Advanced Research in VLSI Caltech 89

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,

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

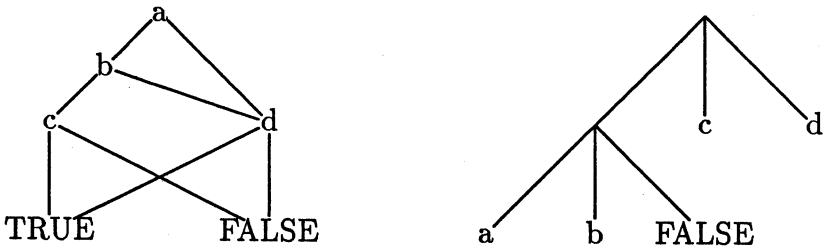

Figure 2.1: BDD and if-then-else DAG for $abc + a'd + b'd$, factored as (**if** ab **then** c **else** d).

for logic synthesis work, mainly for designing differential voltage-cascode 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 (**if** ab **then** c **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 =(**if** a **then** b **else** FALSE) and $(d+e)$ =(**if** d **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

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 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 [11]. The same arguments can be applied to the new canonical form described here. 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.

To make 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. A total ordering is imposed on the atoms of the expression, and all the atoms in the **if**-part must be earlier in the order than all atoms in the **then**- and **else**-parts. 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 in the corresponding BDD, but not enough to make the if-then-else DAG 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 (**if** a **then** b **else** c) and (**if** a' **then** c **else** b) and between (**if** a **then** b **else** c) and (**if** a **then** b' **else** c')$'$. 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. This corresponds to Bryant's choice of atoms as node labels (never negations of atoms).
4. Triples of the form (**if** a **then** TRUE **else** FALSE) and (**if** a **then** FALSE **else** TRUE) are prohibited. The first triple should be represented simply as a, and the second one by a'.
5. Triples of the form (**if** TRUE **then** b **else** c) and (**if** FALSE **then** b **else** c) are prohibited, and should be replaced with b and c respectively.

6. In the triple (**if** a **then** b **else** c), b and c must not share both **then**- and **else**-parts. If b = (**if** b_a **then** b_b **else** c_c) and c = (**if** c_a **then** b_b **else** c_c), then the correct representation is (**if** (**if** a **then** b_a **else** c_a) **then** b_b **else** c_c). If b = (**if** b_a **then** b_b **else** b_c) and c = (**if** c_a **then** b_c **else** b_b), then use (**if** (**if** a **then** b_a **else** c'_a) **then** b_b **else** b_c).

7. In the triple (**if** a **then** b **else** c), b must not contain c as a **then**- or **else**-part. If b = (**if** b_1 **then** b_b **else** c) or b = (**if** b_2 **then** c **else** b_c), then the expression should be represented as (**if** (**if** a **then** b_1 **else** FALSE) **then** b_b **else** c) or (**if** (**if** a **then** b_2 **else** TRUE) **then** c **else** b_c). If c is one of the constants TRUE or FALSE, this restriction amounts to choosing left-associativity for commutative AND or OR operations. The symmetric test for c = (**if** c_1 **then** c_b **else** b) or c = (**if** c_2 **then** b **else** c_c) is also needed.

We can show that imposing the restrictions listed above defines a canonical form by exhibiting an isomorphism with Bryant's canonical form [15]. We can use essentially the same algorithm for converting to either Bryant's canonical form or the new form [15].

3.1 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 [8, page 28], [7, 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 can be modified by changing a literal to TRUE without changing the meaning of the expression. Similarly, an expression in sum-of-products form is *irredundant* if no term can be changed to FALSE without changing the meaning of the expression.

Definition 4: *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 [15].

Both Bryant's canonical form and the new canonical form presented in Section 3 can be shown to be prime and irredundant with the definition presented here [15].

4 Expression complexity, counting literals

When doing logic minimization, the first question is "what exactly is being minimized?" The goal of minimization is to reduce the area, power, or delay of the final circuit after technology mapping, but these costs are dependent on the technology used, and optimizations tied to a particular cell library quickly become obsolete.

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.

The usual way to estimate the area for a network of gates is to estimate the cost for each gate and sum the estimates. The most popular gate area estimators are the number of literals in sum-of-products form and the number of literals in the factored form [7, page 235]. The literal count corresponds closely to the number of transistor pairs needed to implement the function as a static cMOS gate, and is an excellent area estimator if the mapper does not change the decomposition of the circuit. The estimate is not as good when the mapper splits or merges gates.

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; 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). The sum of literals in factored form is a good predictor, with the

ratio of actual area over predicted area having a standard deviation of 17.4% of the mean.

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. The standard cost functions do not have this property, and a cost estimator that hides such inverters should be a better estimator of final area. Subtracting the number of nodes in a network from the sum of literals makes inverters free, and has the added advantage of making the measure less sensitive to the size of the gates used in the decomposition. This measure is an excellent predictor, having a standard deviation of only 12.6% of the mean.

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

We have experimented with several 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 needed if each if-then-else triple were expanded to AND, OR, or XOR gates, and any shared nodes were duplicated.

count a recursively defined function that attempts to match the values of the 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.

$count(x)+count(y)+count(z)$ 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.

$count(x) + count(y) + count(z) + 1$ 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 13.1%.

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 31.4%. 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.

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

Most of the standard benchmarks are available in a standard format, the Berkeley Logic Intermediate Format (BLIF) [4], 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.
- 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 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.

6 Using don't-care information

A substantial part of 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 [3, 14].

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 simplify $e =$ (**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 $d + a'$,
- we can simplify c with the don't-care expression $d + a$,
- and we can simplify a with the don't-care expression $d + (b \oplus c)'$. 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 3.1.

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 expression for shared subexpressions by ANDing together the don't-care expressions derived from each usage. This extension has not yet been implemented.

7 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.

7.1 `Printform` transformations

The `Printform` 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. They do crude factoring while maintaining the weakened version of Restriction 1. This weakened form is still enough to guarantee that an expression is prime and irredundant. The complete set of transformations is presented in [15].

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 in the *pcount* measure. 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 is always reduced, but 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, with size 10, count 6, and pcount 6).

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 [10, page 26]). Unfortunately, the restriction that no variable from the **if**-part appears in the **then**- and **else**-parts means that integer multiplication still requires an exponential representation.

7.2 `LocalFactor` transformations

The `LocalFactor` transformations have some similarity to the `Printform` transformations, but allow some duplication of variables between the **if**-part and the **then**- and **else**-parts. This duplication means that the result is not necessarily prime and irredundant.

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 c', or 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 carry function $ax+ay+xy$, which can be factored in several ways, including $a(x+y)+xy$, $(a+xy)(x+y)$, and $a(x \oplus y)+xy$.

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 6. The don't-care option was not used for the preliminary results reported here, as the implementation is still too slow.

Table 7.1 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;`

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
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
aralis1	3992	10.6	2168	16.2	3240	16.4	2000	9.4
f51m	2392	9.6	2176	10.0	2536	10.8	1448	9.6
rd53-hdl	760	10.8	856	12.4	960	10.8	696	8.0
sao2-hdl	5280	46.6	3008	30.4	5760	41.2	2984	13.6

Table 7.1: Area and delay from misII's technology mapper for four different factoring techniques. Only examples in which `LocalFactor` is better than the standard misII script are shown. A more complete table can be found in [16].

`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.

Figure 7.1 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.

7.3 Reordering transformations

The `Printform` and `LocalFactor` transformations re-order the variables in the if-then-else DAG. Both can re-order the **then**- and **else**-

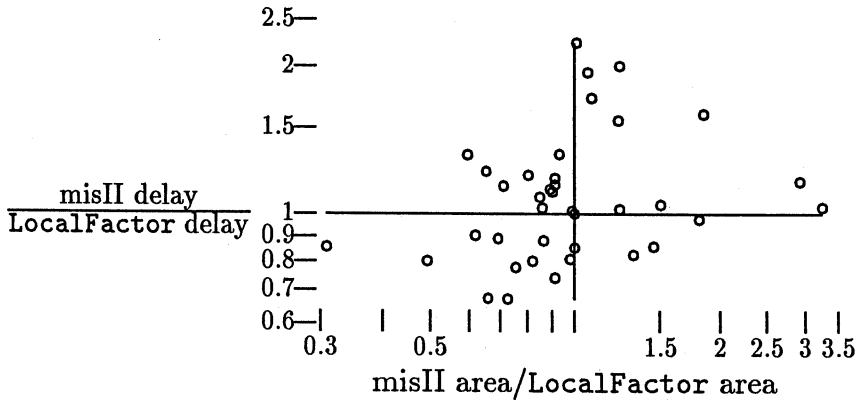

Figure 7.1: Relative effectiveness of the standard misII script and the `LocalFactor` technique.

parts differently, making them potentially more powerful than simply changing the total order used for the canonical form. Sometimes, however, imposing a well-chosen total order allows more sharing between **then**- and **else**-parts, so we have experimented with ways to choose a good total order.

We have had some promising results using the variable ordering obtained by doing a depth-first search of the output of `LocalFactor`. Generating a canonical form with the new order and reapplying `LocalFactor` often decreases the complexity of the DAG. The process of re-ordering, local factoring, and finding a new order can be repeated until no further improvements are made. Using several random starting orders, we have found some very good factorings for small adders. Unfortunately, the implementation is still too slow and memory-intensive to run on large examples.

An exponential algorithm for finding the best ordering for BDDs is given in [13]. More recently, register allocation algorithms have been proposed as a way to order variables heuristically [5]. We are investigating the possibility of adapting these techniques to if-then-else DAGs.

8 Conclusions and current work

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 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.

Some factoring transformations have been developed to reduce the complexity of if-then-else DAGs, and other, more powerful techniques are under investigation.

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 guarantees that an if-then-else DAG is prime and irredundant with respect to the don't-care expression—that is, that any change in meaning occurs only for cases that we know are unimportant.

Acknowledgements

This research was partially supported by NSF grant DCR-8503262 and IBM Faculty Development award. 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; Habib Krit, for studying my code for if-then-else DAGs, and suggesting several improvements; Giulia Pagallo, whose research in learning Boolean expressions inspired the sum-of-products to if-then-else DAG conversion technique described in Section 5; and Søren Søe for reading drafts of this paper.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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, IEEE Computer Society Press, Santa Clara, CA, 9–12 November 1987.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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, IEEE Computer Society Press, Santa Clara, CA, 9–12 November 1987.

[15] Kevin Karplus. *Representing Boolean Functions with If-Then-Else DAGs.* Technical Report UCSC-CRL-88-28, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064, December 1988.

[16] Kevin Karplus. *Using If-then-else DAGs for Multi-level Logic Minimization.* Technical Report UCSC-CRL-88-29, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064, December 1988.

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

[18] 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.

[19] 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.

[20] 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, IEEE Computer Society Press, Santa Clara, CA, 11–13 November 1986.

[21] 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.

Advanced Research In VLSI

Proceedings of the Decennial Caltech Conference on VLSI, March 1989

edited by
Charles L. Seitz

The MIT Press
Cambridge, Massachusetts
London, England

Contents