

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

Kevin Karplus

Board of Studies in Computer Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064

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. 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.

1 What is multi-level logic minimization?

Multi-level logic minimization is the transformation of a specification of a Boolean function into an equivalent representation that can be implemented as a circuit with better characteristics (smaller, faster, or more testable) than a circuit built from the original specification. The function usually has multiple outputs, and may be only partially specified.

Most previous work in multi-level logic synthesis is based on extensions of two-level (sum-of-products) minimization for PLAs [7, 6, 2, 14, 3]. A notable example is the misII multi-level minimization system [9], based on the espresso two-level minimizer [8].

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 XOR operations look more attractive in the if-then-else DAG form (see Section 2) than in sum-of-products form. We are investigating if-then-else DAGs

for multi-level logic minimization, and have some encouraging preliminary results.

Section 2 describes the if-then-else operator, which forms the basis for the representation used, and gives a quick introduction to binary decision diagrams and if-then-else DAGs. Section 3 introduces a new canonical form. Section 4 discusses ways to estimate the area and delay of a circuit in a technology-independent logic minimizer. Section 5 discusses conversion from networks of gates to if-then-else DAGs. Section 6 talks about ways to use don't-care information for simplifying if-then-else DAGs. Section 7 describes some crude factoring techniques that do surprisingly well.

2 Binary decision diagrams and if-then-else DAGs

The research described here is based on a single 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 $ab + a'c$ or, equivalently, $(a + c')(a' + b)$.*

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

- $ab = (\text{if } a \text{ then } b \text{ else FALSE})$
- $a + b = (\text{if } a \text{ then TRUE else } b)$
- $a \oplus b = (\text{if } a \text{ then } b' \text{ else } b)$.

If-then-else trees and DAGs have a long history [17, 1, 10]. 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 DAGs*, 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 often for logic verification work [12, 20, 21, 18]. They are attractive for such work as they are easy to manipulate and have a convenient canonical form (Bryant's canonical form) [10]. They have also been used

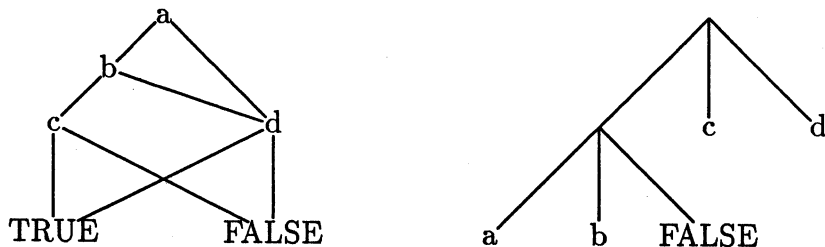


Figure 2.1: BDD and if-then-else DAG for $abc + a'd + b'd$, factored as $(\text{if } ab \text{ then } c \text{ else } d)$.

for logic synthesis work, mainly for designing differential voltage-cascade switches, which implement BDDs directly [19, 13]. 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)).

Negating an expression represented as an if-then-else DAG requires negating all the leaves. To simplify the negation operation, 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.

There is a natural mapping from if-then-else DAGs to BDDs, which can be defined recursively. The **if**-, **then**- and **else**-parts are each mapped to the corresponding BDDs, and the common subBDDs of the **then**- and **else**-parts are merged. All pointers to TRUE in the image of the **if**-part are changed to pointers to the image of the **then**-part, and all pointers to FALSE are changed to pointers to the image of the **else**-part. The leaves of the if-then-else DAG are mapped to the obvious corresponding BDDs. Note that the mapping is many-to-one, with different if-then-else DAGs corresponding to different two-cuts in the BDD [15].

Figure 2.1 shows a BDD and an if-then-else DAG for the expression $abc + a'd + b'd$. Note that the if-then-else DAG represents the expression as $(\text{if } ab \text{ then } c \text{ else } d)$, allowing the ab term to be shared with other expressions.

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.
- 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. This means that 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 have two convenient canonical forms: Bryant's canonical form (as in BDD's) and a new canonical form presented in Section 3. 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.

For example, $ab(d+e)$ is (if (if a then b else FALSE) then (if d then TRUE else e) else FALSE), $c(d+e)$ is (if c then (if d then TRUE else e) else FALSE), and abd is (if (if a then b else FALSE) then d else FALSE). These three functions share the subexpressions $ab = (\text{if } a \text{ then } b \text{ else FALSE})$ and $(d+e) = (\text{if } d \text{ then TRUE else } e)$, while the BDD representations would share only $(d+e)$.

- If-then-else DAGs are a more factored form than BDDs, providing for better printing and logic minimization. With the aid of the transformations described in Section 7, good factorings can often be found from the canonical forms or from arbitrary non-canonical expressions.
- Boolean operations can be computed as if-then-else triples, so that the symbol table used for storing canonical forms can be used for caching the results of operations as well.

3 A canonical form for if-then-else DAGs

A representation is *canonical* if any two expressions that are logically equivalent are identical. For example, if $ab + ab'$ is represented differently from a , then the representation is non-canonical.

Using canonical forms makes checking for equivalence easy—unfortunately, conversion from a non-canonical form to canonical form

