29

ANL/DIS/TM-47

Identification of Functional Components in Combinational Circuits

Decision and Information Sciences Division

Argonne National Laboratory

Operated by The University of Chicago,
under Contract W-31-109-Eng-38, for the

United States Department of Energy

Argonne National Laboratory

This technical memo is a product of Argonne's Decision and Information Sciences (DIS) Division. For information on the division's scientific and engineering activities, contact:

Director, Decision and Information
Sciences Division
Argonne National Laboratory
Argonne, Illinois 60439-4832
Telephone (630) 252-5464
http://www.dis.anl.gov

Presented in this technical memo are preliminary results of ongoing work or work that is more limited in scope and depth than that described in formal reports issued by the DIS Division.

Publishing support services were provided by Argonne's Information and Publishing Division (for more information, see IPD's home page: http://www.ipd.anl.gov/).

Disclaimer

Reproduced directly from the best available copy.

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831; prices available from (423) 576-8401.

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, VA 22161.

ANL/DIS/TM-47

Identification of Functional Components in Combinational Circuits

by T.E. Doom,* J.L. White,* G.H. Chisholm, and A.S. Wojcik*

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

January 1998

Work sponsored by the United States Department of Defense

*Doom, White, and Wojcik are affiliated with the Department of Computer Science, Michigan State University, East Lansing, Michigan.

This report is printed on recycled paper.

CONTENTS

CONTENTS (Cont.)

FIGURES

FIGURES (Cont.)

TABLES

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contributions of Steve Eckmann and Ken Dritz. Their comments and insight have been invaluable in this research.

IDENTIFICATION OF FUNCTIONAL COMPONENTS IN COMBINATIONAL CIRCUITS

by

T.E. Doom, J.L. White, G.H. Chisholm, and A.S. Wojcik

ABSTRACT

Identifying the subcircuits in a detailed circuit description is a fundamental operation in both circuit validation and design recovery. Existing identification techniques rely on finding an exact match for a subcircuit structure within the description. These techniques fail to identify subcircuits that are functionally equivalent but have been obfuscated because a different technology is being used or because the design has been optimized. This report presents a mechanism for identifying subcircuits that are functionally equivalent, irrespective of obfuscating details. It also describes the initial progress made in transforming detailed circuit descriptions into corresponding descriptions based on subcircuits. Such progress depends on enumerating all of the candidate subcircuits within the original detailed description and functionally matching each candidate. The report presents unique solutions for reducing the amount of computation needed for this enumeration.

1 INTRODUCTION

This report is concerned with the reverse engineering (RE) of digital circuits (i.e., developing a functional understanding of existing digital circuits). The goal of RE is to completely transform the description of a digital circuit system from a low level (such as a flat netlist) to a level high enough to be easily understood by a redesign engineer. The first step in such an approach is to extract functional information from the digital circuit descriptions.

Reverse engineering may be viewed as the antithesis of design automation (DA). The goal of DA is to transform the description of a digital circuit from a higher level to a lower level. The DA literature focuses on the computer tools used to do this. Synthesis and layout tools are notable examples.

The DA literature does not currently contain a significant body of work that addresses the RE goal. Preliminary work on extracting functional information from a combinational circuit to verify layout design and supply feedback during the transformation from layout design to logic design is addressed in Ohmura et al. (1990), but, as indicated in that paper, such methods require "additional knowledge" that is not necessarily available in a flat netlist.

In our initial attempts in RE, we started with a detailed description of a digital circuit and transformed that to a specification of its logical function — a transformation from the silicon level to the flat netlist level (see Table 1). The current goal is to develop techniques for determining the modular specification from such a netlist. The first step is to identify common high-level logical functional components within the netlist, such as arithmetic logic units (ALUs), adders, and multiplexers. By identifying such functions within a circuit, we can reduce the complexity of producing functional descriptions and provide tags, identifying data lines, control lines, and additional information that might be useful in specifying the design at higher levels.

1.1 STATEMENT OF PROBLEM

The transformation of a gate-level netlist describing a circuit into a modular-level representation describing the circuit in terms of functional components (such as ALUs and multiplexers) and glue logic is the focus of this phase of the RE Project (Eckmann and Chisholm 1997). This report refers to this problem as the module identification (MI) problem.

- **Definition 1: Module identification (MI) problem**. Given a netlist, identify all subcircuits (clusters) that perform the function of a standard library

TABLE 1 Description of Specification Levels

Specification Level	Description of Function[a]
Design intent	Natural language
Behavioral	Full (VHDL behavioral)
Functional	Mathematical
Modular	Library level (VHDL structural)
Flat netlist	Gate level
Silicon	Transistor level

[a] VHDL = very-high-speed integrated circuit (VHSIC) hardware description language. VHDL is a large, high-level VLSI design language with Ada-like syntax that meets the U.S. Department of Defense standard for hardware description (IEEE 1076).

module. The preliminary approach taken to solve the MI problem consists of solving two subproblems:

- **Candidate subcircuit enumeration (MI-Enum) problem**. Identify clusters of gates within the netlist that may compose a functional component.

- **Subcircuit identification (MI-ID) problem**. Identify a functional component equivalent to the cluster by proving semantic equivalence between the cluster function and some pattern function representing a functional component.

1.2 ASSUMPTIONS

Our preliminary research is directed toward solving basic problems in a tractable (i.e., doable) amount of time. Therefore, for purposes of our initial work, the circuit being reverse engineered is assumed to be an unoptimized implementation. Thus, our approach relies on the following assumptions:

1. The cluster function is assumed to have the same number of inputs and outputs as the pattern function. Inputs that are bridged or "stuck at" (i.e., never change; see Section 2.8) must be represented as such and not optimized or reduced. No output can be ignored; every output of any library entity must either be a primary output or used as the input at some other point in the circuit.

2. The cluster function and pattern function must match exactly; "don't care" sets are not considered. Future extensions that handle don't care conditions are considered in Sections 3.2 and 6.2.2.

The results presented in this report are further restricted to identifying synchronous combinational components with no loops or other timing issues.

2 BACKGROUND AND PREVIOUS WORK

This section of the report covers some common terms, major issues, and published techniques related to the identification of logical functions as required for RE. This overview is not comprehensive; it is the result of preliminary research in this area. The references cited contain additional details.

2.1 STRUCTURAL MATCHING

Previous approaches to this problem relied on the discovery of subgraph isomorphisms to identify subcircuits (Bochner 1988, Luellau et al. 1984, Ohlrich et al. 1993). Although they are useful in such applications as converting a transistor netlist into a gate netlist, techniques that rely on exact structural matching (syntactic algorithms) have limited usefulness when applied to higher levels of design, since high-level components have many valid implementations.

Syntactic techniques have been successfully used to identify isomorphisms between the structures in a circuit description and those in a particular implementation (or set of implementations) of a high-level library entity. The advantage of structural matching is that it is exceptionally efficient, much more so than any more "complete" solution to the MI problem. However, it also has significant drawbacks that cause it to fail as a complete solution to the general MI problem.

A syntactic algorithm can identify only the implementations of a functional component that are contained in its library; thus, nonstandard or intentionally obfuscated implementations are never recognized. Furthermore, any optimization that modifies the implementation of the entity (such as optimizations for don't care conditions) makes the entity unfit for recognition by structural techniques. Structural matching cannot reliably recognize all functional components that exist in a circuit.

Nevertheless, any solution to the general MI problem should include the use of structural matching to recognize standard implementations of functional components within the circuit. When such exceptionally efficient syntactic techniques are applied first, the effective complexity of a circuit can be significantly reduced before more complex approaches are applied.

2.2 SUBGRAPH ENUMERATION

For ease of representation and manipulation, the circuit being reengineered can be represented as a directed graph. Many graph partitioning algorithms have been specifically modified to operate upon such circuit nets. The goal of these algorithms is generally "to divide a system specification into clusters such that the number of intercluster connections is minimized" for use in circuit board layouts (Alpert and Kahng 1995).

Although the problems appear to be similar, several factors distinguish the MI-Enum problem from the traditional partitioning problem. Partitioning implies that the clusters are disjoint, but that assumption cannot be made for the MI-Enum problem. Several modules may share functionality, so they may overlap and share gates. Relying on a strict partitioning algorithm could result in unidentified modules.

Further, because the problem of determining the perfect partition for given constraints is NP-complete,[1] partitioning techniques are generally approximation algorithms. In solving the semantic equivalence problem, all subcircuits whose semantics exactly match those of the high-level module must be identified. An approximate subcircuit will not suffice.

Another important difference between standard partitioning approaches and the approach to the MI-Enum problem is that standard approaches generally use heuristics that gradually improve the partition. Because no method is currently available for judging whether a given subcircuit is semantically close to the high-level module that is being sought, it is not possible to select likely subcircuits and build from them.

2.3 THE EQUIVALENCE PROBLEM

Consider some subcircuit (or subgraph) of a combinational circuit. Such a subcircuit has $|\vec{i}|$ inputs such that $\vec{i} = \langle i_1, \ldots, i_{|\vec{i}|} \rangle$, $|\vec{o}|$ outputs such that $\vec{o} = \langle o_1, \ldots, o_{|\vec{o}|} \rangle$, and a vector of Boolean functions or partial functions (the cluster function) that determines the relationships among them:

$$\vec{F}(\vec{i}) = \left\langle f_1(\vec{i}), \cdots, f_{|\vec{o}|}(\vec{i}) \right\rangle . \tag{1}$$

[1] NP-complete = nondeterministic polynomial time complete: A set or property of computational decision problems that is a subset of NP (i.e., can be solved by a nondeterministic Turing Machine in polynomial time), with the additional property that it is also NP-hard. Thus, a solution for one NP-complete problem would solve all problems in NP. Many (but not all) naturally arising problems in class NP are in fact NP-complete. There is always a polynomial-time algorithm for transforming an instance of any NP-complete problem into an instance of any other NP-complete problem. Therefore, if you could solve one, you could solve any other by transforming it to the solved one. The first problem ever shown to be NP-complete was the satisfiability problem. Another example is Hamilton's problem.

Likewise, for any high-level component with inputs $\vec{x}$ and outputs $\vec{y}$, there exists a vector of Boolean functions (the pattern function) that describes its behavior:

$$\vec{G}(\vec{x}) = \left\langle g_1(\vec{x}), \ldots, g_{|\vec{y}|}(x) \right\rangle . \tag{2}$$

The following two bijections — π_I, the input permutation function, and π_O, the output permutation function — are defined as follows:

$$\pi_I : \left\{ i_1, \cdots, i_{|\vec{i}|} \right\} \rightarrow \left\{ x_1, \cdots, x_{|\vec{x}|} \right\} \tag{3}$$

$$\pi_O : \left\{ f_1, \cdots, f_{|\vec{o}|} \right\} \rightarrow \left\{ g_1, \cdots, g_{|\vec{y}|} \right\} . \tag{4}$$

- **Definition 2: *PP*-equivalent.** Two vectors of Boolean functions $\vec{F}$ and $\vec{G}$ are input-permutation, output-permutation equivalent (*PP*-equivalent) if bijections exist such that:

$$\forall_k, 1 \le k \le |\vec{o}|, f_k(\vec{i}) = \pi_O(f_k)\left(\pi_I(\vec{i})\right). \tag{5}$$

  Single-output functions (often dealt with in technology mapping problems) are simply referred to as being permutation equivalent (*P*-equivalent). We can now define semantic equivalence for combinational designs.

- **Definition 3: Semantic matching.** Two combinational designs D_1 and D_2 with corresponding vectors of Boolean functions $\vec{F}$ and $\vec{G}$ are semantically equivalent if $\vec{F}$ and $\vec{G}$ are *PP*-equivalent. The input bijection π_I and the output bijection π_O under which $\vec{F}$ and $\vec{G}$ are *PP*-equivalent describe the semantic matching between D_1 and D_2.

2.4 BINARY DECISION DIAGRAMS

- **Definition 4: Binary decision diagram (BDD).** A BDD (Bryant 1985) is a directed acyclic graph consisting of two types of nodes. A nonterminal node v is represented by a 3-tuple:[2] $\langle index(v), child_l(v), child_r(v) \rangle$, where $index(v) \in$

[2] 3-tuple = a data object containing three components; a 2-tuple contains two.

$\{0,1, \ldots, n-1\}$ and $child_l(v)$ and $child_r(v)$ are themselves nodes of the BDD. A terminal node v is represented by a 2-tuple $\langle index(v), value(v)\rangle$, where $index(v) = n$ and $value(v) \in \{0.1\}$. A BDD is ordered if for every nonterminal node v, $index(v) < index(child_l(v))$ and $index(v) < index(child_r(v))$. A BDD is reduced if there is no nonterminal node v such that $child_l(v) = child_r(v)$ (redundant nodes) and there are no two nonterminal nodes u and v such that $child_l(u) = child_l(v)$ and $child_r(u) = child_r(v)$ (isomorphic nodes).

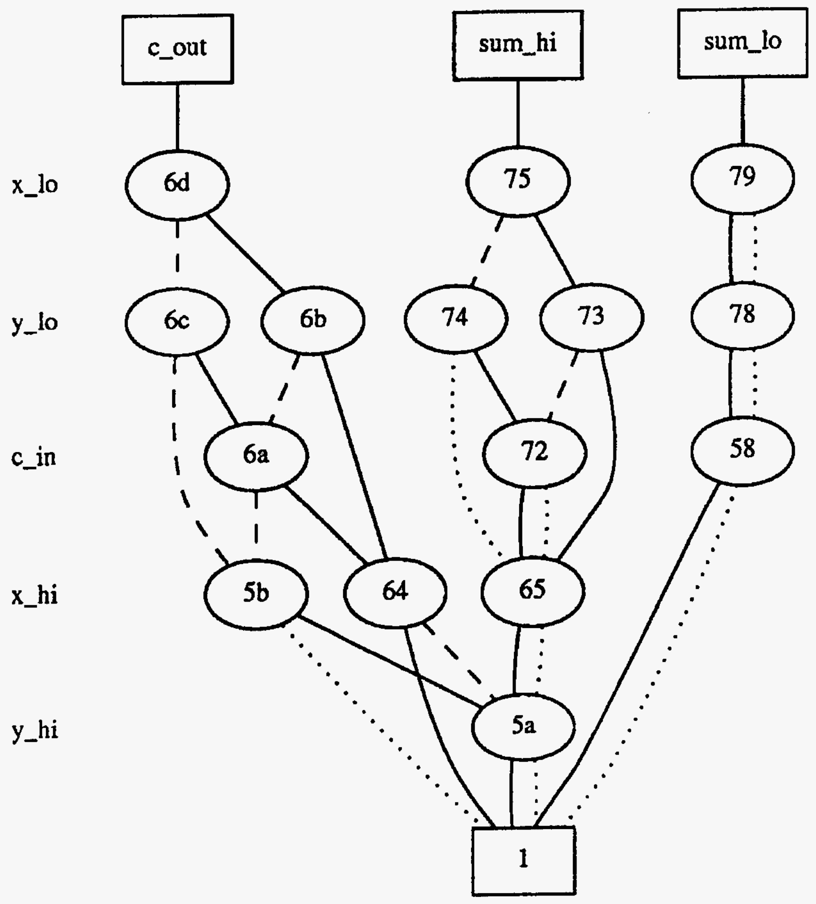
A BDD represents a Boolean function as a directed acyclic graphic. Figure 1 illustrates a reduced, ordered BDD (ROBDD) for the function of a two-bit full adder. Terminal nodes (shown as boxes) have no children and contain values corresponding to possible outputs of the function. Nonterminal, or decision, nodes (shown as circles) are labeled by a variable identifier and possess a labeled, outgoing arc for each value the Boolean variable may take: the "then" arc is taken when the decision variable has the value 1, and the "else" arc is taken when the decision variable has the value 0. A decision node with no incoming arc is called a root node. The functional value for any variable assignment is determined by traversing the path from the root node to a terminal node by following the appropriate branch at each decision node.

2.5 FACTORIAL PERMUTATION

Although testing the equivalence of two single-output functions represented as ROBDDs can be achieved in constant time (Bryant 1985), such testing requires that the correspondences between the input variables be clearly identified. Because input and output variable correspondences are not generally available, the straightforward method for determining if two multiple-output functions are *PP*-equivalent is to test for equivalence over the set of $|\vec{i}|! \cdot |\vec{o}|!$ possible pairs of bijection functions (i.e., over all input and output permutations). When inputs number more than seven to nine, the straightforward permutation technique is computationally intractable.

2.6 LOGIC VERIFICATION

In logic verification, a specification describing some functional behavior is compared with a circuit implementation of that function to prove equivalence. Verification techniques that can deal with problems involving large numbers of inputs, sequential behavior, and significant numbers of intermediate gates do exist. However, such techniques require that correspondences between the implementation and specification be known (Lai et al. 1992). Since we cannot assume such correspondences are available for the MI problem, verification techniques are generally not applicable.



Boxes indicate terminal nodes, circles indicate decision nodes, and decision nodes with no incoming arcs indicate root nodes. The label of each node is a unique random name; all nodes of the same level correspond to the same variable, whose name is shown at the left of the diagram. Solid lines indicate then arcs. Dashed lines indicate else arcs. Dotted lines indicate complemented else arcs and negate the value of the terminal.

FIGURE 1 Multirooted Binary Decision Diagram That Represents the Function Performed by a Two-Bit Adder

2.7 LOGIC SYNTHESIS

2.7.1 Technology Mapping

Technology mapping (also known as cell-library binding) is part of the synthesis process that must be used to transform logic representations into interconnections within a set of implementation-dependent cells. Technology mapping is used to create cost-optimized implementations for some logic function or Boolean network in a particular style in terms of some library of building blocks (cells). The detection of the equivalence of these Boolean functions to cells, referred to as Boolean matching, is well studied (Benini and Micheli 1997).

In many ways, the problem of determining the equivalence between a combinational circuit and a library of high-level entities is similar to the problem of Boolean matching. Boolean matching algorithms are designed to efficiently match small (fewer than six inputs) single-output clusters with a component of their cell libraries that implements the function at the least cost. However, although a general solution to the equivalence problem must be able to efficiently match functions with any number of inputs and outputs, it also needs to be concerned with a single (although possibly multiple-output) pattern function rather than an entire library of such functions. The goal of semantic matching is not to find the "best" implementation of a function from a set of possible implementations but to identify equivalence and variable correspondences between a particular subcircuit and a particular high-level component. It appears that no suitable solution to this problem has been reported on in the literature.

2.7.2 Boolean Signatures and Filters

A signature of a Boolean function is a unique and characteristic representation of some property of the function. Although two otherwise unrelated functions can have the same signature, having equal signatures is a necessary condition for equivalence matching. Functions that share a signature are said to share a signature class.

A signature function is a function that takes a generic function as an input and returns a characteristic signature for that input function. The value of a signature function must be determined only by the behavior of the generic function; variable order, variable labels, and random elements may not be used as part of the determination.

Boolean signatures have been used successfully to increase the efficiency of Boolean matching algorithms (Mailhot and Micheli 1993). Since sharing a signature class is a necessary condition for equivalence, the matching of signature functions can be used to eliminate functions from equivalence consideration. Functions that do not have matching signature characteristics can be filtered from the search space since they cannot be equivalent; thus, they do not need to be considered further in the testing process. The primary limit to the effectiveness of such filtering is the complexity cost of the signature function.

The use of filtering techniques in Boolean matching (Mailhot and Micheli 1993) has resulted in the discovery of a wide variety of signature tests by various researchers. Using signatures as filters to eliminate some permutations from consideration can appreciably reduce the complexity of a *P*-equivalence check. There are two classes of signatures: those that provide information on the behavior of input variables (input signatures), and those that provide information on the behavior of output variables (output signatures). Some of these signatures are discussed briefly here because they offer directions for future research. Lai et al. (1992) contains additional details.

For any function $f(x_1, ..., x_n)$, represented by BDD G of size $|G|$, the following signatures are defined:

- Cardinality of dependence set: The dependence set of a function consists of only those input variables that have an effect on its value. Thus,

$$Dep(f) = \left\{x_i \middle| f(\cdots, x_{i-1}, 0, x_{i+1}, \cdots) \neq f(\cdots, x_{i-1}, 1, x_{i+1}, \cdots)\right\} . \tag{6}$$

  The output signature $F_{dep}(f) = |Dep(f)|$ can be computed in $O(|G|)$ time by using a BDD-based algorithm (Lai et al. 1992). This signature is particularly useful when output-permutation equivalence is being determined. Outputs can be permuted only with outputs of the same dependence set cardinality. Functions that do not have the same number of outputs in each cardinality class cannot be equivalent.

- Cardinality of on-set: The on-set of a function consists of all input assignments that produce a true (on) output. The cardinality of the on-set is one of the more effective Boolean signature functions.

$$F_{on} = \left|\left\{\vec{x} \middle| f(\vec{x}) = 1\right\}\right| . \tag{7}$$

  This signature can be computed in $O(|G|)$ time by using the algorithm presented in Lai et al. (1992). This signature can be used to reduce both the number of output matches between multiple-output functions and the number of input permutations.

- Unateness of input variables: A binate variable is present in both its complemented and uncomplemented forms in the minterm (i.e., minimum term) expression for a function. A unate variable is present in either its complemented or uncomplemented form, but not both. Thus the unateness of each input variable can be used as a signature $F_{unate}(f,x) = \{binate, positive\ unate, negative\ unate\}$. For two functions to be equivalent, corresponding input variables must have similar unateness properties.

  The unateness of input variables can also be used as an output signature. For each output, count the number of binate, positive unate, and negative unate input variables that occur in the function's minterm expression. Its matching function must share these same sums. Computing the unateness of each input variable for each output function is an $O(|G|^2)$ operation, which may be too expensive for the MI problem.

- Symmetry class of input variables: Two variables are symmetric if they can be interchanged without changing the value of the function. Thus *xi* and *xj* are symmetric if and only if $f(\ldots, xi, \ldots, xj, \ldots) = f(\ldots, xj, \ldots, xi, \ldots)$. The input variables can be partitioned into symmetry classes that act as a signature for each output function. In addition, input variables can be matched only to input variables that have equivalent symmetry classes over all output functions. Symmetry computation requires an $O(|G|^2)$ operation for each pair of inputs for each function and is probably too expensive for the MI problem. Although they can be effective for small cells in Boolean matching problems, symmetry classes are not always an effective signature on high-level entities, many of which have few symmetries.

  High-level entities, however, often posses group symmetries. When some group of input variables can be interchanged with a disjoint group of input variables without changing the value of the function, the groups of variables are said to be group symmetric. Group symmetries are also signature functions that might be particularly effective on high-level entities representing arithmetic functions. Calculating group symmetries, however, is not a trivial operation.

- Sizes of hamming distance *k*: This signature is defined to be the set of cardinalities to the $n-1$ sets of pairs of one-points of f whose Hamming distance is k, $0 < k < n$. This signature can be computed in $O(n \cdot |G|)$ by using the algorithm presented in Lai et al. (1992). This signature is quite effective, but the complexity of the computation significantly limits its usefulness.

2.8 OPTIMIZED CIRCUITS

2.8.1 High-Level Design and Synthesis

During both high-level design and the synthesis process, logic functions may be modeled from available units that "almost" fit the necessary function. The actual cells used depend on the specific cell library and the cost metrics associated with the binding processes.

Three common techniques used in high-level design that complicate the RE Project include bridged inputs, stuck-at inputs, and ignored outputs (Mailhot and Micheli 1993). When two (or more) inputs to a library cell are connected to the same input line, such cell inputs are bridged. When a library input is tied to ground (power), the input is stuck at 0 or stuck at 1 (either logical 0 or logical 1). Furthermore, some outputs of the library entity may not be being used. High-level designs that incorporate these features may be difficult to identify in netlists because of local optimizations.

When a logical function with bridged or stuck-at inputs is mapped, its implementation will take advantage of these facts to greatly simplify the details of the design. Furthermore, the number of inputs and outputs of a cluster may not correspond to the number of inputs and outputs of the pattern function that it represents in these three cases. Matching clusters to high-level entities in which such techniques were used remains a complex operation.

Another point to remember is that circuit optimizations may cause the intermediate functions that are traditionally performed by several distinct library units to occur in a single, shared cluster. Therefore, care must be taken to note that not all outputs of a cluster are necessarily outputs of its corresponding library entity, and that an identified cluster cannot always be simply replaced by the high-level entity identified.

It is beyond the scope of our present work to find high-level entities corresponding to clusters that have stuck-at inputs, bridged inputs, or unused outputs. The number of inputs and number of outputs must be equivalent for detection to be successful (Section 1.2).

2.8.2 Don't Care Sets

Consider a circuit with primary inputs $\vec{x}$, primary outputs $\vec{z}$, and the vector of functions $\vec{H}(\vec{x}) = \vec{z}$ (as defined in Equation 1), which determines the relationship between them. Also consider some cluster within this circuit with inputs $\vec{i}$, outputs $\vec{o}$, and vector of functions $\vec{F}(\vec{i}) = \vec{o}$, which similarly determines the behavior of the cluster circuit.

In a completely specified circuit, it is possible to determine the vector of functions $\vec{P}(\vec{x}) = \vec{i}$ (which determines the cluster inputs for any given primary input set) and the function $\vec{Q}(\vec{x},\vec{o}) = \vec{z}$ (which determines the value of the primary outputs on the basis of the value of the cluster outputs and the behavior of the rest of the circuit). These relationships fully describe the environment around the multioutput cluster.

The input controllability don't care set (CDC) for the cluster includes all input conditions that are never produced by the environment (Benini and Micheli 1997). Thus the CDC is defined as follows:

$$CDC = \left\{ \vec{i} \,\middle|\, \vec{i} \text{ is not in } Range(\vec{P}(\vec{x})) \right\} . \tag{8}$$

The output observability don't care set (ODC) for each output of the cluster denotes all input patterns that produce situations in which the output of the cluster is not observed by the environment (Benini and Micheli 1997). Effectively, the ODC set contains all cluster inputs for which the values of the primary outputs do not depend upon the output(s) of the cluster. In mathematical terms:

$$ODC = \left\{ \vec{i} \,\middle|\, \forall \vec{x} \text{ such that } \vec{P}(\vec{x}) = \vec{i},\ \forall \vec{o} \in Range(\vec{F}),\ \vec{Q}(\vec{x},\ \vec{o}) = \vec{H}(\vec{x}) \right\} . \tag{9}$$

These don't care conditions produce degrees of freedom available within the cluster function. Functions within these degrees of freedom will produce behaviors that the environment cannot distinguish from each other. During the selection process, a function that is close to but not identical to the logical function specified by the cluster may be chosen to implement that logical function. That is, some vector function $\vec{F}'$ may be chosen such that:

$$\forall \vec{i} \in \left\{ Range(\vec{P}) - CDC - ODC \right\},\ \vec{F}(\vec{i}) = \vec{F}'(\vec{i}) . \tag{10}$$

The actual function implemented will be one of the functions that obeys these conditions and has a low associated cost. These kinds of don't care optimizations are common in sophisticated synthesis algorithms as well as in hand-optimized designs. The RE Project requires that one be able to determine the high-level function effectively performed by a cluster, even if the actual function performed by the cluster does not behave as expected under the don't care set.

In this initial work, we do not consider problems that contain don't care optimizations. We hypothesize, however, that by identifying the don't care set for a particular cluster function, we can "mask" both the cluster function and the pattern function before checking for *P*-equivalence. Approaches to this problem are discussed in Section 6.

3 CANDIDATE SUBCIRCUIT ENUMERATION

Before all of the library entities within the circuit can be located, all of the potential library entities within the circuit must be located. This is a very challenging task. A section of the circuit that corresponds functionally to a library entity will not necessarily appear to be similar in any structural way, including its size, order, connectivity, or positioning. This lack of similarity impedes any attempt to guide the subgraph generation with meaningful heuristics. To ensure that all potential library entities are located, all possible subgraphs of the initial circuit netlist must be generated.

In a completely connected graph, the number of (not necessarily disjoint) subgraphs that could exist is

$$\sum_{i=1}^{n} C_1^n \ ,$$

where n is the order of the graph. Digital circuits are never completely connected, so the number of subgraphs will be significantly smaller. However, the above equation does give an upper bound and succinctly imparts the enormity of this problem.

To guarantee that all library entities within the graph have been located, we must generate all of the possible matches. As stated above, the order, size, and configuration of a subgraph may differ from those of a functionally equivalent library module. Therefore, we must focus on the attributes that can be assumed to be true about a subgraph and its equivalent library module. Our initial method takes advantage of the following facts:

1. A subgraph representing a function must have a number of inputs and outputs equal to those of the corresponding library module.

2. The inputs must fully define the outputs.

3. A subgraph must be a connected graph.

This section discusses a method used to enumerate all candidate subcircuits of the circuit. The number of subgraphs within a directed graph is exponential, and generating all of these subgraphs cannot be accomplished in polynomial time. However, because the algorithms presented here take advantage of the information that we gain from knowing that the graph represents a digital logic circuit, they can generate all candidate subcircuits quickly for small circuits. Improvements that allow larger circuits to be handled within a reasonable amount of time are discussed in Sections 3.3 and 6.1.

This method is guaranteed not to miss any potentially important subcircuits. It operates efficiently enough that when a few heuristics are applied, it can handle moderately sized circuits in a reasonable amount of time.

3.1 PRELIMINARY DEFINITIONS

In this report, the term "circuit'' refers to a digital circuit. When a graph representing a circuit or subcircuit is being discussed, its gates are called nodes, and its connections are called arcs (Christofides 1975). As a cluster of gates is formed, it is simply called a subgraph until all of its constituent gates are fully specified. At this point, it is called a valid subgraph.

- **Definition 5: Order.** The order of a graph *G* is the number of nodes within *G*.

- **Definition 6: Parent.** The parent of a gate *g* is a gate whose output is an input to gate *g*. The children of a gate *g* are those that have the output from *g* as an input.

- **Definition 7: Fully specified.** A gate is fully specified if and only if either all or none of its parents are contained within the subgraph within which it is contained.

- **Definition 8: Input.** A gate is an input to the subgraph in which it is contained if and only if none of its parents are also in that subgraph.

- **Definition 9: Valid subgraph.** A subgraph *H* represents a valid subgraph if and only if it is connected and each gate in *H* is fully specified.

- **Definition 10: Forward arc.** A forward arc is an arc from a parent to a child. A backward arc is an arc from a child to a parent.

- **Definition 11: $H' = H + V$.** The notation $H' = H + V$ indicates that a new subgraph H' is created by adding a neighboring node v onto an existing subgraph H. The arc between v and H is also added, resulting in the induced subgraph H'.

3.2 SUBCIRCUIT ENUMERATION

Generating all of the subgraphs of a graph for enumeration is a lengthy process that can result in an exponential number of subgraphs. Each of the following algorithms operates by expanding subgraphs. When a subgraph *H* is expanded, a neighboring node is added to *H* such that the resulting graph *H′* is also an induced subgraph of the original graph. We first present a naïve algorithm, Algorithm 1, that generates all of the subgraphs of the graph representing the circuit. Algorithm 1 provides the basis for explaining two algorithms that are more efficient.

Algorithm 2 takes advantage of the fact that any subgraphs that need to be investigated will also be valid subgraphs. A subgraph that does not represent a valid subgraph cannot possibly be semantically equivalent to a known high-level module, so it is unnecessary to generate the subgraph. Algorithm 2 creates only subgraphs that are valid subgraphs, thus remarkably reducing the number of extraneous subgraphs. The remaining subgraphs that are generated are duplicates of already existing subgraphs. Algorithm 3 enforces an ordering on the gates, which reduces the number of duplicate subgraphs, although it cannot completely eradicate them.

All algorithms maintain two pools: *P* and *S*. *P* contains the subgraphs that are to be expanded, and *S* contains those that have already been expanded. When the algorithms terminate, *S* contains all of the subgraphs generated by the algorithm. Figure 2 represents a simple one-bit adder circuit that illustrates the three algorithms.

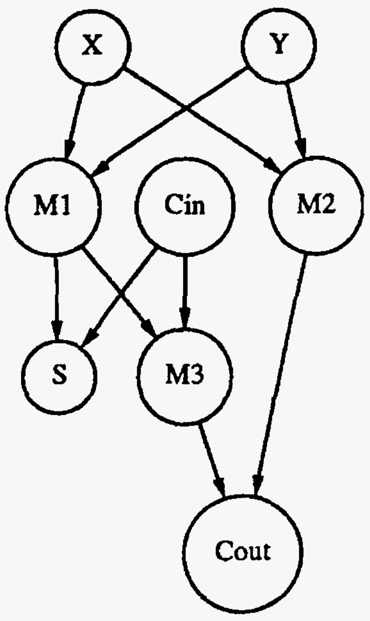


FIGURE 2 One-Bit Adder

3.2.1 Algorithm 1: Naïve Generation

Algorithm 1 begins by initializing a pool *P* of subgraphs, each consisting of a child-parent pair from the graph representing the original circuit, the graph *C*. Each of these pairs becomes an initial subgraph. The initial pool therefore consists of order two graphs, one created from each arc (connection) in the circuit. For each subgraph in the pool *P*, the external arcs are calculated. An external arc is an arc with one end point within the subgraph and one outside. The subgraph is duplicated and extended along each of its external arcs. This algorithm will terminate when every subgraph *P* in has been expanded and moved into *S*. For Figure 2, the initial pool of subgraphs generated by the naïve algorithm will contain 10 subgraphs, one for each arc in the graph, each containing the end points of an arc in the subcircuit.

3.2.1.1 Description

- Step 0: Initialize. Create empty pools *P* and *S*.
- Step 1: Generate initial subgraphs. For each node $v \in C$, create subgraphs such that each subgraph contains *v* and one of its children.
- Step 2: Expand pool. While *P* is not empty, examine subgraph $H \in P$.
  - Step 2.1: Expand subgraph. For each arc leaving from *H* to a node *v*, create a new subgraph $H' = H + v$. Move *H* to *S*.
  - Step 2.2: Add new subgraphs to pool. For each new subgraph, verify that it is not a duplicate subgraph and add H' to *P*.

3.2.1.2 Example

The initial subgraph shown in Figure 3 was formed by the arc connecting M1 and M3. The original graph, Figure 2, has five external arcs. Two are forward: S and Cout. Three are backward: X, Y, and Cin. Five duplicates of the M1–M3 subgraph are then created, and one of the external arcs is added to each. Five new subgraphs of order three are thus formed. They are tested to ensure uniqueness, then added to *P*. The algorithm then proceeds to the next subgraph in *P*.

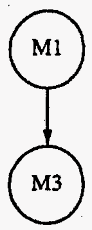


FIGURE 3 Subgraph with M1 and M3

3.2.2 Algorithm 2: Generation of Valid Subgraphs

If we know that graph C represents a logic circuit and that the only subgraphs needed are those that correspond to a valid subcircuit in the original circuit, we can make many improvements to the algorithm. To locate the candidate subcircuits, we are interested in only the valid subgraphs, because only they have the potential of matching a known module. Therefore, we need to generate only the subgraphs that represent valid subgraphs of the original circuit.

We made several modifications to the naïve algorithm to implement this change. The pool P was initialized with valid subgraphs. This was accomplished by creating a subgraph for each node that contains the node and its parents. The subgraphs were no longer extended along both the forward and backward arcs. This algorithm first extends along the forward arcs, picking up any backward arcs necessary to completely specify its internal nodes, thus ensuring that a valid subgraph is created. It also extends backward from its inputs, but instead of adding the parents individually, it simply adds all of the parents at once, because all are necessary to specify the node and the subcircuit.

By expanding only the valid subgraphs, we reduced the number of duplicate subgraphs created, but the new subgraphs generated are not necessarily valid subgraphs. To transform a subgraph into a valid subcircuit, we must fully specify each of its constituent gates. The missing parent or parents of any gate that is incompletely specified will be added to the gate until all the gates are fully specified and the subgraph represents a valid subgraph.

3.2.2.1 Description

- Step 0: Initialize. Create empty pools P and S.

- Step 1: Generate initial subgraphs. For every node $v \in C$ that is not an input to C, create a subgraph H containing H and its parents. Assign a label to H that corresponds to the highest index of the nodes in H.

- Step 2: Expand pool. While P is not empty, examine subgraph $H \in P$.

  - Step 2.1: Expand subgraph forward. For each forward arc from H to a node v, create a new subgraph $H' = H + v$.

  - Step 2.2: Expand subgraph backward. For each input v of H, create a new subgraph H' which contains a copy of H and all of the parents of v. Move H to S.

- Step 2.3: Ensure subcircuit validity. For subgraph *H'*, ensure the validity of the represented subcircuit by adding the nodes necessary to fully specify each node $v \in H'$.

- Step 2.4: Add new subgraphs to pool. For each new subgraph *H'*, verify that it is not a duplicate subgraph and add *H'* to *P*.

3.2.2.2 Example

The order two subgraph of Figure 3 does not represent a valid subgraph because only one of M3's parents is in the subgraph and therefore M3 is not fully specified. To create a valid subgraph from the M1–M3 subgraph, Cin must be included to fully specify M3, resulting in the subgraph shown in Figure 4. This subgraph represents a valid subgraph of the original circuit.

3.2.3 Algorithm 3: Ordered Generation of Valid Subgraphs

The algorithm to generate valid subgraphs generates a significant number of duplicate subcircuits because even when only valid subcircuits are expanded there is more than one way to grow a subgraph. For instance, the subgraph shown in Figure 5 can be grown by adding M2 to the subgraph in Figure 6 or by adding M1 to the subgraph in Figure 7.

To reduce the number of duplicates that are created, the nodes are ordered such that each node has a unique integer index that is higher than the indices of all of its parents. The node ordering for the original circuit is displayed in Figure 8. Rules can then be enforced dictating which nodes can be added to a subgraph when it is being expanded, thus preventing many duplicates from being created.

3.2.3.1 Description

- Step 0: Initialize.

  - Step 0.1: Initialize pools. Create empty pools *P* and *S*.

  - Step 0.2: Initialize circuit. Iterate through *C* in a breadth-first manner, labeling each gate with a unique integer index, such that its index is higher than the indices of its parents.

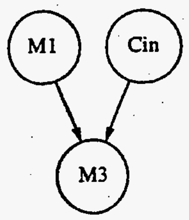


FIGURE 4 Subgraph with Cin, M1, and M3

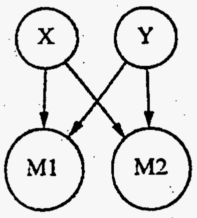


FIGURE 5 Subgraph with M1, M2, X, and Y

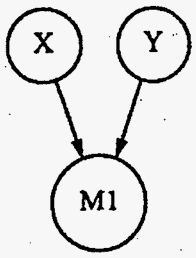


FIGURE 6 Subgraph with M1, X, and Y

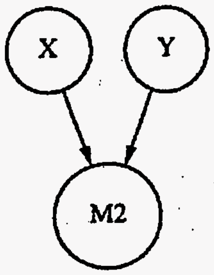


FIGURE 7 Subgraph with M2, X, and Y

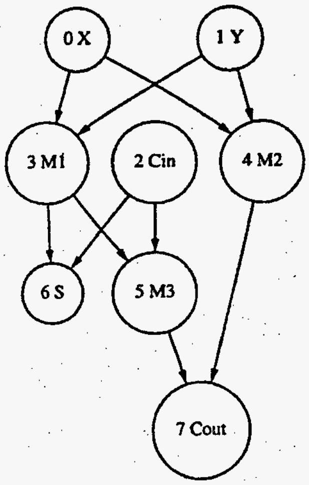


FIGURE 8 Original Circuit with Node Ordering

- Step 1: Generate initial subgraphs. For every node $v \in C$ that is not an input to C, create a subgraph H containing H and its parents. Assign label to H that corresponds to the highest index of the nodes in H.

- Step 2: Expand pool. While P is not empty, examine subgraph $H \in P$.

  - Step 2.1: Expand subgraph forward. For each node v adjacent to subgraph H with an index greater than the label of H, create a new subgraph $H' = H + v$.

  - Step 2.2: Expand subgraph backward. For each input v of H, create a new subgraph H' that contains a copy of H and all of the inputs to v. Move H to S.

  - Step 2.3: Ensure subcircuit validity. For subgraph H', ensure the validity of the represented subcircuit by adding the nodes necessary to fully specify each node $v \in H'$.

  - Step 2.4: Add new subgraphs to pool. For each new subgraph H', verify that it is not a duplicate subgraph and add H' to P.

3.2.3.2 Example

With the ordering now imposed on the creation of subgraphs, the subgraph in Figure 5 can be created only from the subgraph in Figure 6. The ordering displayed in Figure 8 indicates that the index of the graph in Figure 7 is 4, because that is the highest index of its nodes. Therefore, M1 may not be added because its index is not greater than the index of the graph. However, M2, with an index of 4, may be added to the graph in Figure 6 because the index of the graph is only 3, less than the index of M2.

3.3 GENERAL IMPROVEMENTS

As previously discussed, the number of subgraphs for reasonably sized circuits is unreasonably large. Therefore, a method was devised to significantly reduce the number of subcircuits generated by investigating slices of the circuit at a time.

- **Definition 12: Distance.** The distance between two gates within the circuit describes the number of connections traversed in traveling from one gate to the other.

- **Definition 13: Window.** A window refers to a collection of gates in which each gate is no more than distance *n* from another gate, where *n* is the size of the window.

The gates within the initial slice are constrained to be no more than *n* steps from the inputs of the graph. After that slice has been fully investigated (all subcircuits have been generated), the window is slid forward one step, so that the input gates are no longer under consideration, and the gates one level deeper in the graph are now part of the current window.

This method does not generate all of the subcircuits of the circuit; it generates only those circuits with a depth that is less than the size of the window. It is necessary to choose a depth that is large enough to usually generate alternate implementations of the library modules, yet small enough so that the number of subcircuits becomes manageable.

4 CLUSTER IDENTIFICATION

This section describes an algorithm for determining if a semantic match exists between a subcircuit and a high-level component. A general solution to the MI-ID problem requires the identification of high-level components that are more complex then those dealt with in Boolean matching but that lack the input/output correspondences between the logic design and the library components that verification techniques require. Since the function performed by a high-level component may be represented in any number of structural forms, we must identify the subcircuit by proving semantic equivalence (Eckmann and Chisholm 1997). Although semantic techniques are not limited to any particular level of circuit description or application, this report considers only the identification of high-level components from gate-level netlists.

4.1 INPUT SIGNATURES AND SUSPECT SETS

Our approach to the semantic matching problem uses signature information to reduce the number of input correspondences that must be considered. This is accomplished through the use of suspect sets.

As discussed in Section 2.7.2, a signature of a Boolean function is a unique, characteristic representation of some property of the function. The signatures that provide information regarding the behavior of a function's input variables are referred to as input signatures.

- **Definition 14: Signature class.** The signature values for any input signature function can be used to partition the function inputs into classes corresponding to their signature. Such a list of inputs is a signature class.

The following theorem is clear: Input correspondences between the pattern and cluster function can take place only between members of their respective signature classes that have equal signature values.

- **Definition 15: Suspect set.** A cluster input variable i_k's suspect set, S_{ik}, is the subset of pattern function $\vec{G}$'s inputs, $x_1, \cdots, x_{|\bar{x}|}$, that share a signature class with i_k under every input signature for which information is available.

Using suspect sets will allow us to significantly reduce the factorial search space associated with determining function equivalence.

4.2 VECTOR SIGNATURE

We introduce a new signature function that has proven to be an adequate initial filter for many problems. This signature takes advantage of the fact that the vector functions under consideration consist of multiple functions, each corresponding to a single output.

- **Definition 16: Unit vector.** A positive (negative) Boolean unit vector is a vector in which exactly one element has the value 1 (0) and all other elements have the value 0 (1).

- **Definition 17: Vector input signature.** For any vector of Boolean functions $\vec{F}(\vec{i}) = \vec{o}$, i_j's positive unit vector input signature is the sum of the function outputs (i.e., the cardinality of the on-set) when the positive unit vector with input i_j equal to 1 is applied.

$$F_{+vec}(i_j) = \sum_{n=1}^{|\vec{i}|} f_n(\vec{u}), \tag{11}$$

where $u_k = 1$ if and only if $k = j$.

The negative unit vector input signature is defined similarly.

- **Definition 18: Vector signature.** For any vector of Boolean functions $\vec{F}(\vec{i}) = \vec{o}$, the function's vector signature is an ordered set of $|\vec{i}|(x, y)$ pairs, in which each pair corresponds to an input i_j of $\vec{F}$ and x (y) represents the positive (negative) unit vector input signature.

Table 2 shows the results of applying the vector signature to the vector function of a four-bit ALU. The resulting vector signature is $\{2 \times (1, 7), 1 \times (2, 2), 1 \times (2, 5), 6 \times (2, 7), 3 \times (3, 5), 1 \times (6, 5)\}$.

4.2.1 Additional Vector Input Signatures

The signature classes determined by the vector input signature under the positive and negative unit vectors partition the set of input variables into several signature classes. This information can be used to create nonunit vector input signatures.

When any signature class contains a single member (that is, no other input shares its (x, y) signature value), a correspondence is clearly identifiable. The vector signature for the four-bit ALU shown in Table 2 has two signature classes with only a single member (the signature classes for *sel*3 and *m*). Recognizing correspondences for such variables is straightforward. A signature class with multiple members, however, does not differentiate among the inputs sharing the signature class. Such differentiation may be achieved through the use of additional vector signatures.

For each signature class, we create a set of vectors that must create a new set of vectors, which allows additional vector input signatures to be computed and may thus differentiate the inputs within the initial class. For each positive or negative unit vector taken over the set of inputs in a signature class under study, there are 2^p assignments of values to the distinguishable inputs of the other p signature classes. Each of these vectors may be applied to produce an additional vector input signature. These additional vectors can be applied to create more signature classes, allowing more precision in suspect sets. This process can be continued until all additional vectors have been exploited.

Consider a function $\vec{H}$ with seven inputs, *a* through *g* (Table 3). Let the inputs be partitioned by vector signature into three signature classes, as follows: (a, b) (c, d, e) (f, g). Consider the additional vectors that can be created for example function *H* that may be useful in differentiating input *a* from input *b*.

TABLE 2 Vector Input Signature for the TI 54181 Four-Bit Arithmetic Logic Unit[a]

Input Name	Vector Input Signature: Positive	Vector Input Signature: Negative
sel0	2	7
sel1	1	7
sel2	2	7
sel3	2	2
b3	2	7
a3	3	5
b2	2	7
a2	3	5
a1	3	5
b1	2	7
a0	2	5
b0	2	7
m	6	5
Cn′	1	7

[a] The positive and negative coordinate vector input signatures are shown for a four-bit ALU with selection inputs sel0–3, mode input m, carry input Cn′, and data inputs a0–3 and b0–3. The vector signature partitions the function inputs into five signature classes: ⟨1, 7⟩ = {sel1, Cn′}, ⟨2, 2⟩ = {sel3}, ⟨2, 5⟩ = {a0}, ⟨2, 7⟩ = {sel0, sel2, b3, b2, b1, b0}, ⟨3, 5⟩ = {a1, a2, a3}, and ⟨6, 5⟩ = {m}.

TABLE 3 Additional Vectors

a	*b*	*c*	*d*	*e*	*f*	*g*
0	1	0	0	0	0	0
0	1	0	0	0	1	1
0	1	1	1	1	0	0
0	1	1	1	1	1	1
1	0	0	0	0	0	0
1	0	0	0	0	1	1
1	0	1	1	1	0	0
1	0	1	1	1	1	1

4.2.2 Other Considerations

Vector signatures are an effective signature for multiple-output functions in which the number of inputs is not significantly larger than the number of outputs. Their effectiveness is not surprising when we consider that the number of outputs determines the size of the range of the signature function. (The range of the function is $|\vec{o}|^2$.)

An output vector signature can also be computed by considering the number of vectors for which each output has the value one under the set of vectors, including the one vector, zero vector, and positive and negative unit vectors. This use of the vector signature has not yet been fully explored.

Vector input signatures may also be useful in filtering the number of library entities that must be compared with the cluster. A nonunique key for each functional output can be created by concatenating the 1-sum of the zero vector, 1-sum of the one vector, sorted set of positive vector signatures, and sorted set of negative vector signatures. If the don't care set for the cluster is empty, these keys can be used as hashing keys to locate the set of library entities that must be tested. In this way, all library entities that may be equivalent to the cluster can be identified in time linear to the number of cluster outputs, regardless of the size of the entity library. If the don't care set for the cluster is not empty, a less efficient technique must be used.

When the don't care set is not empty, the functions must be normalized to their care sets. This can be accomplished by "masking" the outputs of the function to 0 under the don't care set. The key created by the vector signature for this normalized function can then be compared with the normalized function signatures for each library entity.

4.3 SEMANTIC MATCHING ALGORITHM

Let $\vec{F}(\vec{i}) = \vec{o}$ be the vector of Boolean functions for some subcircuit. Let $\vec{G}(\vec{x}) = \vec{y}$ be the vector of Boolean functions for a high-level component. Semantic equivalence and input/output correspondences between the subcircuit and the high-level component can be determined by the semantic matching algorithm described below.

4.3.1 Description

- Step 1: Create binary decision diagrams. Create BDDs for the outputs of each vector of Boolean functions.

- Step 2: Determine signature classes. Determine the vector signatures for $\vec{F}$ and $\vec{G}$ and partition each function's input variables into signature classes. If the signature classes and partition sizes are not equivalent, the functions cannot be equivalent.

- Step 3: Determine suspect sets. For each input i_j of the cluster function $\vec{F}$, create a suspect set S_j. The suspect set S_j is the subset of inputs of pattern function $\vec{G}$ that have the same signature as the signature of input i_j. Apply additional input signatures (Sections 2.7.2 and 4.2.1) to reduce suspect set size below threshold (Section 4.3.2) if possible.

- Step 4: Iterate though legal input correspondences. Eliminate all matchings that include a correspondence between a cluster function input i_j and any pattern function input that is not in S_j.

- Step 5: Determine legal output correspondences. Compare each pair of BDDs representing a substituted cluster function output and a pattern function output. If an unique output matching for each pair is determined, a legal correspondence has been identified.

4.3.2 Complexity

The technique presented in Section 2.5 requires $|\vec{i}|!|\vec{o}|!$ comparisons. Our algorithm requires a lot fewer comparisons.

Let n represent the cardinality of the largest input suspect set determined in Step 3. An upper bound on the number of legal input correspondences is $n!^{|\vec{i}|}$. As long as n is constrained to a reasonably small size (less than nine), it can be treated as a constant value c, and the input correspondence selection will be exponential in complexity: $O\left(c^{|\vec{i}|}\right)$. Reasonably small values of n can be achieved through pruning suspect set sizes by applying multiple signature values until all suspect set sizes fall below some threshold.

Such pruning is effective for most components except those having large numbers of symmetric inputs (which are indistinguishable from Boolean signatures). In such cases, however, any input matching will succeed for the symmetric inputs, which actually simplifies the process of proving semantic equivalence, because a correspondence will be identified very early in the execution of the algorithm.

Although BDDs are an efficient mechanism for representing the functionality of most components, they may become intractably large for certain functions under some (or all) variable orderings (Bryant 1985). Since we can indicate a "good" variable ordering for our pattern function library, we can eliminate most BDD-based concerns. If the BDD for any cluster function output exceeds the size of the largest BDD representing a pattern function output, we can immediately discard that input matching and discontinue BDD generation, since no legal correspondence can exist between functions that have BDDs of different sizes under the same variable ordering. Pathological functions (such as multipliers) that have no efficient BDD representation remain an open issue.

Since each cluster output BDD is tested against each pattern output BDD exactly one time in Step 5, the complexity of determining legal output correspondence is only $O\left(|\vec{o}|^2\right)$. Therefore, the overall complexity of this approach is $O\left(c^{|\vec{i}|}|\vec{o}|^2\right) = O\left(c^{|\vec{i}|}\right)$. This exponential algorithm is a significant improvement over factorial methods and makes semantic matching feasible for most components of reasonable size.

5 RESULTS

The algorithms discussed previously were implemented in C. Experiments were conducted on a Sun Ultra Enterprise 3000 running Solaris 2.5.1 with 256 MB of main memory and 879 MB of virtual memory. Experimental circuits were taken from the LGSynth93 benchmark suite (McElvain 1993).

5.1 SUBCIRCUIT ENUMERATION RESULTS

The complexity of the enumeration problem can be clearly seen in Table 4. A 15-gate, two-bit adder has 3,408 unique subgraphs (114 valid subgraphs). These subgraphs can be enumerated in acceptable time (0.1 second of CPU time) by any of the three algorithms. Notice, however, the abrupt increase in subgraphs that results from the addition of only seven gates. The 22-gate, three-bit adder generates 98,922 unique subgraphs (566 valid subgraphs).

The increase in subgraphs is not related solely to the number of gates, of course. It also depends on the number of wires in the circuit and their configuration. However, the number of gates does provide a good rough metric for predicting the number of unique subgraphs.

It is more difficult to predict the number of valid subgraphs. Every one of the valid subgraphs must be enumerated. Each of these subgraphs represents a possibly interesting subfunction of the circuit. For instance, in a three-bit adder, three subcircuits representing one-bit adders and two subcircuits representing two-bit adders will be enumerated, as will subgraphs that represent parts of two or more individual adders. Generating these subfunctions is important because they may be expanded to represent the functions that we are seeking, namely the one-bit adder or two-bit adder.

The three algorithms presented in Section 3.2 have been applied to several graphs to demonstrate the improvement provided by the latter two algorithms. Table 4 lists the number of gates, number of connections, and the connectivity ratio ($|\,connections\,|\,/\,|\,gates\,|$) of each of the circuits to be explored. It also lists the number of unique subgraphs and number of valid candidate subcircuits within the circuit.

Table 5 displays the results of applying the three algorithms to the circuits. Included are the total number of subgraphs generated and the amount of processor time consumed. Each algorithm will generate and identify all of the candidate subcircuits. The difference in the performance of the algorithms is a result of the number of duplicate subgraphs that were created during the generation.

TABLE 4 **Circuit Statistics**

Original Circuit	Circuit Function	No. of Gates	No. of Connections	Connectivity Ratio	No. of Unique Subgraphs	No. of Subcircuits
add1	1-bit adder	8	10	1.25	114	18
add2	2-bit adder	15	20	1.333	3,408	108
add3	3-bit adder	22	30	1.363	98,922	462
b1	Logic	25	36	1.44	95,707	901
z4ml	3-bit adder	30	42	1.4	NA[a]	4,360
cm138a	Logic	33	53	1.606	NA	29,362
x2	Logic	54	104	1.923	NA	38,364

[a] NA = not available; not computed because of complexity.

TABLE 5 Results of Candidate Subgraph Generation

	1: Naïve		2: Valid		3: Ordered		
Circuit	No. of Subgraphs	CPU Time (second)	No. of Subcircuits	CPU Time (second)	No. of Subcircuits	CPU Time (second)	Desired Result
add1	271	0.1	39	0.1	30	0.1	18
add2	11,807	0.1	347	0.1	229	0.1	114
add3	434,096	4.2	2,034	0.1	1,174	0.1	566
b1	559,115	2.0	103,78	0.1	2,062	0.1	901
z4ml	NA[a]	NA	47,221	0.1	23,993	0.1	4,360
cm138a	NA	NA	774,005	3.2	127,599	0.1	29,362
x2	NA	NA	978,074	1.2	168,072	0.8	38,364

[a] NA = not available; not computed because of complexity.

5.2 EQUIVALENCE CHECKING RESULTS

Our algorithm for semantic matching was implemented by using the University of Colorado's decision diagram library (Somenzi 1997). Table 6 compares our procedure with the factorial approach. For each component, it shows the size of the subcircuit, size for the BDD representation of the component's pattern function (under some reasonable variable ordering), number of input matchings, and total number of BDD equivalence checks made during the program's run time. The run time shown is the worst-case run time (a complete search of the

TABLE 6 Experimental Results[a]

Circuit Name	No. of Inputs	No. of Outputs	BDD Size	Input Matchings Method 1	Input Matchings Method 2	Correspondences Checked Method 1	Correspondences Checked Method 2	CPU Time (second)
C1908	33	25	127,349	8.7e+36	7.9e+12	1.3e+62	NA[b]	NA
alu2	10	6	231	3.6e+06	2.0e+00	2.9e+10	32	0.2
alu4	14	8	1,452	8.7e+10	8.6e+03	3.5e+15	6.9e+04	232.4
cc	21	20	57	5.1e+19	1.4e+07	1.2e+38	1.5e+09	37,675.5
f51m	8	8	73	4.0e+04	4.8e+01	1.6e+09	4.3e+02	0.1
pml	16	13	42	2.1e+13	2.0e+05	1.3e+23	2.8e+06	273.4
sct	19	15	102	1.2e+17	4.0e+07	1.6e+29	6.0e+08	75,647.4
t481	16	1	202	2.1e+13	2.3e+07	2.1e+13	2.3e+07	88,354.5
z4ml	7	4	47	5.0e+03	5.0e+03	1.2e+05	2.0e+04	4.55

[a] The circuits included in this table are a subset of the LGSynth93 benchmark suite. The results listed for Method 1 are calculated for the factorial permutation approach (Section 2.5). The results presented for Method 2 are experimental results for a single vector signature implementation of the algorithm presented in Section 4.

[b] NA = not applicable.

correspondence space). For nonsymmetric circuits, the time to determine a single correspondence can be considered roughly 50% of the overall run time. For circuits containing symmetries, the entire time is necessary to identify all legal correspondences, but only a fraction of the time is necessary to determine a single correspondence.

The z4ml circuit (a three-bit adder) shows a case in which the inputs are indistinguishable from their vector signature, and thus the number of input matchings is 7!. Note that because the algorithm automatically prunes (i.e., reduces) the output search space, the number of comparisons is only 20,304, an order of magnitude less then the number of comparisons necessary in a 120,160 (7!4!) nonpruned search.

The alu4 circuit (a four-bit ALU) is complex enough to have fairly well-distributed vector signatures and thus is able to take advantage of vector signature information to recognize that only 8,640 of the greater than 87 billion possible input matchings can possibly produce a legal correspondence. The use of vector signatures has made this intractable comparison feasible. Furthermore, note that of the 3.5 million billion total correspondences (14!8!) possible, only 69,411 comparisons are necessary.

Using the vector signature to prune the input permutation search space of the C1908 error-correcting circuit reduces the number of input matchings from 8.7×10^{36} to 7.9×10^{12}. While this practice certainly results in a significant reduction in search space, additional signatures need to be applied to permit semantic matching within a reasonable execution time.

Table 7 summarizes the results from an experiment to identify the functional components contained in a library netlist. Specifically, one- and two-bit adders were found in two- and three-bit adder circuits.

5.3 IDENTIFICATION OF FUNCTIONAL COMPONENTS

By using the MI-Enum algorithm to identify candidate clusters and the MI-ID algorithm to check equivalence, we have created a tool that can find arbitrary library entities in a combinational circuit. The development of this tool is still in progress, but initial results prove that the concept is sound.

As noted previously, the equivalence checking algorithm does not prune the number of input correspondences when vector signatures are used on symmetric functions such as the adder, although it does prune the number of output correspondences. It will be far more interesting to attempt to find a larger function (such as the 181 ALU) in a large netlist.

TABLE 7 Results of Module Identification

Circuit	No. of Gates	No. of Connections	Module	No. Found	CPU Time (second)
2-bit adder	15	20	1-bit adder	2	0.5
3-bit adder	22	30	1-bit adder	3	0.6
3-bit adder	22	30	2-bit adder	2	0.6

6 FUTURE WORK

This section briefly discusses some areas that could be logical next steps in solving the general RE problem by using the module identification method. It concentrates on those extensions that apply toward solving the initial combinational circuit problem introduced in Section 1.

6.1 SUBCIRCUIT ENUMERATION ISSUES

The preliminary effort made to create and implement the graph enumeration algorithm has raised many interesting issues and possible focus areas for future efforts. At this point, the algorithm operates primarily as a "brute force" method. This problem demands that all possible valid subgraphs be explored to ensure a complete modular matching. Many heuristics and pruning methods could be applied to the algorithm to allow it to operate in a reasonable amount of time for reasonably sized problems.

6.1.1 Aggregation

When a match is found between a library entity and a subcircuit, that subcircuit could be replaced with a single node that encapsulates the functionality of the module. This aggregation of the subcircuit nodes into a single node would reduce the order of the graph and therefore the growth of the pool of subcircuits. This approach would also gradually raise the level of abstraction. The matched modules would eventually exist in the circuit connected only by glue logic. Each of these modules would still have the same functionality as the original subcircuit but would be only a single node with multiple unique outputs.

6.1.2 User Interaction

If a user can visually locate areas of the circuit that look as if they may be repeated elements, the user can enter them into a meta-library. These elements can be located by structural matching techniques such as Subgemini (Ohlrich et al. 1993). Any time the repeated element is found, its nodes can be aggregated into a single node, thus reducing the order of the circuit. At this point, MI-Enum could be run on the new circuit as usual.

6.1.3 Parallel Implementation

This algorithm is inherently parallelizable. Each pool of subgraphs could easily be divided into any number of parts and parceled out to individual processors. The only effect would be that some subgraphs might be checked more than once, but no more times than the number of processors involved.

6.1.4 Preliminary Partitioning

A method of addressing the problem of the intractability of large circuits involves the partitioning of the circuit before processing. Instead of attacking a problem of 10,000 gates, the circuit could be split into 100 circuits of 100 gates each. The obvious difficulty with this method is that a library module that exists in the circuit could be split across the cut boundary and never matched. The sliding window method discussed in Section 3.3 provides a solution that works well for reasonably sized circuits, but an extremely large circuit would benefit from initial partitioning.

6.1.5 Primitive Modules

To reduce the order and thus the processing time of the graph, it might be useful to investigate the use of primitive modules. These modules could consist of small common functions that are generally implemented as 5 to 10 gates. By including them in the library, the order of the graph could be greatly reduced by aggregation.

6.1.6 Order Limiting

To reduce the number of subgraphs generated, it helps to limit their order. For example, if the largest module in the library had 20 gates, the candidate subcircuits could be restricted to a maximum of 30 gates. Doing so would probably allow all of the likely implementations to be identified yet significantly reduce the number of subcircuits that need to be generated.

6.2 SEMANTIC MATCHING ISSUES

6.2.1 Effectiveness of Additional Filters

The vector signature alone is not an effective filter for several of the circuits tested. Additional function filters, such as those described in Section 2.7.2, are necessary if this technique is to be used effectively. The effectiveness and the cost of each filter should be explored, and the identification of intractable problems, if any, should be facilitated.

In particular, the discrete Fourier function transformation (FFT) may allow the efficient determination of equivalence between output variables independent of input correspondences. If outputs can be efficiently determined, vector input signatures can consider the particular values of corresponding outputs rather than simple output 1-sums. This technique promises to significantly speed up the computation times reported herein if the FFT calculation can be determined efficiently.

6.2.2 Don't Care Optimizations

The primary problem that must be addressed is the issue of don't care optimizations. Such optimizations are prevalent, and any approach that does not take don't care conditions into consideration cannot be completely successful. It is our intent to formally prove and experimentally demonstrate that structural BDDs (Doom and Wojcik 1997, Doom et al. 1998) can efficiently identify the don't care set of any cluster. As a challenge, we might consider proving the equivalence between two binary coded decimal (BCD) adders for which the output functionality under non-BCD inputs is undefined. The equivalence between a BCD adder to which non-BCD inputs are never supplied and a non-BCD adder should also be provable.

6.2.3 Canonical Variable Ordering

A technique for canonically ordering variables based on the recursive sorting of truth tables by row and column sums is presented in (Wu et al. 1994). If this technique can be implemented efficiently, it will be completely unnecessary to consider searching the factorial matching space to determine P-equivalence. The canonical ordering for the cluster and the canonical order for the entity must indicate an appropriate matching if any such matching exists. A tool based on this mechanism should be developed and tested for efficiency as well as maximum problem size. As a truth-table-based technique, this canonicalization requires $O(2^{|\vec{i}|})$ memory and time, which may limit its utility, but this technique is quite promising.

This technique could be quite useful in performing (exact) equivalence matching, because we would no longer need to test equivalence under all input correspondences. We would merely need to determine the "unique" input order of the function before the test. This technique would be more efficient than current techniques for many functions, particularly those with a large number of inputs and a small number of outputs for which signature-based techniques may prove intractable. If there are no don't cares, this technique can be used to hash to the matching library function, if any. If there are don't cares, the canonicalization will have to be performed on each library unit after the mask is applied.

To the best of our knowledge, no technique for canonicalizing the variables in a BDD has ever been proposed. Perhaps a metric (similar to the row and column sums) by which a BDD could be recursively ordered could be determined. If so, this metric would be a significant contribution to the BDD field as well as RE.

6.2.4 Intractable Functions

Some functions are inherently difficult to describe and match when this technique is used. The multiplier and multiplexer are two such functions. The multiplier is quite sensitive to filtering, and the number of comparisons necessary is relatively small. However, each comparison takes a lot of time. Multipliers are well known to produce exponential graphs when represented as a BDD. Creating the BDD that represents the function of the multiplier under some variable ordering may be prohibitively time consuming.

The multiplexer function, on the other hand, is almost completely insensitive to the vector filter function. A multiplexer consists of n control inputs, 2^n data inputs, and a single output whose value is equal to that of the input selected by the control inputs. Although the n control inputs may be identified by the vector signature, all but two of the data inputs (the $\vec{1}$ and $\vec{0}$ lines) fall into the same equivalence class (since their behavior is never selected by the control inputs). If n is greater than four, there would be at least 14 (i.e., $2^4 - 2 = 16 - 2$) input variables in the same vector class, requiring at least 14! comparisons, which is intractable. The equivalence algorithm can flag such clusters as being intractable comparisons, but some other method has to be used later to consider these cases.

6.2.5 Sequential Circuits

The identification of latches in a sequential circuit seems to be a simple problem, whereas the identification of larger sequential units (such as a shift register) seems to be very challenging. Once the identification of high-level functional units in both combinational and sequential netlists is accomplished, this information can be used by new tools to move understanding to the next level.

We believe that once the problems regarding the identification of combinational circuits are solved, the identification of sequential circuits will be an obvious extension. Sequential elements tend to be clustered more regularly. After the high-level combinational entities between the sequential entities are partitioned out, the identification of the sequential entities should be less complex.

6.3 JOINT ISSUES

6.3.1 Optimized Circuits

Matching entities whose corresponding clusters have fewer inputs or outputs because of bridged inputs, stuck-at inputs, or neglected outputs is a difficult problem. No approach toward solving this problem seems promising at this time. A solution would most likely involve identifying the partial functions and directing the growth of the cluster. This issue remains unexplored.

If the canonical form mentioned in Section 6.2.3 could be found, it might be useful in attacking the enumeration problem. By creating canonical keys for any function, we could create a library of "interesting" keys related to high-order digital devices.

For any one-output cone of logic being tested, if its function was interesting (i.e., if the function's canonical form matched one of the forms determined to be associated with one or more of the high-order devices), it would seem wise to expand the search around that cluster to attempt to find other interesting functions associated with the same device. If all of the functions for some device were discovered, we could replace the cluster with the device.

This technique would not necessarily be better than any of the techniques that we are exploring now, but it does have some possible advantages. Most importantly, it would allow us to find a partial match for a high-order library device. For example, perhaps we could find three functions that matched the outputs of three of the five outputs of some device. In this case, it is quite likely that a detailed search of the area near the cluster would reveal the other two inputs (possibly passed over because of don't care optimizations). This ability to recognize partial matches might make the basis for a good genetic algorithm (GA) evaluation function for a GA approach to partitioning, etc.

6.3.2 Structural Matching

Structural matching techniques are not a suitable solution to the general MI problem. Since there are an infinite number of ways in which any high-level entity might be implemented, identification of such entities by purely structural means is not an adequate solution. Because most circuits are developed by using cell libraries, however, structural techniques might be useful in finding additional instances of a high-level entity (which has been discovered by using functional techniques) elsewhere in the circuit. This approach would be most useful for a logical function with an empty don't care set.

7 CONCLUSION

As discussed in this report, we have met our goal of determining a general method for identifying functional components in a combinational circuit through the use of semantic techniques. This report presents an initial enumeration algorithm that is a working model on which future enumeration algorithms may be based. In addition, it presents a technique that allows a semantic match between a circuit cluster subcircuit and a high-level component to be determined in a tractable number of comparisons. It presents the underlying equivalence problem and provides an algorithm based on the concept of suspect sets capable of solving problems of a reasonable size. Preliminary experiments demonstrate the effectiveness of the technique when a single vector signature filter is used. Future goals include the introduction of additional filters to decrease the run time and increase the capabilities of the program.

In the long term, we will use this technique as an RE tool. Semantic matching techniques allow us to achieve a functional specification of many digital designs by identifying clusters of logic that correspond to higher-level functional components. By identifying high-level components such as ALUs, adders, multiplexers, and other common functional entities within the circuit, we reduce the complexity required to produce functional descriptions and to identify data lines, control lines, and other “additional knowledge” (Ohmura et al. 1990) that might be useful in further specifying the design. Such an approach requires the implementation of efficient enumeration techniques as well as the identification and incorporation of don’t care conditions (Doom in progress) into the semantic matching algorithm.

8 REFERENCES

Alpert, C., and A. Kahng, 1995, "Recent Developments in Netlist Partitioning: A Survey," *Integration: The VLSI Journal* 19(1–2):1–81.

Benini, L., and G. De Micheli, 1997, "A Survey of Boolean Matching Techniques for Library Binding," in *Transactions on Design Automation of Electronic Systems (TODAES)* 2(3):193–226, Association for Computing Machinery (ACM), July.

Bochner, M., 1988, "LOGEX — An Automatic Logic Extractor from Transistor to Gate Level for CMOS Technology," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 517–522.

Bryant, R.E., 1985, "Symbolic Manipulation of Boolean Functions Using a Graphical Representation," in *Proceedings of the ACM/IEEE Design Automation Conference*, Las Vegas, Nev., pp. 688–694.

Christofides, N., 1975, *Graph Theory: An Algorithmic Approach*, Academic Press, Inc.

Doom, T.E., in progress, *Design Recovery for Combinational Logic Exploiting Boolean Relationships*, Ph.D. thesis, Michigan State University, East Lansing, Mich.

Doom, T.E., and A.S. Wojcik, 1997, *Reengineering from Partial Specifications though BDD Representation of Functional Constraint*, Technical Report MSUCPS:TR97-3, Department of Computer Science, Michigan State University, East Lansing, Mich. [URL: http://web.cps.msu.edu/TR/MSUCPS:TR97-3].

Doom, T.E., A.S. Wojcik, and M. Chung, 1998, "Design Recovery for Incomplete Combinational Logic Exploiting Boolean Relationships," submitted for publication in *Proceedings of the 1998 ACM/IEEE Design Automation Conference*.

Eckmann, S., and G.H. Chisholm, 1997, *Assigning Functional Meaning to Digital Circuits*, ANL/DIS/TM-43, Argonne National Laboratory, Argonne, Ill.

Lai, Y., S. Sastry, and M. Pedram, 1992, "Boolean Matching Using Binary Decision Diagrams with Applications to Logic Synthesis and Verification," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 452–458.

Luellau, F., T. Iloepken, and E. Barke, 1984, "A Technology Independent Block Extraction Algorithm," in *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 610–615.

Mailhot, F., and G.D. Micheli, 1993, "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations," *IEEE Transactions on CAD/ICAS* 12(5):599–620.

McElvain, K., 1993, *LGSynth93 Benchmark Set: Version 4.0* [URL: ftp://ftp.mcnc.org/pub/benchmark/Benchmark_dirs/LGSynth93].

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

Ohmura, M., H. Yasuura, and K. Tamaru, 1990, "Extraction of Functional Information from Combinational Circuits," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 176–179.

Somenzi, F., 1997, *CUDD: CU Decision Diagram Package, Release 2.1.2* [URL: http://bessie.colorado.edu/~fabio/CUDD/].

Wu, Q., C. Chen, and J. Acken, 1994, "Efficient Boolean Matching Algorithm for Cell Libraries," in *Proceedings of the IEEE International Conference on Computer-Aided Design*, pp. 36–39.

M98004621

Report Number (14)	ANL/DIS/TM--47
Publ. Date (11)	199801
Sponsor Code (18)	DOD, XF
UC Category (19)	UC-000, DOE/ER

DOE