0)),nand(nand(a0,b0),b0)),cin)),
        nand(nand(nand(nand(a0,nand(a0,b0)),
                         nand(nand(a0,b0),b0)),cin),cin))).

given clause #2: (wt=5) 16 []
OUTPUT(s8,nand(s5,s1)).
** KEPT (pick-wt=19): 20 [16,demod,5,4,2,1,3,1,1]
OUTPUT(
    s8,
    nand(nand(nand(nand(a0,nand(a0,b0)),nand(nand(a0,b0),b0)),cin),
         nand(a0,b0))
).

given clause #3: (wt=5) 17 []
```

```
OUTPUT(cout,nand(s14,s10)).
** KEPT (pick-wt=19): 21 [17,demod,12,11,9,8,10,8,8]
OUTPUT(
    cout,
    nand(nand(nand(nand(a1,nand(a1,b1)),nand(nand(a1,b1),b1)),s8),
         nand(a1,b1))
).

given clause #4: (wt=5) 18 []
OUTPUT(sum1,nand(s15,s16)).
** KEPT (pick-wt=43): 22
  [18,demod,13,11,9,8,10,8,12,11,9,8,10,8,14,12,11,9,8,10,8]
OUTPUT(
    sum1,
    nand(
        nand(
            nand(nand(a1,nand(a1,b1)),nand(nand(a1,b1),b1)),
            nand(nand(nand(a1,nand(a1,b1)),nand(nand(a1,b1),b1)),s8)),
        nand(nand(nand(nand(a1,nand(a1,b1)),
                      nand(nand(a1,b1),b1)),s8),s8))
).

given clause #5: (wt=19) 20 [16,demod,5,4,2,1,3,1,1]
OUTPUT(
    s8,
    nand(nand(nand(nand(a0,nand(a0,b0)),nand(nand(a0,b0),b0)),cin),
         nand(a0,b0))
).

given clause #6: (wt=19) 21 [17,demod,12,11,9,8,10,8,8]
OUTPUT(
    cout,
    nand(nand(nand(nand(a1,nand(a1,b1)),nand(nand(a1,b1),b1)),s8),
         nand(a1,b1))
).

given clause #7: (wt=43) 19
[15,demod,6,4,2,1,3,1,5,4,2,1,3,1,7,5,4,2,1,3,1]
OUTPUT(
    sum0,
    nand(
        nand(
            nand(nand(a0,nand(a0,b0)),nand(nand(a0,b0),b0)),
            nand(nand(nand(a0,nand(a0,b0)),nand(nand(a0,b0),b0)),cin)),
nand(nand(nand(nand(a0,nand(a0,b0)),
                      nand(nand(a0,b0),b0)),cin),cin))
).

given clause #8: (wt=43) 22
   [18,demod,13,11,9,8,10,8,12,11,9,8,10,8,14,12,11,9,8,10,8]
OUTPUT(
    sum1,
    nand(
        nand(
            nand(nand(a1,nand(a1,b1)),nand(nand(a1,b1),b1)),
            nand(nand(nand(a1,nand(a1,b1)),nand(nand(a1,b1),b1)),s8)),
        nand(nand(nand(nand(a1,nand(a1,b1)),
                      nand(nand(a1,b1),b1)),s8),s8))
).
```

```
Search stopped because sos empty.

============ end of search ============
```

The "KEPT" lines contain the functional representations of each output. In the actual REA implementation, we do not manually determine the output and internal gates; this determination is easy to make automatically. Given functional representations of the candidate subcircuit's outputs, the next step is to apply canonicalization rules to them, yielding a single canonical form for each of the functional representations. The following lines actually combine the previous step (rewriting output signals to functional form) and the canonicalization step:

```
include("translate.lib").
include("canonicalize.lib").

   % internal gates; i.e., gates whose outputs are internal to the
circuit.
list(demodulators).

    EQUAL(s1,  nand(a0,b0)).
    EQUAL(s2,  nand(a0,s1)).
    EQUAL(s3,  nand(s1,b0)).
    EQUAL(s4,  nand(s2,s3)).
    EQUAL(s5,  nand(s4,cin)).
    EQUAL(s6,  nand(s4,s5)).
    EQUAL(s7,  nand(s5,cin)).
    EQUAL(s8,  nand(s5,s1)).
    EQUAL(s10, nand(a1,b1)).
    EQUAL(s11, nand(a1,s10)).
    EQUAL(s12, nand(s10,b1)).
    EQUAL(s13, nand(s11,s12)).
    EQUAL(s14, nand(s13,s8)).
    EQUAL(s15, nand(s13,s14)).
    EQUAL(s16, nand(s14,s8)).
end_of_list.

   % external gates; i.e., gates whose outputs are also circuit outputs.
list(sos).
    OUTPUT(sum0, nand(s6,s7)).
    OUTPUT(cout, nand(s14,s10)).
    OUTPUT(sum1, nand(s15,s16)).
end_of_list.
```

The input file above included two other files. The first is `translate.lib`, which translates various logical operators to `exor/and` form:

```
% $Id: translate.lib,v 1.1 1996/10/31 19:21:26 eckmann Exp eckmann $
%
% Rewrite rules (demodulators) and OTTER configuration to
% translate a variety of logical formulas to EXOR/AND form,
% prior to canonicalization.
%
% Add additional rewrite rules as necessary to eliminate
% operators other than EXOR and AND.

set(demod_inf).
set(dynamic_demod).
```

```
% these weights cause OTTER to orient rewrite rules properly.
weight_list(terms).
  weight(and($(0),$(0)),-2).
  weight(exor($(0),$(0)),-2).
end_of_list.

list(demodulators).

  buf1(x) = x.

  inv(x) = exor(x,1).

  and2(x1,x2) = and(x1,x2).
  and3(x1,x2,x3) = and(x1,and(x2,x3)).
  and4(x1,x2,x3,x4) = and(x1,and(x2,and(x3,x4))).
  and5(x1,x2,x3,x4,x5) = and(x1,and(x2,and(x3,and(x4,x5)))).
  and8(x1,x2,x3,x4,x5,x6,x7,x8) =
     and(and4(x1,x2,x3,x4),and4(x5,x6,x7,x8)).

  or2(x1,x2) = inv(and(inv(x1),inv(x2))).
  or3(x1,x2,x3) = inv(and3(inv(x1),inv(x2),inv(x3))).
  or4(x1,x2,x3,x4) = inv(and4(inv(x1),inv(x2),inv(x3),inv(x4))).

  exor2(x1,x2) = exor(x1,x2).

  xnor2(x1,x2) = inv(exor(x1,x2)).

  nor2(x1,x2) = and(inv(x1),inv(x2)).
  nor3(x1,x2,x3) = and3(inv(x1),inv(x2),inv(x3)).
  nor4(x1,x2,x3,x4) = and4(inv(x1),inv(x2),inv(x3),inv(x4)).
  nor8(x1,x2,x3,x4,x5,x6,x7,x8) = and8(inv(x1),inv(x2),inv(x3),inv(x4),
                                       inv(x5),inv(x6),inv(x7),inv(x8)).

  nand(x1,x2) = nand2(x1,x2).
  nand2(x1,x2) = exor(and(x1,x2),1).
  nand3(x1,x2,x3) = exor(and3(x1,x2,x3),1).
  nand4(x1,x2,x3,x4) = exor(and4(x1,x2,x3,x4),1).
  nand5(x1,x2,x3,x4,x5) = exor(and5(x1,x2,x3,x4,x5),1).

end_of_list.
```

The second included file is `canonicalize.lib`, which canonicalizes formulas expressed entirely in terms of `exor` and `and`:

```
% $Id: canonicalize.lib,v 1.1 1996/10/31 19:21:26 eckmann Exp eckmann $
%
% Rewrite rules (demodulators) and OTTER configuration
% to canonicalize EXOR/AND formulas.
%
% Assumptions:
% 1. other logical operators have already been translated
%    to EXOR/AND form using rewrite rules in translate.lib.
% 2. files that include this library have a lex list that
%    specifies order of constants (otherwise you get a normal
%    form, but not a canonical form).

set(demod_inf).
clear(demod_history).
assign(demod_limit, -1).
assign(max_mem, -1).
assign(stats_level,0).

list(demodulators).

exor(0,x)=x.
exor(x,0)=x.
exor(x,x)=0.
exor(x,exor(x,y))=y.
exor(x,y)=exor(y,x).
exor(y,exor(x,z))=exor(x,exor(y,z)).

and(0,x)=0.
and(x,0)=0.
and(1,x)=x.
and(x,1)=x.
and(x,x)=x.
and(x,and(x,y))=and(x,y).
and(x,y)=and(y,x).
and(y,and(x,z))=and(x,and(y,z)).

and(x,exor(y,z))=exor(and(x,y),and(x,z)).

end_of_list.
```

Given the input file above and the two included files, OTTER will produce something like this:

```
----- Otter 3.0.4, August 1995 -----
The job was started by eckmann on eckmann.home, Tue Apr 22 20:01:23 1997
The command was "otter".

include("translate.lib").
------- start included file translate.lib-------
set(demod_inf).
set(dynamic_demod).
   dependent: set(order_eq).

weight_list(terms).
weight(and($(0),$(0)),-2).
weight(exor($(0),$(0)),-2).
end_of_list.

list(demodulators).
1 [] buf1(x)=x.
2 [] inv(x)=exor(x,1).
3 [] and2(x1,x2)=and(x1,x2).
4 [] and3(x1,x2,x3)=and(x1,and(x2,x3)).
5 [] and4(x1,x2,x3,x4)=and(x1,and(x2,and(x3,x4))).
6 [] and5(x1,x2,x3,x4,x5)=and(x1,and(x2,and(x3,and(x4,x5)))).
7 [] and8(x1,x2,x3,x4,x5,x6,x7,x8)=
          and(and4(x1,x2,x3,x4),and4(x5,x6,x7,x8)).
8 [] or2(x1,x2)=inv(and(inv(x1),inv(x2))).
9 [] or3(x1,x2,x3)=inv(and3(inv(x1),inv(x2),inv(x3))).
10 [] or4(x1,x2,x3,x4)=inv(and4(inv(x1),inv(x2),inv(x3),inv(x4))).
11 [] exor2(x1,x2)=exor(x1,x2).
12 [] xnor2(x1,x2)=inv(exor(x1,x2)).
13 [] nor2(x1,x2)=and(inv(x1),inv(x2)).
14 [] nor3(x1,x2,x3)=and3(inv(x1),inv(x2),inv(x3)).
```

```
15 [] nor4(x1,x2,x3,x4)=and4(inv(x1),inv(x2),inv(x3),inv(x4)).
16 [] nor8(x1,x2,x3,x4,x5,x6,x7,x8)=
           and8(inv(x1),inv(x2),inv(x3),inv(x4),
                inv(x5),inv(x6),inv(x7),inv(x8)).
17 [] nand(x1,x2)=nand2(x1,x2).
18 [] nand2(x1,x2)=exor(and(x1,x2),1).
19 [] nand3(x1,x2,x3)=exor(and3(x1,x2,x3),1).
20 [] nand4(x1,x2,x3,x4)=exor(and4(x1,x2,x3,x4),1).
21 [] nand5(x1,x2,x3,x4,x5)=exor(and5(x1,x2,x3,x4,x5),1).
end_of_list.
------- end included file translate.lib-------
include("canonicalize.lib").
------- start included file canonicalize.lib-------
WARNING: set(demod_inf) flag already set.
set(demod_inf).
clear(demod_history).
assign(demod_limit,-1).
WARNING: assign(max_mem,-1) already has that value.
assign(max_mem,-1).
assign(stats_level,0).

list(demodulators).
22 [] exor(0,x)=x.
23 [] exor(x,0)=x.
24 [] exor(x,x)=0.
25 [] exor(x,exor(x,y))=y.
26 [] exor(x,y)=exor(y,x).
27 [] exor(y,exor(x,z))=exor(x,exor(y,z)).
28 [] and(0,x)=0.
29 [] and(x,0)=0.
30 [] and(1,x)=x.
31 [] and(x,1)=x.
32 [] and(x,x)=x.
33 [] and(x,and(x,y))=and(x,y).
34 [] and(x,y)=and(y,x).
35 [] and(y,and(x,z))=and(x,and(y,z)).
36 [] and(x,exor(y,z))=exor(and(x,y),and(x,z)).
end_of_list.
------- end included file canonicalize.lib-------

list(demodulators).
37 [] EQUAL(s1,nand(a0,b0)).
38 [] EQUAL(s2,nand(a0,s1)).
39 [] EQUAL(s3,nand(s1,b0)).
40 [] EQUAL(s4,nand(s2,s3)).
41 [] EQUAL(s5,nand(s4,cin)).
42 [] EQUAL(s6,nand(s4,s5)).
43 [] EQUAL(s7,nand(s5,cin)).
44 [] EQUAL(s8,nand(s5,s1)).
45 [] EQUAL(s10,nand(a1,b1)).
46 [] EQUAL(s11,nand(a1,s10)).
47 [] EQUAL(s12,nand(s10,b1)).
48 [] EQUAL(s13,nand(s11,s12)).
49 [] EQUAL(s14,nand(s13,s8)).
50 [] EQUAL(s15,nand(s13,s14)).
51 [] EQUAL(s16,nand(s14,s8)).
end_of_list.

list(sos).
52 [] OUTPUT(sum0,nand(s6,s7)).
53 [] OUTPUT(cout,nand(s14,s10)).
```

```
54 [] OUTPUT(sum1,nand(s15,s16)).
end_of_list.
lex dependent demodulator: 26 [] exor(x,y)=exor(y,x).
lex dependent demodulator: 27 [] exor(y,exor(x,z))=exor(x,exor(y,z)).
lex dependent demodulator: 34 [] and(x,y)=and(y,x).
lex dependent demodulator: 35 [] and(y,and(x,z))=and(x,and(y,z)).

======= end of input processing =======

=========== start of search ===========

given clause #1: (wt=5) 52 [] OUTPUT(sum0,nand(s6,s7)).
** KEPT (pick-wt=7): 55 [52,demod] OUTPUT(sum0,exor(a0,exor(b0,cin))).

given clause #2: (wt=5) 53 [] OUTPUT(cout,nand(s14,s10)).
** KEPT (pick-wt=41): 56 [53,demod]
    OUTPUT(cout,
        exor(and(a0,and(a1,b0)),
            exor(and(a0,and(a1,cin)),
                exor(and(a0,and(b0,b1)),
                    exor(and(a0,and(b1,cin)),
                        exor(and(a1,b1),
                            exor(and(a1,and(b0,cin)),
                                and(b0,and(b1,cin))))))))).

given clause #3: (wt=5) 54 [] OUTPUT(sum1,nand(s15,s16)).
** KEPT (pick-wt=17): 57 [54,demod]
    OUTPUT(sum1,
        exor(a1,exor(b1,exor(and(a0,b0),
                          exor(and(a0,cin),and(b0,cin)))))).

given clause #4: (wt=7) 55 [52,demod]
OUTPUT(sum0,exor(a0,exor(b0,cin))).

given clause #5: (wt=17) 57 [54,demod]

OUTPUT(sum1,exor(a1,exor(b1,exor(and(a0,b0),exor(and(a0,cin),and(b0,cin)
))))).

given clause #6: (wt=41) 56 [53,demod]
     OUTPUT(cout,exor(and(a0,and(a1,b0)),exor(and(a0,and(a1,cin)),
exor(and(a0,and(b0,b1)),exor(and(a0,and(b1,cin)),
exor(and(a1,b1),exor(and(a1,and(b0,cin)),and(b0,and(b1,cin))))))))).

Search stopped because sos empty.

============ end of search ============
```

The three "KEPT" clauses specify the canonical forms for the 2-bit adder's three outputs.

APPENDIX B:

EQUIVALENCE OF FIGURE 9 AND FIGURE 10

One way to identify the functional equivalence of a circuit is to specify the functionality of the circuit and show that a specific circuit implies that specification. This approach is illustrated by using Figures 2 and 4 as an example. Recall that Figure 2 depicts an implementation of a full 2-bit adder using `nand` gates. Figure 4 depicts an implementation using `exor/and` gates. Figure 7 depicts an expression describing the functionality of a full 2-bit adder. The first section of this appendix describes the use of OTTER to prove the implication that Figure 2 is an implementation of the specification described by Figure 7. The second section describes the proof that Figure 4 is an implementation of the same specification by using a new approach. If Figures 2 and 4 are implementations of the same specification, they must be functionally equivalent.

Section B.2 discusses an approach for demonstrating the equivalence between a specification and an implementation. This demonstration does not depend on the order of the input signal names. We use this approach here to show that the two implementations depicted in Figures 10 and 11 are equivalent. The approach is in two parts. First we demonstrate that Figure 10 is an implementation of the specification for a 2-bit adder. Second we repeat this demonstration for Figure 11. If both circuits are implementations of the same specification, they must be equivalent. These demonstrations may also be interpreted as assigning the identical functional meaning to each of the circuits.

B.1 DEMONSTRATING THAT FIGURE 10 IS AN INSTANCE OF THE 2-BIT ADDER SPECIFICATION

The following is the OTTER input file:

```
% ---------------------------------------------------------------------
% Equivalence between Figure 10 and Specification for 2-bit adder
% ---------------------------------------------------------------------

% inference rules
set(neg_hyper_res).
set(ur_res).
set(unit_deletion).
set(para_into).
clear(para_from_right).
% search
set(pretty_print).
set(input_sos_first).

% processing limits
assign(max_mem, 96000).
assign(max_weight, 99).
```

```
% printing
clear(print_kept).

% lexical ordering -- exor must be last for the canonicalization
strategy
lex([s(x),a0,a1,b0,b1,cin,sum0,sum1,ovfl,and(x,y),exor(x,y)]).

weight_list(pick_and_purge).

   weight(s($1),0).

   % special interest in the output signals
   weight(GATE(ovfl,$(1)),-10).
   weight(GATE(sum0,$(1)),-10).
   weight(GATE(sum1,$(1)),-10).

   % it suffices to consider the original "labels" for the gates
   weight(GATE(and($(1),$(1)),$(1)),999).
   weight(GATE(exor($(1),$(1)),$(1)),999).

end_of_list.

list(usable).

   EQ(x,x).

% The following is the specification for a 2-bit adder

 -GATE(xsum0,exor(xa0,exor(xb0,xcin))) |
 -GATE(xsum1,
        exor(xa1,
                exor(xb1,
                    exor(and(xa0,xb0),
                        exor(and(xa0,xcin),
                            and(xb0,xcin)))))) |
 -GATE(xovfl,
        exor(and(xa0,and(xa1,xb0)),
             exor(and(xa0,and(xa1,xcin)),
                  exor(and(xa0,and(xb0,xb1)),
                       exor(and(xa0,and(xb1,xcin)),
                            exor(and(xa1,xb1),
                                 exor(and(xa1,and(xb0,xcin)),
                                      and(xb0,and(xb1,xcin)))))))) |
  TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).

end_of_list.

list(usable).

   % A Second version of the Netlist for Figure 10  -- paramodulating
   % from these equality units into a GATE literal expands the
expression
   % for a gate's functionality

  EQUAL(sum0,exor(a0,s(1))).
  EQUAL(s(1),exor(b0,cin)).
  EQUAL(sum1,exor(a1,s(2))).
  EQUAL(s(2),exor(b1,s(3))).
  EQUAL(s(3),exor(s(4),s(5))).
  EQUAL(s(4),and(a0,b0)).
  EQUAL(s(5),exor(s(6),s(7))).
  EQUAL(s(6),and(b0,cin)).
  EQUAL(s(7),and(a0,cin)).
  EQUAL(ovfl,exor(s(8),s(10))).
  EQUAL(s(8),and(a0,s(9))).
  EQUAL(s(9),and(b0,a1)).
  EQUAL(s(10),exor(s(11),s(13))).
  EQUAL(s(11),and(a0,s(12))).
  EQUAL(s(12),and(b0,b1)).
  EQUAL(s(13),exor(s(14),s(16))).
  EQUAL(s(14),and(a0,s(15))).
  EQUAL(s(15),and(a1,cin)).
  EQUAL(s(16),exor(s(17),s(19))).
  EQUAL(s(17),and(a0,s(18))).
  EQUAL(s(18),and(b1,cin)).
  EQUAL(s(19),exor(s(20),s(22))).
  EQUAL(s(20),and(b0,s(21))).
  EQUAL(s(21),and(a1,cin)).
  EQUAL(s(22),exor(s(23),s(25))).
  EQUAL(s(23),and(b0,s(24))).
  EQUAL(s(24),and(b1,cin)).
  EQUAL(s(25),and(a1,b1)).

end_of_list.

list(sos).

  % The gate-level netlist description of Figure 10

  GATE(sum0,exor(a0,s(1))).
  GATE(s(1),exor(b0,cin)).
  GATE(sum1,exor(a1,s(2))).
  GATE(s(2),exor(b1,s(3))).
  GATE(s(3),exor(s(4),s(5))).
  GATE(s(4),and(a0,b0)).
  GATE(s(5),exor(s(6),s(7))).
  GATE(s(6),and(b0,cin)).
  GATE(s(7),and(a0,cin)).
  GATE(ovfl,exor(s(8),s(10))).
  GATE(s(8),and(a0,s(9))).
  GATE(s(9),and(b0,a1)).
  GATE(s(10),exor(s(11),s(13))).
  GATE(s(11),and(a0,s(12))).
  GATE(s(12),and(b0,b1)).
  GATE(s(13),exor(s(14),s(16))).
  GATE(s(14),and(a0,s(15))).
  GATE(s(15),and(a1,cin)).
  GATE(s(16),exor(s(17),s(19))).
  GATE(s(17),and(a0,s(18))).
  GATE(s(18),and(b1,cin)).
  GATE(s(19),exor(s(20),s(22))).
  GATE(s(20),and(b0,s(21))).
  GATE(s(21),and(a1,cin)).
  GATE(s(22),exor(s(23),s(25))).
  GATE(s(23),and(b0,s(24))).
  GATE(s(24),and(b1,cin)).
  GATE(s(25),and(a1,b1)).

   % denial of theorem

    -TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
```

```
end_of_list.

list(demodulators).

  % canonicalize exor with respect to commutativity and associativity
  EQ(exor(x,y),exor(y,x)).
  EQ(exor(x,exor(y,z)),exor(y,exor(x,z))).
  EQ(exor(exor(x,y),z),exor(x,exor(y,z))).

  % canonicalize and with respect to commutativity and associativity
  EQ(and(x,y),and(y,x)).
  EQ(and(x,and(y,z)),and(y,and(x,z))).
  EQ(and(and(x,y),z),and(x,and(y,z))).

  % distribute and over exor
  EQ(and(x,exor(y,z)),exor(and(x,y),and(x,z))).
  EQ(and(exor(y,z),x),exor(and(x,y),and(x,z))).

end_of_list.
```

The following is the proof from the OTTER output file:

```
---------------- PROOF ----------------

2 []
-GATE(xsum0,exor(xa0,exor(xb0,xcin))) |
-GATE(
    xsum1,

exor(xa1,exor(xb1,exor(and(xa0,xb0),exor(and(xa0,xcin),and(xb0,xcin)))))
) |
-GATE(
    xovfl,
    exor(
        and(xa0,and(xa1,xb0)),
        exor(
            and(xa0,and(xa1,xcin)),
            exor(
                and(xa0,and(xb0,xb1)),
                exor(
                    and(xa0,and(xb1,xcin)),
                    exor(
                        and(xa1,xb1),

exor(and(xa1,and(xb0,xcin)),and(xb0,and(xb1,xcin)))
                    )
                )
            )
        )
    )
) |
TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
4 []
EQUAL(s(1),exor(b0,cin)).
6 []
EQUAL(s(2),exor(b1,s(3))).
7 []
EQUAL(s(3),exor(s(4),s(5))).
8 []
EQUAL(s(4),and(a0,b0)).
```

```
9 []
EQUAL(s(5),exor(s(6),s(7))).
10 []
EQUAL(s(6),and(b0,cin)).
11 []
EQUAL(s(7),and(a0,cin)).
13 []
EQUAL(s(8),and(a0,s(9))).
14 []
EQUAL(s(9),and(b0,a1)).
15 []
EQUAL(s(10),exor(s(11),s(13))).
16 []
EQUAL(s(11),and(a0,s(12))).
17 []
EQUAL(s(12),and(b0,b1)).
18 []
EQUAL(s(13),exor(s(14),s(16))).
19 []
EQUAL(s(14),and(a0,s(15))).
20 []
EQUAL(s(15),and(a1,cin)).
21 []
EQUAL(s(16),exor(s(17),s(19))).
22 []
EQUAL(s(17),and(a0,s(18))).
23 []
EQUAL(s(18),and(b1,cin)).
24 []
EQUAL(s(19),exor(s(20),s(22))).
25 []
EQUAL(s(20),and(b0,s(21))).
26 []
EQUAL(s(21),and(a1,cin)).
27 []
EQUAL(s(22),exor(s(23),s(25))).
28 []
EQUAL(s(23),and(b0,s(24))).
29 []
EQUAL(s(24),and(b1,cin)).
30 []
EQUAL(s(25),and(a1,b1)).
31 []
GATE(sum0,exor(a0,s(1))).
33 []
GATE(sum1,exor(a1,s(2))).
40 []
GATE(ovfl,exor(s(8),s(10))).
59 []
-TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
60 []
EQ(exor(x,y),exor(y,x)).
61 []
EQ(exor(x,exor(y,z)),exor(y,exor(x,z))).
63 []
EQ(and(x,y),and(y,x)).
64 []
EQ(and(x,and(y,z)),and(y,and(x,z))).
68 [para_into,31.1.2.2,4.1.1]
GATE(sum0,exor(a0,exor(b0,cin))).
69 [para_into,33.1.2.2,6.1.1,demod,60,61]
```

```
GATE(sum1,exor(s(3),exor(a1,b1))).
75 [para_into,40.1.2.1,13.1.1,demod,63,60]
GATE(ovfl,exor(s(10),and(s(9),a0))).
93 [neg_hyper,59,2]
-GATE(x,exor(y,exor(z,u))) |
-GATE(v,exor(w,exor(v6,exor(and(y,z),exor(and(y,u),and(z,u)))))) |
-GATE(
    v7,
    exor(
        and(y,and(w,z)),
        exor(
            and(y,and(w,u)),
            exor(
                and(y,and(z,v6)),
                exor(
                    and(y,and(v6,u)),

exor(and(w,v6),exor(and(w,and(z,u)),and(z,and(v6,u))))
                )
            )
        )
    )
).
94 [para_into,69.1.2.1,7.1.1,demod,61,60,61,60,61,61]
GATE(sum1,exor(s(4),exor(s(5),exor(a1,b1)))).
96 [para_into,75.1.2.2.1,14.1.1,demod,63,63]
GATE(ovfl,exor(s(10),and(a0,and(a1,b0)))).
99 [para_into,96.1.2.1,15.1.1,demod,60,61,60]
GATE(ovfl,exor(s(11),exor(s(13),and(a0,and(a1,b0))))).
100 [para_into,94.1.2.1,8.1.1,demod,61,61,60]
GATE(sum1,exor(s(5),exor(a1,exor(b1,and(a0,b0))))).
106 [para_into,100.1.2.1,9.1.1,demod,61,61,60,61,60,61,61,61,61]
GATE(sum1,exor(s(6),exor(s(7),exor(a1,exor(b1,and(a0,b0)))))).
109 [para_into,99.1.2.1,16.1.1,demod,63,61]
GATE(ovfl,exor(s(13),exor(and(s(12),a0),and(a0,and(a1,b0))))).
120 [para_into,106.1.2.1,10.1.1,demod,61,61,61,60]
GATE(sum1,exor(s(7),exor(a1,exor(b1,exor(and(a0,b0),and(b0,cin)))))).
125 [para_into,109.1.2.2.1.1,17.1.1,demod,63,60]
GATE(ovfl,exor(s(13),exor(and(a0,and(a1,b0)),and(a0,and(b0,b1))))).
134 [para_into,120.1.2.1,11.1.1,demod,61,61,61]
GATE(sum1,exor(a1,exor(b1,exor(and(a0,b0),exor(and(a0,cin),and(b0,cin))
)))).
138 [para_into,125.1.2.1,18.1.1,demod,61,60,61,60,61,61]
GATE(ovfl,exor(s(14),exor(s(16),exor(and(a0,and(a1,b0)),and(a0,and(b0,b1
)))))).
170 [para_into,138.1.2.1,19.1.1,demod,63,61]
GATE(
    ovfl,

exor(s(16),exor(and(s(15),a0),exor(and(a0,and(a1,b0)),and(a0,and(b0,b1))
)))
).
225 [para_into,170.1.2.2.1.1,20.1.1,demod,63,61]
GATE(
    ovfl,
    exor(
        s(16),

exor(and(a0,and(a1,b0)),exor(and(a0,and(a1,cin)),and(a0,and(b0,b1))))
    )
).
```

```
280 [para_into,225.1.2.1,21.1.1,demod,61,61,60,61,60,61,61,61,61]
GATE(
    ovfl,
    exor(
        s(17),
        exor(
            s(19),
            exor(
                and(a0,and(a1,b0)),
                exor(and(a0,and(a1,cin)),and(a0,and(b0,b1)))
            )
        )
    )
).
449 [para_into,280.1.2.1,22.1.1,demod,63,61]
GATE(
    ovfl,
    exor(
        s(19),
        exor(
            and(s(18),a0),
            exor(
                and(a0,and(a1,b0)),
                exor(and(a0,and(a1,cin)),and(a0,and(b0,b1)))
            )
        )
    )
).
701 [para_into,449.1.2.2.1.1,23.1.1,demod,63,61,61,60]
GATE(
    ovfl,
    exor(
        s(19),
        exor(
            and(a0,and(a1,b0)),
            exor(
                and(a0,and(a1,cin)),
                exor(and(a0,and(b0,b1)),and(a0,and(b1,cin)))
            )
        )
    )
).
956
[para_into,701.1.2.1,24.1.1,demod,61,61,61,60,61,60,61,61,61,61,61,61]
GATE(
    ovfl,
    exor(
        s(20),
        exor(
            s(22),
            exor(
                and(a0,and(a1,b0)),
                exor(
                    and(a0,and(a1,cin)),
                    exor(and(a0,and(b0,b1)),and(a0,and(b1,cin)))
                )
            )
        )
    )
).
1397 [para_into,956.1.2.1,25.1.1,demod,63,61]
```

```
GATE(
    ovfl,
    exor(
        s(22),
        exor(
            and(s(21),b0),
            exor(
                and(a0,and(a1,b0)),
                exor(
                    and(a0,and(a1,cin)),
                    exor(and(a0,and(gb0,b1)),and(a0,and(b1,cin)))
                )
            )
        )
    )
).
1833 [para_into,1397.1.2.2.1.1,26.1.1,demod,63,64,61,61,61,60]
GATE(
    ovfl,
    exor(
        s(22),
        exor(
            and(a0,and(a1,b0)),
            exor(
                and(a0,and(a1,cin)),
                exor(
                    and(a0,and(b0,b1)),
                    exor(and(a0,and(b1,cin)),and(a1,and(b0,cin)))
                )
            )
        )
    )
).
2127
[para_into,1833.1.2.1,27.1.1,demod,61,61,61,61,60,61,60,61,61,61,61,61,6
1,61,61]
GATE(
    ovfl,
    exor(
        s(23),
        exor(
            s(25),
            exor(
                and(a0,and(a1,b0)),
                exor(
                    and(a0,and(a1,cin)),
                    exor(
                        and(a0,and(b0,b1)),
                        exor(and(a0,and(b1,cin)),and(a1,and(b0,cin)))
                    )
                )
            )
        )
    )
).
2388 [para_into,2127.1.2.2.1,30.1.1,demod,61,61,61,61]
GATE(
    ovfl,
    exor(
        s(23),
        exor(
                and(a0,and(a1,b0)),
                exor(
                    and(a0,and(a1,cin)),
                    exor(
                        and(a0,and(b0,b1)),
                        exor(
                            and(a0,and(b1,cin)),
                            exor(and(a1,b1),and(a1,and(b0,cin)))
                        )
                    )
                )
            )
        )
).
2527 [para_into,2388.1.2.1,28.1.1,demod,63]
GATE(
    ovfl,
    exor(
        and(s(24),b0),
        exor(
            and(a0,and(a1,b0)),
            exor(
                and(a0,and(a1,cin)),
                exor(
                    and(a0,and(b0,b1)),
                    exor(
                        and(a0,and(b1,cin)),
                        exor(and(a1,b1),and(a1,and(b0,cin)))
                    )
                )
            )
        )
    )
).
2658 [para_into,2527.1.2.1.1,29.1.1,demod,63,61,61,61,61,61,60]
GATE(
    ovfl,
    exor(
        and(a0,and(a1,b0)),
        exor(
            and(a0,and(a1,cin)),
            exor(
                and(a0,and(b0,b1)),
                exor(
                    and(a0,and(b1,cin)),
                    exor(
                        and(a1,b1),
                        exor(and(a1,and(b0,cin)),and(b0,and(b1,cin)))
                    )
                )
            )
        )
    )
).
2866 [ur,93,134,2658]
-GATE(x,exor(a0,exor(b0,cin))).
2867 [binary,2866.1,68.1]
.
------------ end of proof -------------
```

B.2 DEMONSTRATING THAT FIGURE 11 IS AN INSTANCE OF THE 2-BIT ADDER SPECIFICATION

The following is the input file for this example:

```
% -----------------------------------------------------------------------
---
% Equivalence between Figure 11 and Specification for 2-bit adder
% -----------------------------------------------------------------------
---

% inference rules
set(neg_hyper_res).
set(ur_res).
set(unit_deletion).
set(para_into).
clear(para_from_right).
% search
set(pretty_print).
set(input_sos_first).

% processing limits
assign(max_mem, 96000).
assign(max_weight, 99).

% printing
clear(print_kept).

% lexical ordering -- exor must be last for the canonicalization
strategy
lex([s(x),a0,b0,a1,b1,cin,sum0,sum1,ovfl,and(x,y),exor(x,y)]).

weight_list(pick_and_purge).

   weight(s($1),0).

   % special interest in the output signals
   weight(GATE(ovfl,$(1)),-10).
   weight(GATE(sum0,$(1)),-10).
   weight(GATE(sum1,$(1)),-10).

   % it suffices to consider the original "labels" for the gates
   weight(GATE(and($(1),$(1)),$(1)),999).
   weight(GATE(exor($(1),$(1)),$(1)),999).

end_of_list.

list(usable).

   EQ(x,x).

% The following is the specification for a 2-bit adder

 -GATE(xsum0,exor(xa0,exor(xb0,xcin))) |
 -GATE(xsum1,
        exor(xa1,
                exor(xb1,
                    exor(and(xa0,xb0),
                        exor(and(xa0,xcin),
                            and(xb0,xcin)))))) |
 -GATE(xovfl,
       exor(and(xa0,and(xa1,xb0)),
            exor(and(xa0,and(xa1,xcin)),
                 exor(and(xa0,and(xb0,xb1)),
                      exor(and(xa0,and(xb1,xcin)),
                           exor(and(xa1,xb1),
                                exor(and(xa1,and(xb0,xcin)),
                                     and(xb0,and(xb1,xcin)))))))) |
  TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).

end_of_list.

list(usable).

   % A Second version of the Netlist for Figure 11  -- paramodulating
   % from these equality units into a GATE literal expands the
expression
   % for a gate's functionality

  EQUAL(s(1),and(b1,cin)).
  EQUAL(s(2),and(b0,cin)).
  EQUAL(s(3),and(b0,s(1))).
  EQUAL(s(4),and(a1,s(2))).
  EQUAL(s(5),exor(s(3),s(4))).
  EQUAL(s(6),and(a1,b1)).
  EQUAL(s(7),exor(s(5),s(6))).
  EQUAL(s(8),and(b1,cin)).
  EQUAL(s(9),and(a0,s(8))).
  EQUAL(s(10),exor(s(7),s(9))).
  EQUAL(s(11),and(b0,b1)).
  EQUAL(s(12),and(a0,s(11))).
  EQUAL(s(13),and(a1,cin)).
  EQUAL(s(14),exor(s(10),s(12))).
  EQUAL(s(15),and(a0,s(13))).
  EQUAL(s(16),exor(s(14),s(15))).
  EQUAL(s(17),and(a1,b0)).
  EQUAL(s(18),and(a0,s(17))).
  EQUAL(ovfl,exor(s(16),s(18))).
  EQUAL(ss(01),exor(b0,cin)).
  EQUAL(sum0,exor(a0,ss(01))).
  EQUAL(ss(11),and(b0,cin)).
  EQUAL(ss(12),and(a0,cin)).
  EQUAL(ss(13),exor(ss(11),ss(12))).
  EQUAL(ss(14),and(a0,b0)).
  EQUAL(ss(15),exor(ss(13),ss(14))).
  EQUAL(ss(16),exor(b1,ss(15))).
  EQUAL(sum1,exor(a1,ss(16))).

end_of_list.

list(sos).

  % The gate-level netlist description of Figure 11

  GATE(s(1),and(b1,cin)).
  GATE(s(2),and(b0,cin)).
  GATE(s(3),and(b0,s(1))).
  GATE(s(4),and(a1,s(2))).
  GATE(s(5),exor(s(3),s(4))).
  GATE(s(6),and(a1,b1)).
  GATE(s(7),exor(s(5),s(6))).
  GATE(s(8),and(b1,cin)).
  GATE(s(9),and(a0,s(8))).
  GATE(s(10),exor(s(7),s(9))).
  GATE(s(11),and(b0,b1)).
  GATE(s(12),and(a0,s(11))).
  GATE(s(13),and(a1,cin)).
  GATE(s(14),exor(s(10),s(12))).
  GATE(s(15),and(a0,s(13))).
  GATE(s(16),exor(s(14),s(15))).
  GATE(s(17),and(a1,b0)).
  GATE(s(18),and(a0,s(17))).
  GATE(ovfl,exor(s(16),s(18))).
  GATE(ss(01),exor(b0,cin)).
  GATE(sum0,exor(a0,ss(01))).
  GATE(ss(11),and(b0,cin)).
  GATE(ss(12),and(a0,cin)).
  GATE(ss(13),exor(ss(11),ss(12))).
  GATE(ss(14),and(a0,b0)).
  GATE(ss(15),exor(ss(13),ss(14))).
  GATE(ss(16),exor(b1,ss(15))).
  GATE(sum1,exor(a1,ss(16))).

   % denial of theorem

    -TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).

end_of_list.

list(demodulators).

  % canonicalize exor with respect to commutativity and associativity
  EQ(exor(x,y),exor(y,x)).
  EQ(exor(x,exor(y,z)),exor(y,exor(x,z))).
  EQ(exor(exor(x,y),z),exor(x,exor(y,z))).

  % canonicalize and with respect to commutativity and associativity
  EQ(and(x,y),and(y,x)).
  EQ(and(x,and(y,z)),and(y,and(x,z))).
  EQ(and(and(x,y),z),and(x,and(y,z))).

  % distribute and over exor
  EQ(and(x,exor(y,z)),exor(and(x,y),and(x,z))).
  EQ(and(exor(y,z),x),exor(and(x,y),and(x,z))).

end_of_list.
```

The following is the proof excerpted from the OTTER output file:

```
---------------- PROOF ----------------

2 []
-GATE(xsum0,exor(xa0,exor(xb0,xcin))) |
-GATE(
    xsum1,

exor(xa1,exor(xb1,exor(and(xa0,xb0),exor(and(xa0,xcin),and(xb0,xcin)))))
) |
-GATE(
    xovfl,
    exor(
        and(xa0,and(xa1,xb0)),
        exor(
            and(xa0,and(xa1,xcin)),
            exor(
                and(xa0,and(xb0,xb1)),
                exor(
                    and(xa0,and(xb1,xcin)),
                    exor(
                        and(xa1,xb1),

exor(and(xa1,and(xb0,xcin)),and(xb0,and(xb1,xcin)))
                    )
                )
            )
        )
    )
) |
TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
3 []
EQUAL(s(1),and(b1,cin)).
4 []
EQUAL(s(2),and(b0,cin)).
5 []
EQUAL(s(3),and(b0,s(1))).
6 []
EQUAL(s(4),and(a1,s(2))).
7 []
EQUAL(s(5),exor(s(3),s(4))).
8 []
EQUAL(s(6),and(a1,b1)).
9 []
EQUAL(s(7),exor(s(5),s(6))).
10 []
EQUAL(s(8),and(b1,cin)).
11 []
EQUAL(s(9),and(a0,s(8))).
12 []
EQUAL(s(10),exor(s(7),s(9))).
13 []
EQUAL(s(11),and(b0,b1)).
14 []
EQUAL(s(12),and(a0,s(11))).
15 []
EQUAL(s(13),and(a1,cin)).
16 []
EQUAL(s(14),exor(s(10),s(12))).
17 []
EQUAL(s(15),and(a0,s(13))).
18 []
EQUAL(s(16),exor(s(14),s(15))).
19 []
EQUAL(s(17),and(a1,b0)).
20 []
EQUAL(s(18),and(a0,s(17))).
22 []
EQUAL(ss(01),exor(b0,cin)).
24 []
EQUAL(ss(11),and(b0,cin)).
25 []
EQUAL(ss(12),and(a0,cin)).
26 []
```

```
EQUAL(ss(13),exor(ss(11),ss(12))).
27 []
EQUAL(ss(14),and(a0,b0)).
28 []
EQUAL(ss(15),exor(ss(13),ss(14))).
29 []
EQUAL(ss(16),exor(b1,ss(15))).
49 []
GATE(ovfl,exor(s(16),s(18))).
51 []
GATE(sum0,exor(a0,ss(01))).
58 []
GATE(sum1,exor(a1,ss(16))).
59 []
-TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
60 []
EQ(exor(x,y),exor(y,x)).
61 []
EQ(exor(x,exor(y,z)),exor(y,exor(x,z))).
63 []
EQ(and(x,y),and(y,x)).
64 []
EQ(and(x,and(y,z)),and(y,and(x,z))).
85 [para_into,49.1.2.2,20.1.1,demod,63]
GATE(ovfl,exor(s(16),and(s(17),a0))).
86 [para_into,51.1.2.2,22.1.1]
GATE(sum0,exor(a0,exor(b0,cin))).
92 [para_into,58.1.2.2,29.1.1]
GATE(sum1,exor(a1,exor(b1,ss(15)))).
93 [neg_hyper,59,2]
-GATE(x,exor(y,exor(z,u))) |
-GATE(v,exor(w,exor(v6,exor(and(y,z),exor(and(y,u),and(z,u)))))) |
-GATE(
    v7,
    exor(
        and(y,and(w,z)),
        exor(
            and(y,and(w,u)),
            exor(
                and(y,and(z,v6)),
                exor(
                    and(y,and(v6,u)),

exor(and(w,v6),exor(and(w,and(z,u)),and(z,and(v6,u))))
                )
            )
        )
    )
).
94 [para_into,92.1.2.2.2,28.1.1]
GATE(sum1,exor(a1,exor(b1,exor(ss(13),ss(14))))).
96 [para_into,85.1.2.2.1,19.1.1,demod,63,63]
GATE(ovfl,exor(s(16),and(a0,and(b0,a1)))).
99 [para_into,96.1.2.1,18.1.1,demod,60,61,60]
GATE(ovfl,exor(s(14),exor(s(15),and(a0,and(b0,a1))))).
101 [para_into,94.1.2.2.2.2,27.1.1,demod,60]
GATE(sum1,exor(a1,exor(b1,exor(and(a0,b0),ss(13))))).
106 [para_into,101.1.2.2.2.2,26.1.1]
GATE(sum1,exor(a1,exor(b1,exor(and(a0,b0),exor(ss(11),ss(12)))))).
110 [para_into,99.1.2.2.1,17.1.1,demod,63]
GATE(ovfl,exor(s(14),exor(and(s(13),a0),and(a0,and(b0,a1))))).
```

```
120 [para_into,106.1.2.2.2.2.1,24.1.1]
GATE(sum1,exor(a1,exor(b1,exor(and(a0,b0),exor(and(b0,cin),ss(12)))))).
125 [para_into,110.1.2.2.1.1,15.1.1,demod,63,60]
GATE(ovfl,exor(s(14),exor(and(a0,and(b0,a1)),and(a0,and(a1,cin))))).
134 [para_into,120.1.2.2.2.2.2,25.1.1,demod,60]
GATE(sum1,exor(a1,exor(b1,exor(and(a0,b0),exor(and(a0,cin),and(b0,cin)))
))).
138 [para_into,125.1.2.1,16.1.1,demod,61,60,61,60,61,61]
GATE(ovfl,exor(s(10),exor(s(12),exor(and(a0,and(b0,a1)),and(a0,and(a1,ci
n)))))).
171 [para_into,138.1.2.2.1,14.1.1,demod,63]
GATE(
    ovfl,
    exor(
        s(10),
        exor(and(s(11),a0),exor(and(a0,and(b0,a1)),and(a0,and(a1,cin))))
    )
).
224 [para_into,171.1.2.2.1.1,13.1.1,demod,63,61]
GATE(
    ovfl,
    exor(
        s(10),

exor(and(a0,and(b0,a1)),exor(and(a0,and(b0,b1)),and(a0,and(a1,cin))))
    )
).
284 [para_into,224.1.2.1,12.1.1,demod,61,61,60,61,60,61,61,61,61]
GATE(
    ovfl,
    exor(
        s(7),
        exor(
            s(9),
            exor(
                and(a0,and(b0,a1)),
                exor(and(a0,and(b0,b1)),and(a0,and(a1,cin)))
            )
        )
    )
).
443 [para_into,284.1.2.2.1,11.1.1,demod,63]
GATE(
    ovfl,
    exor(
        s(7),
        exor(
            and(s(8),a0),
            exor(
                and(a0,and(b0,a1)),
                exor(and(a0,and(b0,b1)),and(a0,and(a1,cin)))
            )
        )
    )
).
670 [para_into,443.1.2.2.1.1,10.1.1,demod,63,61,61,60]
GATE(
    ovfl,
    exor(
        s(7),
        exor(
            and(a0,and(b0,a1)),
            exor(
                and(a0,and(b0,b1)),
                exor(and(a0,and(a1,cin)),and(a0,and(b1,cin)))
            )
        )
    )
).
884
[para_into,670.1.2.1,9.1.1,demod,61,61,61,60,61,60,61,61,61,61,61,61]
GATE(
    ovfl,
    exor(
        s(5),
        exor(
            s(6),
            exor(
                and(a0,and(b0,a1)),
                exor(
                    and(a0,and(b0,b1)),
                    exor(and(a0,and(a1,cin)),and(a0,and(b1,cin)))
                )
            )
        )
    )
).
1307 [para_into,884.1.2.2.1,8.1.1,demod,61,61,61,60]
GATE(
    ovfl,
    exor(
        s(5),
        exor(
            and(a0,and(b0,a1)),
            exor(
                and(a0,and(b0,b1)),

exor(and(a0,and(a1,cin)),exor(and(a0,and(b1,cin)),and(a1,b1)))
            )
        )
    )
).
1617
[para_into,1307.1.2.1,7.1.1,demod,61,61,61,61,60,61,60,61,61,61,61,61,61
,61,61]
GATE(
    ovfl,
    exor(
        s(3),
        exor(
            s(4),
            exor(
                and(a0,and(b0,a1)),
                exor(
                    and(a0,and(b0,b1)),
                    exor(
                        and(a0,and(a1,cin)),
                        exor(and(a0,and(b1,cin)),and(a1,b1))
                    )
                )
            )
        )
    )
).
2046 [para_into,1617.1.2.1,5.1.1,demod,63,61]
GATE(
    ovfl,
    exor(
        s(4),
        exor(
            and(s(1),b0),
            exor(
                and(a0,and(b0,a1)),
                exor(
                    and(a0,and(b0,b1)),
                    exor(
                        and(a0,and(a1,cin)),
                        exor(and(a0,and(b1,cin)),and(a1,b1))
                    )
                )
            )
        )
    )
).
2291 [para_into,2046.1.2.2.1.1,3.1.1,demod,63,61,61,61,61]
GATE(
    ovfl,
    exor(
        s(4),
        exor(
            and(a0,and(b0,a1)),
            exor(
                and(a0,and(b0,b1)),
                exor(
                    and(a0,and(a1,cin)),
                    exor(
                        and(a0,and(b1,cin)),
                        exor(and(b0,and(b1,cin)),and(a1,b1))
                    )
                )
            )
        )
    )
).
2422 [para_into,2291.1.2.1,6.1.1,demod,63]
GATE(
    ovfl,
    exor(
        and(s(2),a1),
        exor(
            and(a0,and(b0,a1)),
            exor(
                and(a0,and(b0,b1)),
                exor(
                    and(a0,and(a1,cin)),
                    exor(
                        and(a0,and(b1,cin)),
                        exor(and(b0,and(b1,cin)),and(a1,b1))
                    )
                )
            )
        )
    )
```

```
).
2549 [para_into,2422.1.2.1.1,4.1.1,demod,63,64,61,61,61,61]
GATE(
    ovfl,
    exor(
        and(a0,and(b0,a1)),
        exor(
            and(a0,and(b0,b1)),
            exor(
                and(a0,and(a1,cin)),
                exor(
                    and(a0,and(b1,cin)),
                    exor(
                        and(b0,and(a1,cin)),
                        exor(and(b0,and(b1,cin)),and(a1,b1))
                    )
                )
            )
        )
    )
).
2747 [ur,93,86,134,demod,63,64,61,60,61]
-GATE(
    x,
    exor(
        and(a0,and(b0,a1)),
        exor(
            and(a0,and(b0,b1)),
            exor(
                and(a0,and(a1,cin)),
                exor(
                    and(a0,and(b1,cin)),
                    exor(
                        and(b0,and(a1,cin)),
                        exor(and(b0,and(b1,cin)),and(a1,b1))
                    )
                )
            )
        )
    )
).
2748 [binary,2747.1,2549.1]
------------ end of proof -------------
```

APPENDIX C:

EQUIVALENCE OF FIGURE 2 AND FIGURE 4

Appendix A demonstrates how rewrite rules transform a gate-level netlist – expressed as a set of GATE(...) clauses – into logical expressions for the outputs of a 2-bit adder. These logical expressions were in an `exor/and` canonical form based on the default OTTER lexical order. This appendix builds on the Appendix A approach by illustrating how OTTER demonstrates the equivalence of the circuits depicted in Figures 2 and 4 of the report.

C.1. EQUIVALENCE OF FIGURE 2 AND FIGURE 4 USING REWRITE RULES

The following statements indicate that the two libraries described in Appendix A (i.e., `translate.lib` and `canonicalize.lib`), will be included:

```
----- Otter 3.0.3k, July 1995

include("translate.lib").
include("canonicalize.lib").
set(sos_queue).

list(demodulators).
```

The following is a gate-level description of the circuit depicted in Figure 4 of the report:

```
37 [] EQUAL(s1,and(b1,cin)).
38 [] EQUAL(s2,and(b0,s1)).
39 [] EQUAL(s3,and(a0,s1)).
40 [] EQUAL(s4,and(b0,b1)).
41 [] EQUAL(s5,and(a0,s4)).
42 [] EQUAL(s6,exor(s2,s3)).
43 [] EQUAL(s7,exor(s5,s6)).
44 [] EQUAL(s8,and(cin,a1)).
45 [] EQUAL(s9,and(a1,b0)).
46 [] EQUAL(s10,and(b0,s8)).
47 [] EQUAL(s11,and(a0,s8)).
48 [] EQUAL(s12,and(a0,s9)).
49 [] EQUAL(s13,exor(s10,s11)).
50 [] EQUAL(s14,exor(s12,s13)).
51 [] EQUAL(s15,exor(s7,s14)).
52 [] EQUAL(s16,and(a1,b1)).
53 [] EQUAL(oflow,exor(s15,s16)).
54 [] EQUAL(ss01,exor(b0,cin)).
55 [] EQUAL(sm0,exor(a0,ss01)).
56 [] EQUAL(ss11,and(b0,cin)).
57 [] EQUAL(ss12,and(a0,cin)).
58 [] EQUAL(ss13,exor(ss11,ss12)).
59 [] EQUAL(ss14,and(a0,b0)).
60 [] EQUAL(ss15,exor(ss13,ss14)).
61 [] EQUAL(ss16,exor(b1,ss15)).
62 [] EQUAL(sm1,exor(a1,ss16)).
```

The following is a gate-level description of the circuit depicted in Figure 2 of the report:

```
63 [] EQUAL(s(1),nand(a0,b0)).
64 [] EQUAL(s(2),nand(a0,s(1))).
65 [] EQUAL(s(3),nand(s(1),b0)).
66 [] EQUAL(s(4),nand(s(2),s(3))).
67 [] EQUAL(s(5),nand(s(4),cin)).
68 [] EQUAL(s(6),nand(s(4),s(5))).
69 [] EQUAL(s(7),nand(s(5),cin)).
70 [] EQUAL(s(8),nand(s(5),s(1))).
71 [] EQUAL(s(10),nand(a1,b1)).
72 [] EQUAL(s(11),nand(a1,s(10))).
73 [] EQUAL(s(12),nand(s(10),b1)).
74 [] EQUAL(s(13),nand(s(11),s(12))).
75 [] EQUAL(s(14),nand(s(13),s(8))).
76 [] EQUAL(s(15),nand(s(13),s(14))).
77 [] EQUAL(s(16),nand(s(14),s(8))).
78 [] EQUAL(sum0,nand(s(6),s(7))).
79 [] EQUAL(sum1,nand(s(15),s(16))).
80 [] EQUAL(ovfl,nand(s(10),s(14))).
end_of_list.

list(sos).
```

Our approach for this demonstration will be to state that one circuit is a 2-bit adder:

```
81 [] TWO_BIT_ADDER(sm0,sm1,oflow).
```

and that the other is not. If the two circuits are equivalent, on the basis of the rules described in `translate.lib` and `canonicalize.lib`, OTTER will find a conflict. Specifically, *if* both circuits are instances of a 2-bit adder, a contradiction exists, and this constitutes a proof that both circuits are 2-bit adders.

```
82 [] -TWO_BIT_ADDER(sum0,sum1,ovfl).
end_of_list.
lex dependent demodulator: 26 [] exor(x,y)=exor(y,x).
lex dependent demodulator: 27 [] exor(y,exor(x,z))=exor(x,exor(y,z)).
lex dependent demodulator: 34 [] and(x,y)=and(y,x).
lex dependent demodulator: 35 [] and(y,and(x,z))=and(x,and(y,z)).

======= end of input processing =======

=========== start of search ===========

Starting on level 1, last kept clause of level 0 is 82.


given clause #1: (wt=4) 81 [] TWO_BIT_ADDER(sm0,sm1,oflow).
** KEPT (pick-wt=60): 83 [81,demod]
TWO_BIT_ADDER(exor(a0,exor(b0,cin)),exor(a1,exor(b1,exor(and(a0,b0),exor
(and(a0,cin),and(b0,cin))))),exor(and(a0,and(a1,b0)),exor(and(a0,and(a1,
cin)),exor(and(a0,and(b0,b1)),exor(and(a0,and(b1,cin)),exor(and(a1,b1),e
xor(and(a1,and(b0,cin)),and(b0,and(b1,cin))))))))).

given clause #2: (wt=4) 82 [] -TWO_BIT_ADDER(sum0,sum1,ovfl).
```

```
** KEPT (pick-wt=60): 84 [82,demod] -
TWO_BIT_ADDER(exor(a0,exor(b0,cin)),exor(a1,exor(b1,exor(and(a0,b0),exor
(and(a0,cin),and(b0,cin))))),exor(and(a0,and(a1,b0)),exor(and(a0,and(a1,
cin)),exor(and(a0,and(b0,b1)),exor(and(a0,and(b1,cin)),exor(and(a1,b1),e
xor(and(a1,and(b0,cin)),and(b0,and(b1,cin))))))))).

----> UNIT CONFLICT at   0.29 sec ----> 85 [binary,84.1,83.1] $F.

Length of proof is 2.  Level of proof is 1.
```

The following is a summary of the OTTER proof:

```
---------------- PROOF ----------------

81 [] TWO_BIT_ADDER(sm0,sm1,oflow).
82 [] -TWO_BIT_ADDER(sum0,sum1,ovfl).
83 [81,demod]
TWO_BIT_ADDER(exor(a0,exor(b0,cin)),exor(a1,exor(b1,exor(and(a0,b0),exor
(and(a0,cin),and(b0,cin))))),exor(and(a0,and(a1,b0)),exor(and(a0,and(a1,
cin)),exor(and(a0,and(b0,b1)),exor(and(a0,and(b1,cin)),exor(and(a1,b1),e
xor(and(a1,and(b0,cin)),and(b0,and(b1,cin))))))))).
84 [82,demod] -
TWO_BIT_ADDER(exor(a0,exor(b0,cin)),exor(a1,exor(b1,exor(and(a0,b0),exor
(and(a0,cin),and(b0,cin))))),exor(and(a0,and(a1,b0)),exor(and(a0,and(a1,
cin)),exor(and(a0,and(b0,b1)),exor(and(a0,and(b1,cin)),exor(and(a1,b1),e
xor(and(a1,and(b0,cin)),and(b0,and(b1,cin))))))))).
85 [binary,84.1,83.1] $F.

------------ end of proof -------------


Search stopped by max_proofs option.

============ end of search ============
```

However, this approach is of limited use with regard to reverse engineering. The problem is that both circuits, as described in the GATE(...) clauses, have identical input signal names. This situation raises two issues:

1. OTTER assumes a lexical order based on the appearance of terms in its input file. The identical signal names in conjunction with this order simplifies processing.

2. The gate-level netlists that are the subject of a reverse engineering analysis will have inputs with randomly ordered input signal names.

Thus, the above proof is propitious and, unfortunately, inappropriate for our application. The next section demonstrates an improved approach.

C.2. EQUIVALENCE OF FIGURE 4 AND FIGURE 7

One way to identify the functional equivalence between two circuits is to specify the functionality of the circuits and show that both instances of circuits imply that specification. This approach is illustrated by using Figure 4 as an example. Recall that Figure 4 depicts an implementation of a 2-bit adder using `exor/and` gates. Figure 7 depicts an expression describing part of the functionality of a full 2-bit adder. This section of Appendix C describes how OTTER proves the implication that Figure 4 is an implementation of the specification described by Figure 7.

The following information is excerpted from the actual OTTER input file. In this example, the set of inference rules being used differs from that used in the previous example. These rules (in conjunction with weighting and lexical ordering) intuitively direct OTTER in search of the desired proof. An explanation of these inference rules is beyond the scope of this report; see McCune (1994) for further information.

A further distinctive feature of this approach is that it removes dependence on lexical order. To illustrate this change, the inputs to the netlist have been arbitrarily assigned. Specifically, the inputs are randomly assigned the names `i0-i4`, and there is no correlation between this assignment and the previously assigned names of `a0, a1, b0, b1`, and `cin`.

```
% inference rules
set(neg_hyper_res).
set(ur_res).
set(unit_deletion).
set(para_into).
clear(para_from_right).
set(demod_inf).
set(dynamic_demod).
% search
set(input_sos_first).

% processing limits
assign(max_mem, 96000).
assign(max_weight, 99).

% printing
clear(print_kept).

% lexical ordering -- exor must be last for the canonicalization
strategy
lex([s(x),i0,i1,i2,i3,i4,sum0,sum1,ovfl,and(x,y),exor(x,y)]).

weight_list(pick_and_purge).

   weight(s($1),0).

   % special interest in the output signals
   weight(GATE(ovfl,$(1)),-10).
   weight(GATE(sum0,$(1)),-10).
   weight(GATE(sum1,$(1)),-10).
   % it suffices to consider the original "labels" for the gates
   weight(GATE(and($(1),$(1)),$(1)),999).
   weight(GATE(exor($(1),$(1)),$(1)),999).

end_of_list.

list(usable).

   EQ(x,x).

% specify the functionality of a 2-bit adder

 -GATE(xsum0,exor(xa0,exor(xb0,xcin))) |
 -GATE(xsum1,
        exor(xa1,
                exor(xb1,
                    exor(and(xa0,xb0),
                        exor(and(xa0,xcin),
                            and(xb0,xcin)))))) |
 -GATE(xovfl,
        exor(and(xa0,and(xa1,xb0)),
             exor(and(xa0,and(xa1,xcin)),
                  exor(and(xa0,and(xb0,xb1)),
                       exor(and(xa0,and(xb1,xcin)),
                            exor(and(xa1,xb1),
                                 exor(and(xa1,and(xb0,xcin)),
                                      and(xb0,and(xb1,xcin)))))))) |
  TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).

end_of_list.
```

The following is a description of the circuit depicted in Figure 4. This description is repeated in the set of support list, substituting GATE for EQ. The combination of these two descriptions allows OTTER to rewrite atoms as necessary to prove that the circuit as described by a netlist implements the functions specified above.

```
list(usable).

EQ(sum0,exor(s(3),i1)).
EQ(s(1),and(i4,i1)).
EQ(s(2),and(i2,i4)).
EQ(s(3),exor(i2,i4)).
EQ(s(4),and(i2,i1)).
EQ(s(5),exor(s(1),s(2))).
EQ(s(6),exor(s(4),s(5))).
EQ(s(7),and(i3,s(6))).
EQ(s(8),and(i0,i3)).
EQ(s(9),exor(i0,i3)).
EQ(s(10),and(s(6),i0)).
EQ(s(11),exor(s(7),s(8))).
EQ(sum1,exor(s(6),s(9))).
EQ(ovfl,exor(s(10),s(11))).

end_of_list.

list(sos).

The next set of clauses describe the gate-level netlist.
GATE(sum0,exor(s(3),i1)).
GATE(s(1),and(i4,i1)).
GATE(s(2),and(i2,i4)).
GATE(s(3),exor(i2,i4)).
GATE(s(4),and(i2,i1)).
GATE(s(5),exor(s(1),s(2))).
GATE(s(6),exor(s(4),s(5))).
GATE(s(7),and(i3,s(6))).
GATE(s(8),and(i0,i3)).
GATE(s(9),exor(i0,i3)).
GATE(s(10),and(s(6),i0)).
GATE(s(11),exor(s(7),s(8))).
GATE(sum1,exor(s(6),s(9))).
GATE(ovfl,exor(s(10),s(11))).
```

The following denial is used to state that the circuit as described by the gate-level netlist does not implement the functionality described for a TWO_BIT_ADDR with five input signals (xa0,xa1,xb0,xb1,xcin).

```
% denial of theorem
-TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
end_of_list.
list(demodulators).
```

The first set of demodulators (called rewrite rules in the report; demodulators are OTTER's way of applying rewrite rules) ensures that a canonical form is maintained within the OTTER environment (recall that this form is relative to the lex list noted at the beginning of Appendix B). Without a canonical form and a lexlist, OTTER would be unable to efficiently perform the required processing.

```
% canonicalize exor with respect to commutativity and associativity
EQ(exor(x,y),exor(y,x)).
EQ(exor(x,exor(y,z)),exor(y,exor(x,z))).
EQ(exor(exor(x,y),z),exor(x,exor(y,z))).

% canonicalize and with respect to commutativity and associativity
EQ(and(x,y),and(y,x)).
EQ(and(x,and(y,z)),and(y,and(x,z))).
EQ(and(and(x,y),z),and(x,and(y,z))).

% distribute and over exor
EQ(and(x,exor(y,z)),exor(and(x,y),and(x,z))).
EQ(and(exor(y,z),x),exor(and(x,y),and(x,z))).
```

The following is an outline of the proof generated by OTTER.

```
---------------- PROOF ----------------
17 []
-GATE(xsum0,exor(xa0,exor(xb0,xcin))) |
-GATE(
    xsum1,

exor(xa1,exor(xb1,exor(and(xa0,xb0),exor(and(xa0,xcin),and(xb0,xcin)))))
) |
-GATE(
    xovfl,
    exor(
        and(xa0,and(xa1,xb0)),
        exor(
            and(xa0,and(xa1,xcin)),
            exor(
                and(xa0,and(xb0,xb1)),
                exor(
                    and(xa0,and(xb1,xcin)),
                    exor(
                        and(xa1,xb1),

exor(and(xa1,and(xb0,xcin)),and(xb0,and(xb1,xcin)))
                    )
                )
            )
        )
    )
) |
TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
19 []
EQ(s(1),and(i4,i1)).
20 []
EQ(s(2),and(i2,i4)).
21 []
EQ(s(3),exor(i2,i4)).
22 []
EQ(s(4),and(i2,i1)).
23 []
EQ(s(5),exor(s(1),s(2))).
24 []
EQ(s(6),exor(s(4),s(5))).
25 []
EQ(s(7),and(i3,s(6))).
26 []
EQ(s(8),and(i0,i3)).
27 []
EQ(s(9),exor(i0,i3)).
28 []
EQ(s(10),and(s(6),i0)).
29 []
EQ(s(11),exor(s(7),s(8))).
32 []
GATE(sum0,exor(s(3),i1)).
44 []
GATE(sum1,exor(s(6),s(9))).
45 []
GATE(ovfl,exor(s(10),s(11))).
46 []
-TWO_BIT_ADDR(xa0,xa1,xb0,xb1,xcin).
55 [para_into,32.1.2.1,21.1.1,demod]
```

On the basis of weighting, OTTER, when selecting which clause to process next, will concentrate its search on clauses that match the structure indicated by the following three clauses:

```
GATE(sum0,exor(i1,exor(i2,i4))).
68 [para_into,44.1.2.2,27.1.1]
GATE(sum1,exor(s(6),exor(i0,i3))).
69 [para_into,45.1.2.1,28.1.1,demod]
GATE(ovfl,exor(s(11),and(s(6),i0))).
71 [neg_hyper,46,17]
-GATE(x,exor(y,exor(z,u))) |
-GATE(v,exor(w,exor(v6,exor(and(y,z),exor(and(y,u),and(z,u)))))) |
-GATE(
    v7,
    exor(
        and(y,and(w,z)),
        exor(
            and(y,and(w,u)),
            exor(
                and(y,and(z,v6)),
                exor(
                    and(y,and(v6,u)),

exor(and(w,v6),exor(and(w,and(z,u)),and(z,and(v6,u))))
                )
            )
        )
    )
).
72 [para_into,68.1.2.1,24.1.1,demod]
GATE(sum1,exor(s(4),exor(s(5),exor(i0,i3)))).
74 [para_into,69.1.2.2.1,24.1.1,demod]
GATE(ovfl,exor(s(11),exor(and(s(4),i0),and(s(5),i0)))).
79 [para_into,72.1.2.1,22.1.1,demod]
GATE(sum1,exor(s(5),exor(i0,exor(i3,and(i1,i2))))).
87 [para_into,79.1.2.1,23.1.1,demod]
GATE(sum1,exor(s(1),exor(s(2),exor(i0,exor(i3,and(i1,i2)))))).
93 [para_into,74.1.2.2.1.1,22.1.1,demod]
GATE(ovfl,exor(s(11),exor(and(s(5),i0),and(i0,and(i1,i2))))).
102 [para_into,87.1.2.1,19.1.1,demod]
GATE(sum1,exor(s(2),exor(i0,exor(i3,exor(and(i1,i2),and(i1,i4)))))).
107 [para_into,93.1.2.2.1.1,23.1.1,demod]
GATE(ovfl,exor(s(11),exor(and(s(1),i0),exor(and(s(2),i0),and(i0,and(i1,i
2)))))).
110 [para_into,102.1.2.1,20.1.1,demod]
GATE(sum1,exor(i0,exor(i3,exor(and(i1,i2),exor(and(i1,i4),and(i2,i4)))))
).
134 [para_into,107.1.2.2.1.1,19.1.1,demod]
GATE(
    ovfl,

exor(s(11),exor(and(s(2),i0),exor(and(i0,and(i1,i2)),and(i0,and(i1,i4)))
))
).
151 [para_into,134.1.2.2.1.1,20.1.1,demod]
GATE(
    ovfl,
    exor(
        s(11),

exor(and(i0,and(i1,i2)),exor(and(i0,and(i1,i4)),and(i0,and(i2,i4))))
    )
).
157 [para_into,151.1.2.1,29.1.1,demod]
GATE(
    ovfl,
 exor(
        s(7),
        exor(
            s(8),
            exor(
                and(i0,and(i1,i2)),
                exor(and(i0,and(i1,i4)),and(i0,and(i2,i4)))
            )
        )
    )
).
206 [para_into,157.1.2.2.1,26.1.1]
GATE(
    ovfl,
    exor(
        s(7),
        exor(
            and(i0,i3),
            exor(
                and(i0,and(i1,i2)),
                exor(and(i0,and(i1,i4)),and(i0,and(i2,i4)))
            )
        )
    )
).
239 [para_into,206.1.2.1,25.1.1,demod]
GATE(
    ovfl,
    exor(
        and(s(6),i3),
        exor(
            and(i0,i3),
            exor(
                and(i0,and(i1,i2)),
                exor(and(i0,and(i1,i4)),and(i0,and(i2,i4)))
            )
        )
    )
).
255 [para_into,239.1.2.1.1,24.1.1,demod]
GATE(
    ovfl,
    exor(
        and(s(4),i3),
        exor(
            and(s(5),i3),
            exor(
                and(i0,i3),
                exor(
                    and(i0,and(i1,i2)),
                    exor(and(i0,and(i1,i4)),and(i0,and(i2,i4)))
                )
            )
        )
    )
).
341 [para_into,255.1.2.1.1,22.1.1,demod]
GATE(
    ovfl,
    exor(
        and(s(5),i3),
        exor(
            and(i0,i3),
            exor(
                and(i0,and(i1,i2)),
                exor(
                    and(i0,and(i1,i4)),
                    exor(and(i0,and(i2,i4)),and(i1,and(i2,i3)))
                )
            )
        )
    )
).
359 [para_into,341.1.2.1.1,23.1.1,demod]
GATE(
    ovfl,
    exor(
        and(s(1),i3),
        exor(
            and(s(2),i3),
            exor(
                and(i0,i3),
                exor(
                    and(i0,and(i1,i2)),
                    exor(
                        and(i0,and(i1,i4)),
                        exor(and(i0,and(i2,i4)),and(i1,and(i2,i3)))
                    )
                )
            )
        )
    )
).
427 [para_into,359.1.2.1.1,19.1.1,demod]
GATE(
    ovfl,
    exor(
        and(s(2),i3),
        exor(
            and(i0,i3),
            exor(
                and(i0,and(i1,i2)),
                exor(
                    and(i0,and(i1,i4)),
                    exor(
                        and(i0,and(i2,i4)),
                        exor(and(i1,and(i2,i3)),and(i1,and(i3,i4)))
                    )
                )
            )
        )
    )
).
434 [para_into,427.1.2.1.1,20.1.1,demod]
GATE(
    ovfl,
    exor(
        and(i0,i3),
        exor(
            and(i0,and(i1,i2)),
            exor(
                and(i0,and(i1,i4)),
                exor(
                    and(i0,and(i2,i4)),
                    exor(
                        and(i1,and(i2,i3)),
                        exor(and(i1,and(i3,i4)),and(i2,and(i3,i4)))
                    )
                    )
                )
            )
        )
    ).
    462 [ur,71,55,110,demod]
    -GATE(
        x,
        exor(
            and(i0,i3),
            exor(
                and(i0,and(i1,i2)),
                exor(
                    and(i0,and(i1,i4)),
                    exor(
                        and(i0,and(i2,i4)),
                        exor(
                            and(i1,and(i2,i3)),
                            exor(and(i1,and(i3,i4)),and(i2,and(i3,i4)))
                        )
                    )
                )
            )
        )
    ).
    463 [binary,462.1,434.1].
```

This clause is interpreted as follows: The contradiction between clause 463 and 434 proves that the circuit, as implemented with a random variable order of the input signals, is an instance of the specification.

```
------------ end of proof -------------
```