Using if-then-else DAGs for Multi-Level Logic Minimization

Kevin Karplus*

UCSC-CRL-88-29
30 November 1988

Baskin Center for
Computer Engineering & Information Sciences
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

ABSTRACT

This article describes the use of if-then-else DAGs for multi-level logic minimization.

A new canonical form for if-then-else DAGs, analogous to Bryant's canonical form for binary decision diagrams (BDDs), is introduced. Two-cuts are defined for binary decision diagrams, and a relationship is exhibited between general if-then-else expressions and the two-cuts of a BDD for the same function. The canonical form is based on representing the lowest non-trivial two-cut in the corresponding BDD, instead of the highest two-cut, as in Bryant's canonical form.

The definitions of *prime* and *irredundant* expressions are extended to if-then-else DAGs. Expressions in Bryant's canonical form or in the new canonical form can be shown to be prime and irredundant.

Objective functions for minimization are discussed, and estimators for predicting the area and delay of the circuit produced after technology mapping are proposed.

A brief discussion of methods for applying don't-care information and for factoring expressions is included.

*This research was partially supported by NSF grant DCR-8503262 and and IBM Faculty Development award.

1 Introduction

1.1 What is multi-level logic minimization?

Multi-level logic minimization is the transforming of a specification of a Boolean function into an equivalent representation that can be implemented as a circuit with better characteristics than a circuit built from the original specification. The function usually has multiple outputs, and may be only partially specified. The parameters to be optimized are usually the area and delay of the final circuit, which must be estimated from the representation of the function.

The minimization may be done by global or local optimization techniques. The main global techniques are factoring by weak division, algebraic substitution to reuse already computed functions, and various algorithms to find common factors. Local optimization techniques are primarily "peephole" optimizations that apply ad hoc transformations to small parts of a circuit. The two classes of techniques overlap, as some transformations can be applied to make global changes, and factoring techniques can be applied locally.

Most previous work in multi-level logic synthesis is based on extensions of two-level (sum-of-products) minimization for PLAs [Bra87, BHL+87, BCDH86, HL87, BBH+88]. A notable example is the misII multi-level minimization system [BRSW87], based on the espresso two-level minimizer [BHMS84].

Some subproblems of multi-level minimization may be easier in representations other than sum-of-products. For example, tautology checking, finding common subexpressions, and extracting exclusive-or operations look more attractive in the if-then-else DAG form (see Section 1.3) than in sum-of-products form. We are investigating using if-then-else DAGs to do multi-level logic minimization, and have some encouraging preliminary results.

Section 1.2 describes the if-then-else operator, which forms the basis for the representation used, and Section 1.3 gives a quick introduction to binary decision diagrams and if-then-else DAGs. Section 2 will review binary decision diagrams and Bryant's canonical form, then introduce *two-cuts*, which give a natural mapping from binary decision diagrams to if-then-else DAGs. Section 3 will introduce *two-cut canonical forms* (of which Bryant's canonical form is a special case). Section 4.1 discusses ways to estimate rapidly the area and delay of a circuit in a technology-independent logic minimizer, Sections 4.2 and 4.3 discuss the conversions needed between networks of gates in sum-of-products form and if-then-else DAGs, Section 4.4 talks about ways to use don't-care information for simplifying if-then-else DAGs, and Section 4.5 describes some crude factoring techniques that do surprisingly well.

1.2 The if-then-else operator

The choice of operators used in the representation is critical. For simulation and other expression evaluation applications, arbitrary operators can be used, as long as an executable definition is provided. If simplification or comparison of expressions is needed, the class of operators must be restricted to a set of operators whose semantics are understood by the program. The most extreme form of restriction is to use a single, universal operator, and to define all other operators using it. For example, the misII technology mapper first converts the input description into a circuit using only 2-input NANDs, then does graph matching between that circuit and a technology library [DGR+87].

The research described here is based on a different universal operator—the if-then-else operator.

Definition 1: *The* if-then-else *operator is a ternary Boolean function, with* (**if** a **then** b **else** c) *defined as* $ab + \neg ac$ *or, equivalently,* $(a + \neg c)(\neg a + b)$.

All binary Boolean functions are easily defined with the if-then-else operator. For example,

- $ab =$ (**if** a **then** b **else** FALSE)
- $a + b =$ (**if** a **then** TRUE **else** b)
- $a \oplus b =$ (**if** a **then** $\neg b$ **else** b).

1.3 Binary decision diagrams, if-then-else trees, and if-then-else DAGs

If-then-else trees and DAGs have a long history—references to if-then-else trees include [Lee59], [Ake78], and [Bry86]. We divide if-then-else representations into two classes: *binary decision diagrams*, in which the **if**-part is always a simple variable, and *if-then-else* DAG*s*, which may have arbitrary expressions in the **if**-part.

Definition 2: *A* binary decision diagram *is a binary directed acyclic graph with two leaves* TRUE *and* FALSE*, in which each non-leaf node is labeled with an atom and has two out-edges pointing to the* **then**-*part and the* **else**-*part. The meaning of a binary decision diagram is defined recursively as* (**if** *label(node)* **then** *meaning(***then**-*part)* **else** *meaning(***else**-*part)*).

Binary decision diagrams (BDDs) have been used extensively for logic verification work, for example in [Bry85, RI86, SF86, MWBS88]. They are attractive for such work as they are easy to manipulate and have a convenient canonical form (Bryant's canonical form). They have also been used for logic synthesis work, mainly for designing differential voltage-cascode switches, which implement BDDs directly [NB86, FS87]. For logic minimization in other technologies, the mismatch between the BDD structure and the circuit structure has restricted their use.

If-then-else DAGs generalize binary decision diagrams by not restricting the **if**-parts to single variables:

Definition 3: *An* if-then-else DAG *is a ternary directed acyclic graph in which each leaf is labeled with* TRUE*,* FALSE *or a literal, and each internal node has three out-edges pointing to the* **if**-*,* **then**-*, and* **else**-*parts. The meaning of a leaf node is the label on the node, and the meaning of an internal node is defined recursively as*

(**if** *meaning(***if**-*part)* **then** *meaning(***then**-*part)* **else** *meaning(***else**-*part)*).

If-then-else DAGs offer several advantages over sum-of-products and Boolean decision diagram representations.

- If-then-else DAGs can be used to represent BDDs and sum-of-products expressions, but neither BDDs nor sum-of-products forms can represent if-then-else DAGs.
- Circuits built out of arbitrary gates can be converted to if-then-else DAGs without losing any sharing of common subexpressions.
- If-then-else DAGs, like BDDs, have a convenient canonical form. Canonical forms are particularly valuable for tautology checking.
- The new canonical form for if-then-else DAGs allows more sharing of subexpressions than Bryant's canonical form for BDDs. Any shared subexpressions in Bryant's form have corresponding sharing in the new canonical form, but the new form allows more sharing in the **if**-part.

2 Binary decision diagrams

Binary decision digrams (BDDs, for short) use a single universal operator, are easily converted to canonical forms, and can share subexpressions either within an expression or across all expressions. BDDs are easy to evaluate, but unless extra restrictions are put on them, they can be difficult to simplify or to compare for equality. By applying appropriate restrictions, we can create a canonical form for BDDs, making equality checks trivial.

A representation is *canonical* if any two expressions that are logically equivalent are identical. For example, if $ab + a\neg b$ is represented differently from a, then the representation is non-canonical. We can distinguish between *weak canonical forms*, in which logically equivalent expressions have identical structure but may occur in different locations in memory, and *strong canonical forms* in which expressions in different locations represent different Boolean functions. Strong canonical forms are particularly useful, because they guarantee that any explicitly represented subexpression is shared by all expressions that need it.

Using canonical forms makes checking for equivalence easier. For a strong canonical form, only one pointer has to be checked for equality, and for other canonical forms, a simple traversal of the data structure (taking $O(n)$ time) suffices. Unfortunately, conversion from a non-canonical form to canonical form may take a lot of time or memory. Because equivalence checking in canonical form is fast, but equivalence checking in a non-canonical form (such as clause form) is equivalent to the the NP-complete problem SATISFIABILITY, we are essentially guaranteed that the conversion to any canonical form is exponential in the worst-case. For most commonly used Boolean functions, however, a well-chosen canonical form can be small and easy to manipulate, and exponential blow-up is rare.

A recent paper by Randy Bryant shows that one common function, integer multiplication, requires exponentially many nodes to represent in his canonical form, no matter what ordering is used for the variables [Bry88]. The same arguments can be applied to the new canonical form described in Section 3.1. Small if-then-else DAGs for integer multiplication are easy to construct from small circuits, but they all involve duplicating variables, and so are not canonical.

2.1 Bryant's canonical form

For binary decision diagrams, Bryant's canonical form is commonly used [Bry86]. As originally described, it is a weak canonical form, but adding a permanent symbol table to give unique ids to if-then-else nodes makes it a strong canonical form.

Bryant's canonical form is obtained by putting two restrictions on BDDs. The first restriction is to require that the set of all atoms be ordered, and that the atom at each node of the diagram be earlier in the order than the atoms of the children. Among other things, this restriction guarantees that no atom appears twice on any path from the root to a leaf. The other restriction is that distinct nodes represent non-equivalent expressions.

After implementing several different variants of Bryant's representation, I observed that his method for performing boolean operations on BDDs can be considerably simplified. Bryant implemented all the standard binary operators with a general `Apply()` operator, which took as arguments the operands and a description of what the operator did on the leaves of the BDD. All these Boolean operations can be defined in terms of a single if-then-else operator. The if-then-else operator can be implemented by the same sort of traversal as is used for `Apply()`, but without having to keep track of the operator to be applied at the leaves.

The if-then-else operator can be defined to generate canonical from representations automatically, by recursively using two simpler operations: `UniqTriple` and `split`. `UniqTriple` is a symbol table routine that looks up the if-then-else triple passed as arguments and creates a new entry in the symbol table if the triple is not found. We define `split(a,v,left)` to be the left subDAG `a->left` if `v` is the atom in the root, and the whole DAG `a` otherwise. To change (**if** a **then** b **else** c) to canonical form (with each of a, b, and c already in canonical form), we choose the smallest atom in the three arguments (call it `m`) and return

```
UniqTriple(m, IfThenElse(split(a,m,left),split(b,m,left),split(c,m,left)),
              IfThenElse(split(a,m,right),split(b,m,right),split(c,m,right)) )
```

We stop the recursion when we get to an obvious special case—for more details, see [Karon].

2.2 Dominators and two-cuts in binary decision diagrams

One of the first problems we encountered in using BDDs was printing them out in a readable fashion. We found that heavily parenthesized if-then-else trees were hard to understand, and that printing all paths to TRUE generated voluminous output, with many more terms than needed, and extra literals in some of the terms. For example, printing all paths to TRUE in $ab+cd+ef$ (represented by the binary decision diagram in Figure 2.1) would produce $ab + a\neg bcd + a\neg bc\neg def + a\neg b\neg cef + \neg acd + \neg ac\neg def + \neg a\neg cef$.

Figure 2.1: Binary Decision Diagram for the expression $ab+cd+ef$, showing the two-cuts.

Attempting to print expressions more compactly is essentially a multi-level logic minimization problem, with the rather unusual objective of minimizing the number of literals after all shared expressions have been duplicated. Dipen Moitra and I looked for properties of the graphs that could be used to do this minimization. We identified two such properties—*dominators* and *two-cuts*.

Definition 4: *Vertex v of a rooted* DAG *is a* dominator *of vertex w, if, and only if, every path from the root to w contains v.*

One particularly interesting set of nodes in a BDD is the dominators of the leaf node FALSE. In [KM90], we proved that a BDD with a non-trivial dominator of FALSE (that is, a dominator other than the root or FALSE) can be split into two BDDs that are OR'd together. This *OR-split* is particularly useful for converting BDDs into sum-of-products form. For example, in Figure 2.1, the nodes labeled with c and e are dominators of FALSE, and the expression can be printed as $ab+cd+ef$. In a similar way, the dominators of TRUE can be used to do an *AND-split*.

The dominators of TRUE and FALSE in a BDD can easily be computed as the BDD is built, usually taking only $O(n)$ time to build the dominator structure for a BDD with n nodes, but in the worst case taking $O(n^2)$ operations [Karon].

Because the dominators of TRUE and FALSE were so useful for printing expressions, we tried to generalize the concept, looking for a more powerful way to reduce the complexity of the expression. The useful property of dominators was that we could cut a BDD into two parts by removing one interior node. A natural generalization was to look at ways to cut a BDD apart by removing two nodes.

Definition 5: *A pair of vertices $\{x, y\}$ is a* two-cut *between the root r and a pair of vertices $\{v, w\}$, if, and only if, every path from r to v or w contains at least one of x or y. If a two-cut is mentioned without giving $\{v, w\}$ explicitly, then the pair $\{\text{TRUE}, \text{FALSE}\}$ is assumed.*

Every BDD has two trivial two-cuts: the leaves TRUE and FALSE themselves, and the two children of the root. Every dominator of TRUE or FALSE is part of a two-cut (with the other leaf node), but other two-cuts may exist. The paper [KM90] describes several useful properties of two-cuts and gives proofs.

Figure 2.2: Canonical binary decision diagram for $abc + \neg ad + \neg bd$, showing the two-cuts.

Figure 3.1: If-then-else DAG for $abc + \neg ad + \neg bd$, factored as (**if** ab **then** c **else** d).

The main use of dominators was in factoring an if-then-else DAG, by doing an OR-split at dominators of FALSE. If x is a dominator of FALSE, then $\{\text{TRUE}, x\}$ and $\{\text{TRUE}, \text{FALSE}\}$ are both two-cuts, and share a common vertex (TRUE). It turns out that any two-cuts that share a common vertex can be used to simplify the expression. Let's call such pairs of two-cuts *collapsed two-cuts*.

Consider the expression $abc + \neg ad + \neg bd$, whose binary decision diagram is shown in Figure 2.2. The two-cuts of the DAG are $\{2, 4\}$, $\{3, 4\}$, and $\{\text{TRUE}, \text{FALSE}\}$. Notice that the DAG has no dominators of either TRUE or FALSE, so printing using only dominator information yields $abc + a\neg bd + \neg ad$, which has an unnecessary a in the second term. Because two of the two-cuts share a common node ($\{2, 4\}$ and $\{3, 4\}$ share the node 4), we can do better on this expression. The whole expression can be viewed as (**if** ab **then** c **else** d), which is $abc + (\neg a + \neg b)d$.

3 If-then-else DAGs with an expression in the if-part

Finding two-cuts useful for simplifying binary decision diagrams, I looked for a new representation in which the two-cuts were more naturally represented. We can view a binary decision diagram with a two-cut as having three parts: the DAG from the root to the cut, and the two subDAGs below the cut. For example, for the BDD in Figure 2.2, the parts are ab, c, and d. If we allow arbitrary expressions in the **if**-part of if-then-else expressions, we can represent the two-cut explicitly as (**if** ab **then** c **else** d), as shown in Figure 3.1.

If-then-else DAGs are not new, but they do have several properties that make them more attractive than binary decision diagrams for CAD work. If-then-else DAGs

- provide a single representation scheme for representing binary decision diagrams, sum-of-products, and arbitrary combinations of 1- and 2-input gates. Every 1- or 2-input gate can be represented as an if-then-else triple, so every acyclic network of gates can be represented by replacing each gate by the appropriate if-then-else triple.
- expose more subexpressions for potential sharing than do binary decision diagrams. The same sharing of **then**- and **else**-parts is possible in both BDDs and if-then-else DAGs, but only the if-then-else DAGs allow sharing subexpressions in the **if**-part. For example, the three functions $ab(d+e)$, $c(d+e)$, and abd are represented as (**if** (**if** a **then** b **else** FALSE) **then** (**if** d **then** TRUE **else** e) **else** FALSE), (**if** c **then** (**if** d **then** TRUE **else** e) **else** FALSE), and (**if** (**if** a **then** b **else** FALSE) **then** d **else** FALSE), sharing the subexpressions (**if** a **then** b **else** FALSE) and (**if** d **then** TRUE **else** e).
- have at least two useful canonical forms: Bryant's canonical form and a new canonical form introduced in this paper.
- are a more factored form than BDDs, providing for better printing and logic minimization. With the aid of the transformations described in Section 4.5, good factorings can often be found from the canonical forms or from arbitrary non-canonical expressions.
- express Boolean operations naturally as if-then-else triples, so that the symbol table used for storing canonical forms can be used for caching the results of operations.

A free-form DAG using any subset of the operators AND, OR, XOR, NOT, and IF can easily be converted to an if-then-else DAG having the same structure. This conversion can also be done in the reverse direction. The mapping is not an isomorphism, as some information about the grouping of operands is lost when converting to the if-then-else DAG.

The experimental multi-level logic minimization programs I have been working on accept DAGs of arbitrary operators (in BLIF format [Ber88a]), convert them to if-then-else DAGs, transform them to reduce their complexity (as measured by the functions in Section 4.1), then convert to n-input AND, n-input OR, 2-input XOR, and NOT gates.

3.1 Two-cut canonical forms

One of the attractive features of binary decision diagrams, especially for verification applications, is the ease of computing a canonical form—Bryant's canonical form. Of course, binary decision diagrams are a special case of if-then-else DAGs, so we could use Bryant's canonical form for if-then-else DAGs as well, but there is another canonical form that lets us represent all the two-cuts explicitly, not just the two trivial ones (the two children of the root and the two leaves TRUE and FALSE).

An if-then-else triple corresponds to a BDD with a two-cut. The **if**-part corresponds to the BDD above the cut, and the **then**- and **else**-parts correspond to the subDAGs below the two-cut. Restricting the **if**-part to simple variables is equivalent to choosing always to represent the topmost two-cut in the BDD. If, instead of choosing the topmost two-cut, we always choose the non-trivial one closest to the leaves, then the triple for the **if**-part of an expression corresponds to the next two-cut up. Following the chain of **if**-parts until we get to a literal gives us the two-cuts in order from the bottom up.

To simplify negation, and to reduce the storage needed for representing expressions, we allow negation of an if-then-else DAG to be represented by flipping one flag bit, which we keep in the low-order bit of the pointer to the DAG.

To make these if-then-else DAGs canonical, we must place some restrictions on the expressions allowed in the **if**-, **then**-, and **else**-parts of the structure. Of the seven restrictions, the first three are slightly modified versions of the corresponding restrictions in Bryant's canonical form.

1. All the atoms in the **if**-part must be earlier in the order than all atoms in the **then**- and **else**-parts. This restriction is directly translated from Bryant's restriction that the atom in a node is earlier in the order than the atoms of the subDAGs. A weaker restriction, that the variables of the **if**-part be disjoint from those of the **then**- and **else**-parts, would be enough to eliminate paths with duplicate variables, but not enough to make the form canonical. Non-canonical expressions using this weaker version of the restriction are useful for factoring.
2. The **then**- and **else**-parts of an expression must be distinct Boolean functions—exactly as in Bryant's canonical form.
3. A systematic choice must be made between the equivalent expressions $(\textbf{if } a \textbf{ then } b \textbf{ else } c)$ and $(\textbf{if } \neg a \textbf{ then } c \textbf{ else } b)$ and between $(\textbf{if } a \textbf{ then } b \textbf{ else } c)$ and $\neg(\textbf{if } a \textbf{ then } \neg b \textbf{ else } \neg c)$. This corresponds to Bryant's choice of atoms as node labels (never negations of atoms). We require that **if**- and **then**-parts of an expression be pure pointers, with negation allowed only for the **else**-part or the entire expression.
4. Triples of the form $(\textbf{if } a \textbf{ then } \text{TRUE} \textbf{ else } \text{FALSE})$ and $(\textbf{if } a \textbf{ then } \text{FALSE} \textbf{ else } \text{TRUE})$ are prohibited. The first triple should be represented simply as a, and the second one by $\neg a$.
5. Triples of the form $(\textbf{if } \text{TRUE} \textbf{ then } b \textbf{ else } c)$ and $(\textbf{if } \text{FALSE} \textbf{ then } b \textbf{ else } c)$ are prohibited, and should be replaced with b and c respectively.
6. In the triple $(\textbf{if } a \textbf{ then } b \textbf{ else } c)$, b and c must not share both **then**- and **else**-parts. If $b = (\textbf{if } b_a \textbf{ then } b_b \textbf{ else } c_c)$ and $c = (\textbf{if } c_a \textbf{ then } b_b \textbf{ else } c_c)$, then the correct representation for the original expression is $(\textbf{if } (\textbf{if } a \textbf{ then } b_a \textbf{ else } c_a) \textbf{ then } b_b \textbf{ else } c_c)$. If $b = (\textbf{if } b_a \textbf{ then } b_b \textbf{ else } b_c)$ and $c = (\textbf{if } c_a \textbf{ then } b_c \textbf{ else } b_b)$, then convert the original expression to $(\textbf{if } (\textbf{if } a \textbf{ then } b_a \textbf{ else } \neg c_a) \textbf{ then } b_b \textbf{ else } b_c)$.
7. In the triple $(\textbf{if } a \textbf{ then } b \textbf{ else } c)$, b must not contain c as a **then**- or **else**-part. If $b = (\textbf{if } b_1 \textbf{ then } b_b \textbf{ else } c)$ or $b = (\textbf{if } b_2 \textbf{ then } c \textbf{ else } b_c)$, then the expression should be represented as $(\textbf{if } (\textbf{if } a \textbf{ then } b_1 \textbf{ else } \text{FALSE}) \textbf{ then } b_b \textbf{ else } c)$ or $(\textbf{if } (\textbf{if } a \textbf{ then } b_2 \textbf{ else } \text{TRUE}) \textbf{ then } c \textbf{ else } b_c)$. If c is one of the constants TRUE or FALSE, this condition amounts to choosing left-associativity for commutative AND or OR operations. The symmetric test for $c = (\textbf{if } c_1 \textbf{ then } c_b \textbf{ else } b)$ or $c = (\textbf{if } c_2 \textbf{ then } b \textbf{ else } c_c)$. is also needed.

We can show that imposing the conditions listed above defines a canonical form by exhibiting an isomorphism with Bryant's canonical form [Karon]. We can use essentially the same algorithm for converting to either Bryant's canonical form or the new form [Karon].

3.2 Two-cut canonical forms are prime and irredundant

Other researchers in multi-level minimization, working primarily with sum-of-products representations, have found the concepts of *primality* and *irredundancy* to be important, particularly for producing testable circuits [BHMS84, page 28], [Bra87, page 202]. Both concepts have natural analogs in if-then-else DAG representations.

In sum-of-products form, an expression is said to be *prime* if no term could be modified by changing a literal to TRUE without changing the meaning of the expression. Similarly, an expression in sum-of-products form is said to be irredundant if no term can be changed to FALSE without changing the meaning of the expression.

Definition 6: *An if-then-else* DAG *is* prime *if no pointer to a literal, sub*DAG*, or the constant* FALSE *could be replaced with a pointer to* TRUE *without changing the meaning of the expression. An if-then-else* DAG *is* irredundant *if no pointer to a literal, sub*DAG*, or the constant* TRUE *could be replaced with a pointer to* FALSE *without changing the meaning of the expression.*

The new definitions of "prime" and "irredundant" correspond to existing ones for sum-of-products and factored forms. For example, we can use an if-then-else tree (so that no sharing is done between terms) to represent a sum-of-products expression by replacing each binary AND or OR operator by the corresponding if-then-else triple. If the if-then-else tree is prime or irredundant, then the sum-of-products expression must be, because the substitutions to be tested in the sum-of-products form are a subset of those tested in the if-then-else tree. The extra tests for primality and irredundancy in the if-then-else tree are easily satisfied for trees corresponding to sum-of-products representations [Karon].

Factored forms, literals combined with arbitrary combinations of AND and OR operators, can also be represented as if-then-else trees. Brayton's definitions of prime and irredundant for factored forms [Bra87, page 203]) correspond to the definitions for if-then-else trees.

Both Bryant's canonical form and the new canonical form presented in Section 3.1 can be shown to be prime and irredundant (for empty don't-care sets) with the definition presented here [Karon].

4 Multi-level logic optimization

Logic synthesis usually consists of several somewhat separable stages. My current work has been applying if-then-else DAGs to one of those stages—technology-independent multi-level logic optimization. In this stage of the process, the main goal is to reduce the complexity of a logic network, so that technology mappers can find good implementations.

4.1 Expression complexity, counting literals

When doing logic minimization, the first question is "what exactly is being minimized?" Usually there are constraints on signal delays and on implementation costs, and the goal is to find the best (fastest or cheapest) design that meets the constraints. Unfortunately, both signal delay and implementation cost are very dependent on the technology used, and optimizations tied to a particular cell library quickly become obsolete.

When doing technology-independent logic minimization, determining whether a particular proposed change in an expression is an improvement can be difficult. The goal of minimization is to reduce the area, power, or delay of the final circuit after technology mapping, but without having to do the computationally expensive mapping repeatedly.

Just as compiler writers have found code optimizations that work well independent of the target machine, we look for logic optimizations that will work well independent of the target technology. After doing what optimization we can in a technology-independent way, a *technology mapper* generates a specific implementation. Some optimizations are done by the mapper, equivalent to the peephole optimizations done by the code generator of a compiler.

For technology-independent minimization to work, we need measures that are not dependent on any particular cell library (or that are parameterized and easily tuned for different technologies), and that roughly approximate the cost or speed obtained by a technology mapper. Technology-independent delay estimates are hard to come up with, and so most research has concentrated on size minimization, leaving the delay minimization to the technology mapper. Other researchers have used

- the number of literals in sum-of-products form,
- the number of literals in the factored form,
- the number of distinct literals (a and $\neg a$ count separately) the function depends on, or
- the number of distinct variables that the function depends on

as an estimate of the complexity of a complex gate, and added the size estimates for all gates in a network to get a size estimate for the network [Bra87, page 235].

Most technology-independent minimization work has used the same measure for determining the quality of their results—the sum over all gates of the number of literals in the description of the gate. This measure is usually justified as an approximation of the area of the resulting circuitry—especially for cMOS, where it corresponds fairly closely with the number of transistor pairs needed. The literal count is an excellent area estimator if the mapper does not change the decomposition of the circuit, but does not work as well when the mapper splits or merges gates.

Many technology mappers do polarity assignment, adding or removing inverters to minimize delay or area. For such mappers, adding inverters in the input description usually does not increase the cost of the final solution, and so should have zero cost for the technology-independent minimizer. None of the standard cost functions described above have this property. A cost estimator that hides such inverters should be a better estimator of final area.

Figure 4.1: Scatter diagram of the ratio between actual area and predicted area versus the actual area for the conventional estimator: the sum of the number of literals in the factored forms of the gates. The technology mapper is misII's `map -m1` command followed by `phase -g` using the `msu.genlib` library.

We have made some attempts to calibrate area estimators for misII's technology mapper on a collection of different designs, including the MCNC benchmarks. The mapper we attempted to calibrate was the command `map -m1` followed by `phase -g`. We looked at several different measures, including the ones reported by misII (number of nodes, sum of literals in sum-of-products form, sum of literals in factored form). Of the standard measures, the sum of literals in factored form was the best predictor, with the ratio of actual area over predicted area having a standard deviation of 16.8% of the mean ratio. See Figure 4.1 for a scatter diagram of the error of this estimate.

We also tried a measure intended to make inverters free, by subtracting the number of nodes in the Boolean network from the sum of literals in factored form. This measure is a slightly better fit, having a standard deviation of only 12.1% of the mean ratio. See Figure 4.2 for a scatter diagram.

The standard measures described above are useful when a network has been decomposed into gates, but are not directly applicable to a Boolean network described as an if-then-else DAG with multiple roots. New measures are needed that are as useful for estimating circuit size.

Several possible size measures could be used to estimate the area or delay of if-then-else DAGs. Rather than choosing one measure in an *ad hoc* way, we propose to use a parameterizable measure that we can calibrate differently for each technology mapper and library. Once we find good estimators for the delay and area achievable by a technology mapper, we can use them to guide the minimization process.

We have experimented with severalnon-parameterized estimators, including the following:

triples the number of if-then-else triples in the DAG,

size the number of triples plus the number of distinct variables,

opcount the number of n-input AND, n-input OR, and 2-input XOR gates produced by our decomposition algorithm,

height the longest path from a root to a leaf,

pcount the number of literals that would be leaves of an if-then-else tree created by duplicating any shared nodes, which is the number of times literals would appear in the factored expression, if no intermediate variables are introduced,

Figure 4.2: Scatter diagram of the ratio between actual area and predicted area versus the actual area for an improved estimator that counts the literals in the factored forms and subtracts the number of nodes in the network. The technology mapper is misII's `map -m1` command followed by `phase -g` using the `msu.genlib` library.

count a recursively defined function that attempts to match the values of the sum-of-products estimator (literals(factored)-nodes). **count** is

- 0 for the constants TRUE and FALSE.
- 1 for literals.
- 1 for a subDAG that has been previously counted.
- $c(x) + c(y) + c(z)$ for (**if** x **then** y **else** z), if the triple represents a 2-input AND or OR, that is, if y or z is a constant.
- $c(x) + c(y) + c(z) + 1$ for other triples (**if** x **then** y **else** z).

Of these new functions, `count` is the best predictor of area for our benchmarks, with a standard deviation of 12.73%. See Figure 4.3 for a scatter diagram showing the error of the estimate.

Delay estimation may be harder than area estimation. Our best predictor so far is the height of the if-then-else DAG, with a standard deviation of 34.8% (see Figure 4.4). We may be able to estimate delay better by adding a penalty to nodes that are used repeatedly, and by using smaller costs for triples that have a constant **then**- or **else**-parts. An active area of our research is to find good estimators of both area and delay for popular mapper-library pairs.

4.2 Coverting from BLIF (sum-of-products) format

Most of the standard benchmarks are available in a standard format, the Berkeley Logic Intermediate Format (BLIF, for short) [Ber88a], and so we need to convert BLIF files into if-then-else DAGs. In BLIF, combinational logic is described as a directed, acyclic network of gates, and each gate is described in sum-of-products form.

Building an if-then-else DAG from a network of gates is easy if each gate is described as an if-then-else DAG—the only tricky part is converting the sum-of-products descriptions of the gates into if-then-else DAGs. We have several choices:

- Build a canonical DAG for the function expressed by the gate. For gates of the form $ab + cd + ef + gh + \cdots$, the wrong variable ordering can cause an exponential blowup in size.

Figure 4.3: Scatter diagram of the ratio between actual area and predicted area versus the actual area for the best estimator found. The technology mapper is misII's `map -m1` command followed by `phase -g` using the `msu.genlib` library.

Figure 4.4: Scatter diagram of the ratio between actual delay and predicted delay versus the actual delay for the best estimator found. The technology mapper is misII's `map -m1` command followed by `phase -g` using the `msu.genlib` library.

- Preserve the original and-or structure of the sum-of-products expression. This is guaranteed not to be too big, but offers few advantages over simply using sum-of-product representations.
- Build a partially factored expression for the gate.

  We use a recursive function to get an if-then-else DAG E for a set of terms T. The terms are sorted, grouping together those that don't use the first input variable (T_d), those that use $\neg v_1$ (T_0), and those that use v_1 (T_1). We then strip the first variable off the terms in each group, and apply the routine recursively to get expressions E_d, E_0, and E_1. We build the expression E as (**if** E_d **then** TRUE **else** (**if** v **then** E_1 **else** E_0)). This idea can be improved by sorting the variables with the most frequently used ones first.

  This algorithm is essentially the same as the popular method of factoring out one-literal cubes, and produces expressions that are often significantly smaller than either the canonical form or the straight sum-of-products form.

After building an expression for a gate, we can try factoring the gate with the `Printform` or `LocalFactor` transformations, or we can try reordering the variables in various ways to attempt to reduce the size. When the gates in the input BLIF are large and complex, extra effort spent in minimizing them is valuable. When the gates are simple AND, OR, NAND, or NOR gates, no simplification is possible in a single gate.

After building if-then-else DAGs for all the gates, we can compose them to get a multiply-rooted if-then-else DAG for the entire logic module. The preliminary results in this paper were obtained by applying the transformations to each gate as it was built, and again after each composition (except for `seq`, in which memory limitations forced us to do factoring only on the gates).

4.3 Converting to BLIF networks (decomposition)

The misII technology mapper expects BLIF files as input, and so we need to convert if-then-else DAGs back into BLIF format. This task is a decomposition of the if-then-else DAG into simple gates, attempting to preserve any shared subDAGs as explicit nodes in the BLIF network. The decomposition procedure starts at an output, choosing a gate for the output, then recursively choosing gates for the necessary inputs to the selected gate. Each gate selected is an inverter, a n-input AND, a n-input OR, a 2-input XOR, or a 2-to-1 selector.

A simple set of heuristics for doing the decomposition is

- If the internal representation of the if-then-else DAG is a negated expression, use an inverter.
- If the DAG has the form (**if** a **then** b **else** $\neg b$), use an XOR gate.
- If the DAG has the form (**if** a **then** TRUE **else** c), use an n-input OR gate, collecting as many subexpressions as possible to use as inputs.
- If the DAG has the form (**if** a **then** b **else** FALSE), use an an n-input AND gate, collecting as many subexpressions as possible to use as inputs.
- Otherwise, use a selector to represent (**if** a **then** b **else** c) directly.

One small refinement has been added, in that subexpressions that are known to be shared are not expanded when collecting subexpressions for an AND or OR gate. For example, if abc is known to be shared, then $abcde$ will be decomposed as $(abc)(d)(e)$, rather than as $(a)(b)(c)(d)(e)$. The mechanism used for recognizing shared subexpressions is currently quite crude, and could be greatly improved by using the rectangle-covering techniques of misII.

Other refinements are possible. For example, n-input XORs could be recognized and converted to a properly balanced tree of 2-input XORs that attempts to minimize delay. We could also expand (**if** a **then** b **else** c) to either $ab + \neg ac$ or $(a + c)(\neg a + b)$, depending on which produces the smaller description.

4.4 Using don't-care information

A substantial part of the multi-level logic minimization is to determine when the value of some expression is irrelevant, and to use this *don't-care* information to simplify the expression [BBH+88, HL87].

The most important don't-care information in previous work is the so-called *global don't-care information*, which associates each node of a network with the function of the network up to that node. This information is explicit in if-then-else DAGs, and can be automatically used whenever operations are performed on the node. The *fanout don't-care* information for a node is used to decide when the function of the node is irrelevant, allowing us to change the function implemented by the node.

Determining the fanout don't-care information for an if-then-else DAG is fairly easy. For example, if we are trying to simply $e =$ (**if** a **then** b **else** c), knowing that we don't care what the value is when d is true, then

- we can simplify b with the don't-care expression $d + \neg a$,
- we can simplify c with the don't-care expression $d + a$, and
- we can simplify a with the don't-care expression $d + (\neg b \oplus c)$. If we have already simplified b or c, then we have to use the simplified version to build the new don't-care expression.

The simplification presently implemented is a simple algorithm. If e implies d, then e can be simplified to a special variable DON'T-CARE. The triple (**if** a **then** DON'T-CARE **else** c) simplifies to c, and the triple (**if** a **then** b **else** DON'T-CARE) simplifies to b. The triple (**if** DON'T-CARE **then** b **else** c) can be simplified to either b or c, choosing whichever is cheaper. This simple algorithm guarantees that the resulting expression is *prime* and *irredundant*, according to the definitions in Section 3.3 of [Karon].

Note that a shared subexpression may be simplified differently in its different uses, resulting in reduced sharing and, possibly, an increase in the overall size of the network. We can be a little more careful constructing the don't-care set for shared subexpressions by ANDing together the don't-care expressions derived from each usage. This extension has not yet been implemented.

4.5 Factoring

Factoring is the transformation of an expression to make it smaller. In work by other researchers, this has meant the conversion from sum-of-products form to a free form containing only AND and OR operators, minimizing the number of literals in the process. For if-then-else DAGs, the goal is to minimize whatever measure we have decided best predicts the property (area or delay) that we are trying to minimize. For example, the `Printform` transformations described below attempt to reduce the *pcount* metric. A better, but slower, set of transformations (`LocalFactor`) attempts to minimize the *count* metric.

The process of reducing the complexity of an expression is usually called *factoring*, because the main techniques used by other researchers involve finding shared parts of terms in a sum-of-products representation, and factoring them out. For example, $abc + ad + cd$ might be factored as $a(bc+d)+cd$. Our most effective factoring techniques involve transformations that change the order of the variables, either locally for one part of the DAG, or for the entire DAG.

`Printform` transformations

A complete description of the transformations used is beyond the scope of this paper, but I will illustrate the concept with a particularly simple set of transformations. These transformations were originally intended to be applied to if-then-else DAGs in my new canonical form, to make them easier to read when printed, and so are called *Printform* transformations. They are presented in more detail in [Karon].

expression	canonical form size	canonical form count	canonical form pcount	Printform size	Printform count	Printform pcount
$abcd$	7	4	4	7	4	4
$a+b+c+d$	7	4	4	7	4	4
$a \oplus b \oplus c \oplus d$	10	10	4	10	10	4
$ad+be+cf$	29	24	25	11	6	6
$a(c+d+e+g)+b(c+d+e+f)+$ $c(e+f)+d(f+g)$	32	24	30	36	23	23
$(a+b+c+d)(\neg a+\neg b+\neg c+\neg d)$	12	8	8	12	8	8
$(b+c)(a(e+g)(\neg e+f+\neg g))+h+i)+$ $d(e+g)(\neg e+f+\neg g)$	27	20	37	21	14	19
$a(c+d+e+g)+b(c+d+e+f)+$ $(c+d)(dg+e+f)$	30	23	31	38	24	24

Table 4.1: The change in size that results from applying the Printform transformations to the canonical forms of several expressions. The variable ordering used is alphabetical.

The Printform transformations do crude factoring while maintaining a weakened version of Condition 1. The variables in the **if**-part are required to be disjoint from the variables in the **then**- and **else**-parts, but are not required to come earlier in the variable ordering. This weakened form of Condition 1 is still enough to guarantee that an expression is prime and irredundant.

Another set of transformations (the `LocalFactor` transformations) are similar, but allow some duplication of variables between **if**-part and the **then**- and **else**-parts. The `LocalFactor` transformations are far more powerful, and provide most of the improvements obtained by my factoring programs, but do not guarantee that the result is prime and irredundant.

By rearranging the if-then-else DAGs (converting them to a non-canonical form), the `Printform` transformations increase the number of times the constants TRUE and FALSE appear, without increasing the number of nodes in the DAG, thus decreasing the size of the DAG by most of the measures. The Printform transformations may increase the size and count measures of an if-then-else DAG, because rearranging the **then**-part may reduce the amount of sharing with the **else**-part. For example, the canonical form for $(c+d)(dg+e+f)$ is (**if** c **then** (**if** d **then** $e+f+g$ **else** $e+f$) **else** $d(e+f+g)$), which has size 13, count 10, and pcount 13. After applying the Printform transformations, we get (**if** c **then** $dg+e+f$ **else** $d(e+f+g)$), which has size 16, count 10, and pcount 10.

The pcount measure, which estimates the complexity of the printed form, is always reduced by the transformations, as one would expect from transformations originally intended for improving printing. Unfortunately, the pcount measure does not correlate particularly well with area. The Printform transformations are still valuable as a factoring tool, as they enable the more powerful `LocalFactor` transformations to find factorings ($(c+d)(dg+e+f)$ for the above example).

Table 4.1 shows the sizes of the results of applying the `Printform` transformations to the canonical forms for a few simple expressions.

The Printform transformations re-order the variables in the if-then-else DAG, and can re-order the variables differently in the **then**- and **else**-parts, and are thus potentially more powerful than simply re-ordering variables (a technique suggested by Randal Bryant [Bry86, page 26]). A combination approach using transformations and heuristics for doing complete variable re-ordering of if-then-else DAGs has yielded the most powerful factoring techniques, but appears to be too slow to be practical.

We are still investigating ways to get good variable orderings and apply them quickly. An exponential algorithm for finding the best ordering for BDDs is given in [FS87]. More recently, register allocation algorithms have been proposed as a way to order variables heuristically [Ber88b]. We are investigating the possibility of adapting these techniques to if-then-else DAGs.

LocalFactor transformations

A full description of the rather ad hoc `LocalFactor` transformations would lengthen this paper significantly, but a quick, sketchy overview is possible. The basic idea is to simplify (**if** a **then** b **else** c) when b implies c, c implies b, b implies $\neg c$, or $\neg b$ implies c. For example, if c implies b, (**if** a **then** b **else** c) can be factored as $ab + c$. The transformations also try to decompose b and c as $c = xy$ and $b = x + y$, recognizing (**if** a **then** b **else** c) as the symmetric function $ax + ay + xy$, which can be factored in several ways, including $a(x + y) + xy$, $(a + xy)(x + y)$, and $a(xxory) + xy$. These symmetric decompositions appear to be particularly useful for factoring arithmetic expressions.

The `LocalFactor` procedure also checks to see whether reducing to canonical form or applying the `Printform` transformations will reduce the complexity of the expression. As an option, the `LocalFactor` procedure will apply don't-care information as described in Section 4.4. The don't-care option was not used for the preliminary results reported here, as the implementation is still too slow.

Table 4.2 gives the area and delay obtained by several different factoring techniques. For each technique, the results were mapped to the `msu.genlib` library with the misII mapper commands `map -m1; phase -g`. The first column is the result of mapping the original problem specification, without any attempt at factoring. The second column is obtained by running the default script provided with misII release 2.0. The third column is the result of applying the `Printform` transformations to the input. The `Printform` transformations often are worse than doing nothing at all, partly because they are designed to optimize the `pcount` metric, which is a poor predictor of area, and partly because they are applied separately to each output, sometimes eliminating sharing that already exists. The fourth column is the result of applying the `LocalFactor` transformations.

Two of the columns of Table 4.2 are compared in Figure 4.5, which is a scatter diagram of the ratio of the areas versus the ratio of the delays for the misII and `LocalFactor` factoring techniques. The upper right quadrant contains the examples for which the misII script is clearly superior, and the lower left quadrant contains the examples for which local factoring is clearly superior. Neither method is universally superior to the other, suggesting that a combination of approaches may be a fruitful area for exploration. We are particularly interested in exploring a combined technique that uses rectangle covering to select common subexpressions from the operands of associative operations found by local factoring.

5 Conclusions and current work

This paper has presented a new way of looking at if-then-else DAGs, based on two-cuts in a corresponding binary decision diagram. The relationship between the two representations leads to a natural canonical form for if-then-else DAGs.

If-then-else DAGs are attractive for VLSI CAD work, because they provide sharing of subexpressions, a compact canonical form for tautology checking, easy manipulation for all the standard Boolean operators, straightforward representation of networks of gates, and factoring by simple transformations of the DAG.

Several implementations of binary decision diagrams and if-then-else DAGs have been written, and used for switch-level verification tasks [Kar86] and for multi-level logic minimization. Current work focusses on using transformations of if-then-else DAGs to do logic minimization. In addition to the Printform transformations described in this paper, more powerful transformations, which violate even the weakened version of Condition 1, have been implemented. These transformations, together with heuristics for finding good variable orderings for arbitrary if-then-else DAGs, have given some promising results, but are still under intensive development.

Other researchers have found that using don't-care sets provides significant improvement in multi-level minimizers. A transformation that simplifies expressions modulo don't-care sets has been implemented and is being debugged and evaluated. The transformation is intended to guarantee that an if-then-else DAG is prime and irredundant with respect to the don't-care set—that is, that any change in meaning occurs only for cases that we know are unimportant.

name	no factor		misII		Printform		LocalFactor	
	area	delay	area	delay	area	delay	area	delay
5xp1	2552	10.0	2080	9.8	2456	9.6	1680	9.6
5xp1-hdl	1264	13.6	1152	14.8	1280	20.2	1336	16.8
9sym	4104	12.2	3848	14.0	4736	14.8	1184	13.6
9sym-hdl	2248	21.4	2152	20.8	2984	23.0	1168	13.0
9symml	3824	11.6	3464	15.8	3864	12.0	1184	13.6
C17	144	3.8	168	4.8	144	3.8	136	2.4
C499	7712	18.6	7648	25.6	10480	36.6	10512	38.2
alupla	2288	16.2	2520	18.4	2792	16.8	2768	15.6
aralis	4800	12.8	2896	11.2	4720	12.2	4688	12.4
aralis1	3992	10.6	2168	16.2	3240	16.4	2000	9.4
b9	2376	9.6	2096	8.4	2472	10.2	2144	10.4
bitcount.clp	744	5.8	480	6.4	528	5.6	528	5.6
bw	4456	13.8	3264	13.4	4392	13.8	4064	11.2
clip-ab.clp	4136	12.4	2168	11.2	5032	15.8	3136	12.6
con1	368	5.2	328	4.0	384	4.8	400	5.0
duke2	11416	18.6	6440	15.2	13568	19.6	13072	19.0
f2	416	4.6	416	4.6	400	4.4	416	5.4
f51m	2392	9.6	2176	10.0	2536	10.8	1448	9.6
f51m-hdl	1208	10.4	1096	15.6	1200	17.2	1112	15.4
ftest	25456	39.2	6160	20.2	27152	24.8	5784	10.4
ftest.clp	7952	14.6	4952	17.2	10216	17.0	5328	13.0
legan.clp	248	3.6	168	4.8	256	4.8	168	4.8
misex1	1464	6.8	1104	7.8	1384	9.0	1288	7.6
misex2	2520	6.6	1760	8.2	2576	7.0	2472	7.2
misex3	16560	19.8	9544	25.8	14696	23.2	14656	21.2
misex3c	11680	17.6	8944	15.6	15176	20.4	13584	23.2
rd53	904	7.0	920	6.6	904	8.0	696	8.0
rd53-hdl	760	10.8	856	12.4	960	10.8	696	8.0
rd73	3104	11.2	1992	10.8	2320	15.0	1368	12.6
rd73-hdl	1232	13.6	1192	13.6	1752	14.8	1408	12.6
rd84	6152	16.2	3528	14.4	5592	17.6	1952	14.8
rd84-hdl	1696	16.6	1696	16.6	2392	19.0	1888	15.0
rot	12568	31.0	11640	28.4	15112	32.2	?	?
sao2	2976	13.2	2664	10.4	3712	13.4	3528	13.4
sao2-hdl	5280	46.6	3008	30.4	5760	41.2	2984	13.6
seq	53384	43.4	?	?	60000	45.6	51784	47.2
vg2	3752	10.4	1504	11.8	5000	10.2	4872	13.8
xor5.clp	176	3.4	160	3.4	176	4.6	176	4.6
z4ml	1152	9.4	624	10.8	1432	9.0	1048	8.2
z4ml-hdl	848	9.4	888	11.2	968	9.6	1000	10.0

Table 4.2: Area and delay from misII's technology mapper for four different factoring techniques.

Several size measures, including the ones presented in this paper, are being evaluated to determine which ones provide the best predictors of circuit size and delay after technology mapping. Ideally, we would like to generate an appropriate measuring function directly from a description of the technology.

Because sum-of-products representations are so important for PLA implementation, we are working on methods to convert prime, irredundant if-then-else DAGs into prime, irredundant sum-of-products representations.

Figure 4.5: A graphical comparison between the misII and `LocalFactor` factoring techniques.

Acknowledgements

I would like to thank Liisa Raiha, for implementing a version of Bryant's canonical if-then-else DAGs for me; Dipen Moitra, for examining the graph theoretic properties of if-then-else DAGs and identifying dominators as an interesting feature; Habib Krit, for studying my undocumented code for general if-then-else DAGs, reading early drafts of [Kar91], and suggesting several improvements to both the code and the writing; and Giulia Pagallo, whose research in learning Boolean expressions inspired the sum-of-products to if-then-else DAG conversion technique described in Section 4.2.

References

[Ake78] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[BBH+88] Karen A. Bartlett, Robert K. Brayton, Gary D. Hachtel, Reily M. Jacoby, Christopher R. Morrison, Richard L. Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. Multilevel logic minimization using implicit don't cares. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 7(6):723–740, June 1988.

[BCDH86] Karen Bartlett, William Cohen, Aart De Geus, and Gary Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-5(4):582–596, October 1986.

[Ber88a] UC Berkeley. Berkeley logic interchange format (BLIF). In Rick Spickelmier, editor, *Oct Tools Distribution 2.1*. Electronics Research Laboratory, University of California, Berkeley, 25 March 1988.

[Ber88b] C. Leonard Berman. Circuit width, register allocation, and reduced function graphs. Technical Report RC 14129, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1988.

[BHL+87] D. Bostick, Gary D. Hachtel, Michael R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft. The Boulder Optimal Logic Design system. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 62–65, Santa Clara, CA, 9–12 November 1987. IEEE Computer Society Press.

[BHMS84] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[Bra87] Robert K. Brayton. Algorithms for multi-level logic synthesis and optimization. In G. De Micheli, Alberto Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design Systems for VLSI Circuits—Logic Synthesis and Silicon Compilation*, pages 197–247. Martinus Nijhoff Publishers, 1987.

[BRSW87] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: a multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1081, November 1987.

[Bry85] Randal Everitt Bryant. Symbolic verification of MOS circuits. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 419–438. Computer Science Press, 1985.

[Bry86] Randal Everitt Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry88] Randal Everitt Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication, 27 September 1988. Unpublished paper.

[DGR+87] Ewald Detjens, Gary Gannot, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Technology mapping in MIS. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 116–119, Santa Clara, CA, 9–12 November 1987. IEEE Computer Society Press.

[FS87] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *ACM IEEE* 24^{th} *Design Automation Conference Proceedings*, pages 348–355, Miami Beach, FL, 28 June–1 July 1987.

[HL87] Gary D. Hachtel and Michael R. Lightner. Don't care conditions in top down synthesis. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 316–319, Santa Clara, CA, 9–12 November 1987. IEEE Computer Society Press.

[Kar86] Kevin Karplus. Exclusion constraints, a new application of graph algorithms to VLSI design. In *4th MIT Conference on Advanced Research in VLSI*, pages 123–139, Cambridge, MA, April 7–9 1986.

[Kar91] Kevin Karplus. BOOLE, an abstract data type for manipulating Boolean expressions. Technical Report UCSC-CRL-91-??, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064, forthcoming, probably 1991.

[Karon] Kevin Karplus. Using If-Then-Else DAGs for multi-level logic minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, in preparation.

[KM90] Kevin Karplus and Dipen Moitra. Using dominators for Boolean function manipulation. Technical report, Department of Computer Science, Cornell University, Ithaca, NY 14853, forthcoming, probably 1990.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, July 1959.

[MWBS88] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *IEEE International Conference on Computer-Aided Design ICCAD-88*, pages 6–9, Santa Clara, CA, 7–10 November 1988.

[NB86] Ravi Nair and Daniel Brand. Construction of optimal DCVS trees. Technical Report RC 11863, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 19 March 1986.

[RI86] Douglas S. Reeves and Mary Jane Irwin. Functional verification of digital MOS circuits. In *IEEE International Conference on Computer-Aided Design ICCAD-86*, pages 496–499, Santa Clara, CA, 11–13 November 1986. IEEE Computer Society Press.

[SF86] Kenneth J. Supowit and Steven J. Friedman. A new method for verifying sequential circuits. In *ACM IEEE* 23^{rd} *Design Automation Conference Proceedings*, pages 200–207, Las Vegas, NV, 29 June–2 July 1986.