Logic Decomposition during Technology Mapping

Eric Lehman and Yosinori Watanabe
Digital Equipment Corporation
77 Reed Road, Hudson, MA 01749, USA

Abstract

This paper presents a procedure which performs logic decomposition during technology mapping.

A problem in technology mapping is that quality of the final implementation depends significantly on the initially provided circuit structure. This problem is critical especially for mapping with tight and complicated constraints. Conventional techniques iteratively apply technology independent transformations and technology mapping, so that the implementation obtained by technology mapping is restructured and remapped. Although some progress can be made, the effectiveness of these techniques is limited, since when a circuit is restructured, it is not clear how it is implemented eventually.

The central problem is that technology independent transformations and technology mapping are applied separately. In this paper, we propose a procedure which simultaneously applies technology mapping and algebraic logic decomposition, a key technology independent operation for changing circuit structures. A set of circuit structures is compactly encoded in a single graph, and the procedure dynamically modifies the set during technology mapping by introducing new structures while deleting others based on the actual cost function used in the mapping. State-of-the-art technology mapping algorithms are naturally extended, so that the procedure finds an optimal tree implementation over all the circuit structures examined. We show that the procedure effectively explores the same solution space obtained by applying algebraic decomposition exhaustively. However, the run time is proportional to the size of the graph, which is typically logarithmic in the number of decompositions.

The procedure has been implemented and used for commercial design projects. We present experimental results on benchmark examples to demonstrate its effectiveness.

1 Introduction

Technology mapping is a crucial phase in logic synthesis, since it determines quality of the final implementation. However, technology mapping is usually applied on a particular structure (decomposition) of logic expressions. Therefore, quality of the final implementation could severely depend upon the structure initially provided to the technology mapper.

The most common approach for generating the initial structure of logic expressions for technology mapping consists of two phases: technology independent transformations and subject-graph generation [2]. In the first phase, logic expressions are optimized in a technology independent manner. The result is represented by a graph, which is sometimes called a *boolean network* [2]. Then in the second phase, the resulting expressions are converted into a special type of boolean network, where nodes represent particular functions such as AND or INVERTER. Such a graph is called a *subject graph* [11] and the functions used in a subject graph are called *base functions*. Technology mapping is the third and last phase of synthesis, where the subject graph is *mapped* onto a set of library gates to generate the final implementation.

Synthesis techniques based on these three phases have an obvious drawback: optimizations applied in each of the phases are disconnected. Although decisions made in the first and the second phases critically affect the final results, it is not clear in these phases how the resulting logic expressions are actually implemented. On the other hand, in technology mapping, real data on constraints and library gates are available, but the degree of freedom for the final implementation is limited by the decisions made in the preceding phases. This drawback is fatal when one needs to handle tight and complicated constraints with delay, area, and load. Conventionally, this problem has been addressed by the following two approaches [2]. One is rule-based techniques. A rule-based system, e.g. [4, 8], can combine technology independent transformations and technology mapping. However, by the nature of rule-based techniques, transformations are made only locally, and in general it is difficult to explore large solution space as is done by algorithmic techniques. The other approach repeats the three phases. A variety of techniques have been proposed in the literature [1, 7, 18, 6, 13, 19, 21, 16]. Some of them globally change structures, while using data on delay or area of the implementation generated by technology mapping. Even though these techniques can yield improvements to some extent, the effectiveness is still limited because the disconnection between the first two and the third phases cannot be fully resolved. Data from the initial mapping are quickly invalidated by operations of the first and the second phases, and cease to be an effective guide. Moreover, without information about library gates, one can foresee little about the final implementation, and thus transformations are made somewhat blindly.

The central problem is that technology independent transformations and technology mapping do not cooperate. This is the problem addressed in this paper. Among technology independent transformations, our focus is on algebraic logic decomposition [3], a key operation for changing circuit structures. We asked ourselves if it is possible to apply algebraic logic decomposition and technology mapping simultaneously. The answer is yes, and we present a procedure which effectively explores the entire solution space that would be obtained if one could apply algebraic decomposition exhaustively. The proposed procedure compactly encodes a *set of decompositions (structures)* into a single graph, and applies a technology mapping algorithm on it. In the meanwhile, it dynamically modifies the set of decompositions encoded in the graph by introducing new decompositions while deleting others based on the actual cost function used in technology mapping. It is guaranteed that the procedure finds an optimal solution among all the decompositions examined.

The graph for representing a set of decompositions is called a *mapping graph*. It is obtained as the union of the subject graphs representing the decompositions, where nodes of the subject graphs are merged as much as possible. Each subject graph corresponds to a single decomposition, where we use two-input ANDs (AND2s) and inverters for base functions. Such decompositions are called *AND2/INV decompositions*. A mapping graph has a property that the number of decompositions that can be represented in it is exponential in the size of the graph. According to our experiments, for a sum-of-products expression f with n product terms, where each product term has k literals, we can encode all AND2/INV decompositions of f in a mapping graph when n and k are at most 10. Note that there are $D(n)D(k)^n$ such decompositions in general, where $D(k) = \frac{1}{2k-1}\frac{(2k)!}{k!2^k}$. State-of-the-art technology mapping techniques, e.g. [11, 20], can be naturally extended for mapping graphs, retaining linear run time and optimal results[1].

A set of AND2/INV decompositions are generated through iterative local transformations defined on a mapping graph. We define three such transformations: associative transformation, distributive transformation, and inverter transformation. The associative transformation is based on the associative law, i.e. two AND2s representing $a(bc)$ are transformed into two AND2s for $(ab)c$ and vice versa. Similarly, the distributive transformation is based on the distributed law, and transforms $ab + ac$ to $a(b + c)$. The inverter transformation adds or removes two consecutive inverters between two consecutive nodes in a mapping graph. These transformations are the vehicle to introduce new decompositions.

In this paper, we examine two sets of AND2/INV decompositions. The first is the closure of the set of AND2/INV decompositions of a given boolean network η under the associative and inverter transformations. Namely, the two transformations are applied on those AND2/INV decompositions until nothing new is found. The result is the closure under these transformations, and is denoted by Λ_η. We show that the second phase of the conventional logic synthesis approach, where a boolean network is converted to a subject graph, is completely subsumed by Λ_η, i.e. an arbitrary AND2/INV decomposition obtained by converting η is contained in Λ_η. Therefore if Λ_η is encoded in a mapping graph and technology mapping is applied, an optimal result over all possible AND2/INV decompositions for η can be obtained. We show a procedure which, instead of iteratively applying those transformations, directly encodes Λ_η into a mapping graph. This technology mapping procedure is referred to as the Λ-*mapping* procedure.

The second set of AND2/INV decompositions we examine is an even larger closure. It is the closure of Λ_η under the distributive transformation, which we denote by Δ_η. We show that Δ_η subsumes the second phase as well as a key part of the first phase, i.e. algebraic logic decomposition. In other words, no matter how algebraic logic decomposition is applied on a given boolean network η in the first phase, and no matter how the result is converted to an AND2/INV decomposition in the second phase, the resulting decomposition is contained in Δ_η. Therefore, the technology mapper finds an optimal result over all possible AND2/INV decompositions given through all possible algebraic decompositions. We call this procedure Δ-*mapping*. Unlike the Λ-mapping procedure, Δ-mapping dynamically applies the distributive transformation during technology mapping, so that new decompositions are introduced on the fly. It also deletes some decompositions, but does so only when it is known that at least equally good implementations can be obtained without them, and thus quality of the final result is not affected. Therefore, it controls the size of the set of decompositions encoded in a mapping graph, and effectively explores the entire Δ_η.

Both the Λ and Δ mapping procedures have been implemented, where an extension was made to handle sequential circuits, with which logic decompositions across latches, as well as retiming possibilities, can be also taken into account. They are being used for real design projects, where Λ-mapping is first used, and then Δ-mapping is applied on timing-critical regions to further speed up the final implementations. We show that both procedures are feasible in terms of complexity for most practical examples, and demonstrate their effectiveness on benchmark examples.

This paper is organized as follows. In Section 2, we define basic terminology. Section 3 introduces a mapping graph, and show how technology mapping is applied on it. The Λ-mapping procedure is also presented in this section. Section 4 introduces Δ-mapping, and describes how to improve efficiency of the procedure. Experimental results are given in Section 5, and Section 6 concludes the paper.

[1] As of ordinary techniques, the generalized technology mapping is also tree-based, and thus optimality is claimed on tree implementations. A heuristic is applied to generate non-tree implementations.

2 Preliminaries

A *boolean network* is a directed acyclic graph. A node n has one of three types. If n is a *primary input*, it has zero in-degree and an associated variable y_n. If n is *internal*, it has an associated variable y_n and sum-of-products expression f_n. If n is a *primary output*, it has zero out-degree and in-degree of 1. A boolean network may not contain a directed cycle.

The expression f_n contains variable y_m whenever there is a directed arc from node m to node n. In this case, m is called a *fanin* of n and n a *fanout* of m. The *transitive fanin* of node n is the set of nodes connected to n by some directed path ending at n (n itself is included). Likewise, the *transitive fanout* of node n is the set of nodes connected to n by some directed path beginning at n.

Two boolean networks η and $\tilde{\eta}$ are *isomorphic* if there is a graph isomorphism ϕ mapping each node n in η to node $\phi(n)$ in $\tilde{\eta}$ such that nodes n and $\phi(n)$ are of the same type and f_n is logically equivalent to $f_{\phi(n)}$, provided each variable y_m in the former is replaced by $y_{\phi(m)}$.

An internal node n in a boolean network can be replaced by a logically equivalent collection of two-input AND nodes (AND2s) and inverters. Working from the sum-of-products expression f_n, each product term is represented by a tree of AND2s. The sum is represented by a tree of AND2s with inverters at the leaves and root. The result is called an *AND2/INV decomposition* of n. An AND2/INV decomposition of a boolean network is formed by replacing each internal node by an AND2/INV decomposition of that node.

A boolean network in which every internal node is either an AND2 or inverter is called an *AND2/INV network*. An AND2/INV decomposition of a boolean network is always an AND2/INV network. Since our technology mapping procedures are tree-based, whether internal nodes in an AND2/INV network are shared or duplicated is irrelevant in this paper. Therefore, it is assumed that all AND2/INV networks are converted by duplication to a canonical form in which every internal node has exactly one fanout.

A *library* is a set of gates, each accompanied by area, delay, and loading characteristics and by a single-output boolean network representing its logic function. A boolean nerwork in which every internal node is associated with a logically equivalent library gate is called a *mapped network*.

A conventional approach to logic synthesis can be divided into three phases: (1) technology independent processing, (2) AND2/INV decomposition, and (3) technology mapping. The inherent flaw in this three-step approach is that quality of the mapped network depends heavily on the structure of the AND2/INV decomposition generated through phases (1) and (2). However, accurate timing, area, and loading information is not available until phase (3). Therefore, the downfall of this traditional approach is that choices in phases (1) and (2) affect results critically, but are made almost blindly.

Though the focus of this paper is combinational logic circuits, our results can be extended as discussed in Section 5.

3 Mapping on a Set of Decompositions

In this section we introduce two basic tools: a *mapping graph* data structure which encodes a set of AND2/INV networks and a *graph-mapping* algorithm which effectively applies tree-mapping [11] to each encoded AND2/INV network. These tools are then combined to construct a technology mapping procedure called Λ-*mapping* which absorbs phase (2) of synthesis into technology mapping. The result of Λ-mapping is at least as good as can be obtained by applying tree-mapping to every possible AND2/INV decomposition of a boolean network.

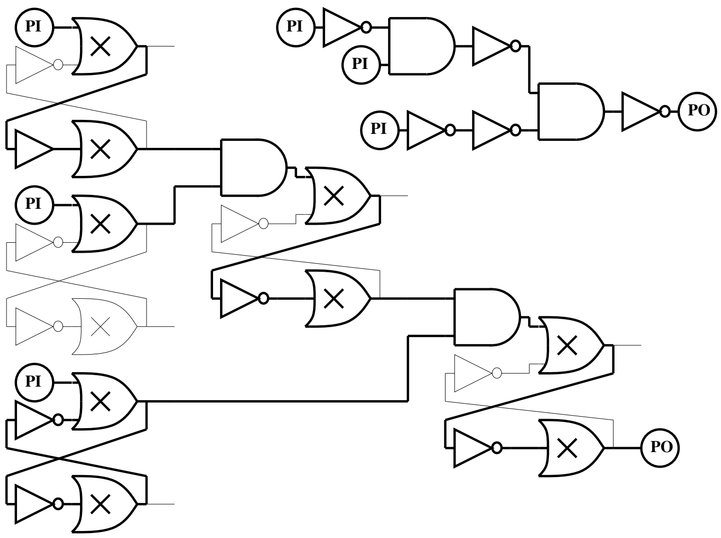

Figure 1: The left diagram is a mapping graph and the right diagram is an AND2/INV network encoded within. The path of the depth-first search which generated the AND/INV network is highlighted on the left.

3.1 Mapping Graph Data Structure

Conceptually, a *mapping graph* is an AND2/INV network with four modifications. An example is given in the left diagram of Figure 1.

First, a new type of node is introduced. A *choice* node b has associated variable y_b and may have arbitrary in- and out-degree. Intuitively, the technology mapper will first view the choice node as a wire connecting the first fanin to the output of the choice node, then as a wire connecting the second fanin to the output, etc. Therefore, several AND2/INV representations of the same boolean function can be supplied to the mapper by making the representations available at the fanins of a single choice node. Formally, we let $y_b = y_c$ whenever c is a fanin of b, and so all fanins of a choice node must be logically equivalent. A choice node is drawn as an OR gate marked with an X.

Second, a mapping graph may contain directed cycles. Directed cycles are useful in specifying that a function f may be expressed in terms of g (creating a directed path from a node computing g to a node computing f) and also vice versa (completing the directed cycle). The technology mapper considers both implementations. A typical example of a directed cycle consists of two inverters and two choice nodes as appears several places in Figure 1. This cycle structure encodes the notion of an arbitrary length inverter chain.

Third, a mapping graph can be partitioned into disjoint subgraphs, such that each subgraph is a *ugate*. This term is defined in Figure 2. This restriction imposes a degree of regularity on the structure of a mapping graph, without sacrificing any expressive power. In practice, this regularity greatly simplifies implementation (see Section 5). A good mental picture of a mapping graph is a directed graph containing interconnected, but isolated clumps of nodes, where each clump is a ugate. In subsequent algorithms, we will associate some values with each ugate instead of with each node.

Fourth, a mapping graph must be *reduced*; there can not exist two choice nodes with logically equivalent outputs. This restriction can be met by merging ugates. If ugates u and v have logically equivalent choice nodes, then copies of the positive and negative AND2s of u are added to the corresponding halves of v. If two AND2s in the same half of v have identical inputs, one is discarded. All fanouts of the positive and negative outputs of u are connected to the corresponding outputs of v. The nodes of u are deleted. (A similar process is used if the positive

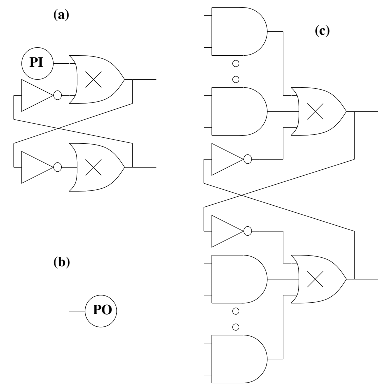

Figure 2: A ugate is a subgraph of a mapping graph. There are three types: (a) primary input (b) primary output (c) AND. For the primary input and AND ugates, the top half of each is the *positive half* and the bottom half is the *negative half*. The top output is the *positive output* and the bottom is the *negative output*. Similarly, there is a *positive inverter* and a *negative inverter*. Every AND ugate contains at least one AND2, but otherwise there may be any number. An AND2 in the top half of an AND ugate is a *positive AND2* and an AND2 in the bottom half is a *negative AND2*. The set of positive AND2s in ugate u is denoted H_u^1 and the set of negative AND2s is denoted by H_u^0.

output of u is logically equivalent to the negative output of v and vice versa.) This restriction not only shrinks the graph, but also provides greater freedom to the mapper by collecting together all known ways of expressing a function with AND2s and inverters. Intuitively, wherever one such expression was required, all are now available.

An AND2/INV network encoded by a mapping graph is defined as follows. Consider a depth-first search rooted at a primary output of the mapping graph with primary inputs at the leaves. At an AND2 node, the search proceeds to both fanins. At a choice node, the search proceeds to only one fanin. The nodes and arcs traversed in the search form a tree, assuming nodes and arcs visited more than once are duplicated. Replace each choice node in the tree by a wire connecting the output of the choice node to the single fanin explored by the search. See Figure 1. Each complete AND2/INV network encoded in a mapping graph is the union of one such tree for each primary output. The set of all complete AND2/INV networks obtainable in this way is the set of AND2/INV networks encoded by a mapping graph.

3.2 Graph-Mapping

This section describes the graph-mapping procedure, an extension of tree-mapping [11]. The algorithm transforms a mapping graph μ into a mapped network in which primary outputs are driven by disjoint trees of gates with primary inputs at the leaves. In Section 5 we describe a heuristic method for sharing logic between trees. This is essential in practice, but in the present theoretical discussion we assume no sharing is used.

3.2.1 Preliminaries

This section reviews three notions used in the graph-mapping algorithm: matching, mapping, and cost.

For a given choice node and library gate, *matching* identifies all single-output subgraphs of μ rooted at the choice node which are logically equivalent to the library gate. Since numerous methods for matching are available [9], we only formalize problem in this section.

An AND2/INV decomposition of the boolean network for a library gate is called a *pattern*. A *match* ϕ of a pattern at internal node a in mapping graph μ is a bijective function such that: (1) ϕ projects each internal node in the pattern to an internal node in μ. (2) ϕ projects the fanin of the pattern's primary output to a. (3) Nodes b and $\phi(b)$ are either both AND2s or else both inverters. (4) For nodes b, c in the pattern there is a directed arc from b to c if and only if there is a directed path from $\phi(b)$ to $\phi(c)$ crossing only choice nodes. (Intuitively, an arc in the pattern may reach across a choice node in the mapping graph.) Every match at internal node a is also considered a match at the choice node which is a fanout of a. Let C be the image of ϕ. The nodes in C and choice nodes between nodes in C are said to be *covered* by the match ϕ. A fanin of a node in C which is not covered by the match is a *fanin* of the match. From this definition, every fanin of a match is a choice node.

A *mapping* at a choice node c in μ is defined as a set of matches. If c is the positive choice node of a primary input ugate, then the set may be empty. Otherwise, the set consists of a match ϕ at c and the union of a mapping at each fanin of ϕ. A mapping uniquely defines a tree of library gates with primary input nodes at the leaves.

Each such mapping has a *cost*. This cost may measure the total area of the tree of gates, the arrival time at the root of the tree, etc. Generally, any notion of cost used in tree-mapping [5, 20] is acceptable[2]. However, to guarantee that graph-mapping terminates, the library and cost notion must be designed so that an optimal implementation exists; more precisely, there can exist no infinite sequence of logically equivalent trees of gates with increasing size and non-increasing cost. At each choice node c in μ, the graph-mapping procedure maintains a value *cost(c)*. This is the cost of the least-cost mapping at c yet found.

3.2.2 The Procedure

This section describes graph-mapping as a four-step procedure.

First, for each choice node c with a fanin which is a primary input, *cost(c)* is assigned an externally-supplied initial value. For example, if cost measures the arrival time at the root of a tree of gates, then this initial value is the input arrival time. For every other choice node c in μ, *cost(c)* is set to infinite.

Second, each ugate u is assigned a number, $label(u)$, such that the following three conditions hold. If there is only a directed path from a node in u to a node in v, then $label(u) < label(v)$. If there is also a directed path from a node in v to a node in u, then $label(u) = label(v)$. If there is no directed path between a node in u and a node in v, then $label(u) \neq label(v)$. This requires a straightforward graph algorithm which assigns increasing labels from inputs to outputs and the same label to all ugates in a cycle.

Third, a forward pass is made through μ. Ugates are visited in order of increasing label and a function `Update` is called on each visited ugate. This function considers mappings at each choice node c in the ugate and updates *cost(c)* whenever a new, lower-cost mapping is found. The cost of a mapping at node c with root match ϕ is computed from the gate g generating the pattern of match ϕ and the costs of mappings at the fanins of ϕ. This is done by using `Cost_Function`. For example, if cost measures the sum of the areas of gates in a tree, then `Cost_Function` should add the area of gate g to the *cost* value stored at each match fanin. (This is analogous to the cost function used in tree-mapping.) `Cost_Function` must be non-decreasing in all input costs to guarantee an optimal solution. If there is set of ugates all with the same label, then `Update` is iteratively called on every ugate in the set until there is a complete iteration during which no *cost* value changes at any choice node in any ugate in the cycle.

Fourth, a second pass is made through μ by working backward recursively from each primary output. The purpose is to construct a mapped network reflecting the *cost* values computed in step three. Several implementations are possible. A simple procedure visits each primary output in turn. Suppose the fanin of a particular primary output is the choice node c in ugate u. `Update(`u`)` is called in a special mode so that when the match which generated *cost(c)* is redicovered, the function halts and returns the match fanins and the gate g which produced the

[2]The present discussion assumes the set of possible costs is totally ordered; like tree-mapping, however, graph-mapping extends to partially ordered costs.

```
procedure  Update(u)

    do
        for each choice node c in u
            for each match φ of a library gate g at c
                let f_1 ... f_k be the fanins of match φ
                let cost(c) = MIN(cost(c),  Cost_Function(g, cost(f_1), ..., cost(f_k)))
    while the cost at some node changed during this iteration
```

Figure 3: Update function

matched pattern. Gate g is added to the mapped network and the procedure is applied recursively to the match fanins.

We now show that graph-mapping applied to a mapping graph produces a result as good as obtained by applying tree-mapping to every encoded AND2/INV network.

Theorem 3.1 *Suppose graph-mapping is applied to a mapping graph μ, and tree-mapping is applied to an AND2/INV network θ encoded in μ. For each primary output, the implementation produced by graph-mapping has cost less than or equal to the implementation produced by tree-mapping with the same* `Cost_Function`.

Proof: We require there exist no infinite sequence of logically equivalent trees of gates with increasing size and non-increasing cost. An infinite loop in step 3 or unbounded recursion in step 4 would imply such a sequence.

Since θ is encoded in μ, an AND2/INV network isomorphic to θ can be generated by a depth-first search from each primary output of μ. This establishes a correspondence between nodes in θ and nodes in μ, and also between matches in θ and matches in μ.

Let a be an internal node or primary input in θ, b be the corresponding node in μ, and c the choice node with fanin b. If graph-mapping is applied to μ and tree-mapping applied to θ, then it can be shown by induction that the cost computed by tree-mapping at node a in θ is greater than or equal to the the cost computed by graph-mapping at the choice node c in μ. An implementation of the function computed by c with cost equal to *cost(c)* can be constructed by the procedure given in step 4. Likewise, tree-mapping constructs a solution with cost equal to *cost(a)*. Since a and c can be the fanins of corresponding primary outputs, graph-mapping produces an implementation for each primary output with cost less than or equal to that produced by applying tree-mapping to any encoded AND2/INV network θ. ■

The algorithm usually runs in linear time in the number of nodes in the mapping graph (steps one, two, and three) plus the size of the mapped network (step four). The **while** loop in `Update` typically runs a small number of times. Loops in step three are uncommon in practice.

The efficiency of graph-mapping compared to exhaustive application of tree-mapping is therefore closely related to the compactness with which a mapping graph encodes a set of AND2/INV networks. In fact, the mapping graph representation is often extremely effective. For example, there are 2.22×10^{93} distinct AND2/INV decompositions of the MCNC91 cht benchmark. However, this entire set can be encoded in a mapping graph with 400 ugates containing a total of 599 AND2 nodes using the procedure described in the next section.

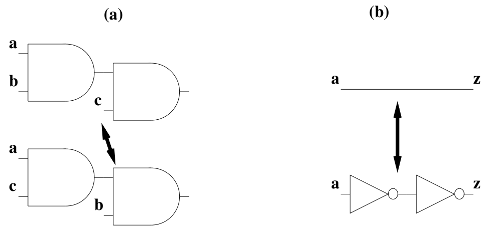

Figure 4: Two transformations of an AND2/INV network: (a) associative transformation (b) inverter transformation.

3.3 Λ-Mapping

For a boolean network η, the set Λ_η is defined as the closure of the set of AND2/INV decompositions of η under the *associative* and *inverter* tranformations. These transformations are depicted in Figure 4. The Λ-mapping procedure encodes the set Λ_η in a mapping graph μ using the procedure described below. Then graph-mapping is applied to μ. By Theorem 3.1 the result is as good or better than obtained by applying tree-mapping to every possible AND2/INV decomposition of η. As a consequence, all decisions taken blindly in phase (2) of the conventional synthesis approach are now taken optimally during technology mapping.

The following three-step procedure is used to build a mapping graph μ encoding Λ_η from boolean network η.

In the first step, η is used to produce a new boolean network $\tilde{\eta}$. Initially, $\tilde{\eta}$ is just a copy of η. Then each internal node n in $\tilde{\eta}$ is replaced by a set of nodes, each designating either an inverter or multi-input AND. One AND is created for each product term in sum-of-products expression f_n. One more AND, with an inverter on the output and every input, is created for the sum. After all nodes are transformed, every pair of consecutive inverters in $\tilde{\eta}$ is deleted. Finally, if one AND drives another, the two are combined to create a single AND with more inputs. (This may require duplication of a node if a single AND drives several.)

In the second step, each node of $\tilde{\eta}$ is translated to a set of nodes in μ. The translation works from inputs to outputs. A particular ugate output, denoted *root(n)*, is associated with each node n in the network $\tilde{\eta}$. Each primary input n in $\tilde{\eta}$ generates a primary input ugate u in μ where *root(n)* is the positive output of u. An inverter n with fanin m does not generate a ugate. However, if *root(m)* is an output of the ugate u, then *root(n)* is the other output of ugate u. A primary output n with fanin m generates a primary output ugate driven by *root(m)*. The most complex case is an AND node n with fanins $m_1, \ldots, m_k$. Let R be the set $root(m_1), \ldots, root(m_l)$. For each non-empty subset S of R, the variable v_S will denote a ugate output. For each single-element subset $S = root(m_i)$ of R, let the variable v_S be $root(m_i)$. For each multiple-element subset S of R, let v_S be the positive output of a newly created AND ugate. This new ugate should contain a positive AND2 with inputs v_T and v_U for each partition of S into non-empty, disjoint subsets T and U. Let *root(v)* be v_R. See Figure 5.

In the third step, μ is reduced as described in Section 3.1.

In practice, it is usually feasible to encode Λ_η in a mapping graph provided no AND in the intermediate network $\tilde{\eta}$ has more than about 10 inputs. This means that Λ-mapping can be applied to most practical examples. Of course, if there is a larger AND in $\tilde{\eta}$, it is still possible to encode numerous decompositions of the node.

Theorem 3.2 *For a general boolean network η, every element of Λ_η is encoded in the mapping graph μ generated by the above construction.*

Proof: Observe that every AND2 a contained in an AND2/INV network in Λ_η is naturally associated with an AND n in $\tilde{\eta}$. The inputs of a are logically equivalent to ANDs of disjoint subsets of fanins of n. Also note that for any internal node b in an AND2/INV network in Λ_η, there is a logically equivalent choice node in μ due to the construction in step two.

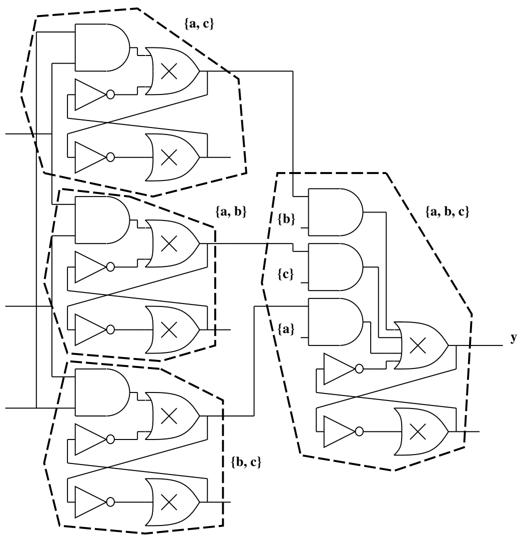

Figure 5: A portion of a mapping-graph which encodes all AND2/INV decompositions of y = a b c is shown. Ugates are circled and the associated subset of inputs is indicated. The ugates producing signals a, b, and c are not shown.

Now choose an AND2/INV network in Λ_η. Working inputs to outputs, it is proven inductively that for each node b in the AND2/INV network, there is a depth-first search from the logically equivalent choice node in μ which generates a tree isomorphic to the transitive fanin of b. It follows that the AND2/INV network is encoded in μ.

The hypothesis is certainly true in the base case where b is a primary input. Suppose b is an inverter. The depth-first search begins properly because every choice node is driven by an inverter (see Figure 2) and the remainder of the transitive fanin can be generated by induction. Suppose b is an AND2 associated with AND n in $\tilde{\eta}$. The inputs of b are logically equivalent to ANDs of disjoint subsets of the fanins of n. The depth-first search begins properly because in step two an AND2 was added to μ for every such combination of disjoint subsets. The remainder of the transitive fanin can be generated by induction. Suppose b is a primary output. The depth-first search begins properly because there is a corresponding primary-output in μ and is completed by induction. ∎

4 Dynamic Logic Decomposition

The previous section presented a technology mapping procedure called Λ-mapping, which takes into account a set of AND2/INV networks encoded in a mapping graph. In this section, we extend the procedure, so that the set of AND2/INV networks is dynamically modified during technology mapping. This process is done by dynamically performing logic decomposition on a mapping graph.

4.1 Overview

We define a new transformation on an AND2/INV network, the vehicle for dynamic logic decomposition. The transformation is based on the *distributive law*, and is referred to as the *distributive transformation*. The distributive transformation is defined as a transformation from $xy + xz$ to $x(y + z)$, which is illustrated in Figure 6, where Figure 6-(a) corresponds to $xy + xz$ and Figure 6-(b) corresponds to $x(y + z)$. We call the structure shown in the rectangle of Figure 6-(a) the *distributed pattern*, or *D-pattern* for short. Similarly, the structure shown in Figure 6-(b) is referred to as the *factored pattern*, or *F-pattern* for short. Note that the functionality of a D-pattern

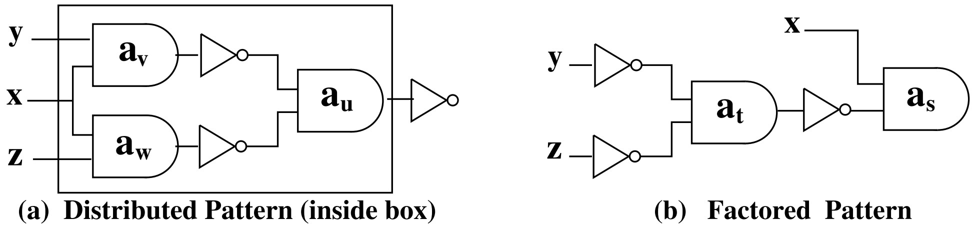

Figure 6: Distributive Transformation

is the complement of that of the corresponding F-pattern.

Let Δ_η denote the closure of Λ_η under the distributive transformation. Recall that Λ_η is the closure of the set of AND2/INV decompositions of a boolean network η under the associative and inverter transformations. As of the previous section, where the scope captured by Λ-mapping was characterized by Λ_η, we show that with dynamic decomposition, we can explore the entire space Δ_η.

4.2 Δ-Mapping

The idea of dynamic decomposition is simple: while traversing a mapping graph during technology mapping, we identify D-patterns, and add the corresponding F-patterns to the mapping graph. We call this technology mapping procedure Δ-mapping. We illustrate the basis of the procedure in this section, and refine it in Sections 4.4 and 4.5.

The Δ-mapping procedure is an iteration of the graph-mapping procedure. Once a mapping graph is constructed for a given boolean network, we apply the graph-mapping procedure while finding D-patterns and adding F-patterns. Once graph-mapping is done, we check if there is a solution which meets user-specified constraints. If so, we terminate Δ-mapping by generating the result. Otherwise, we apply the graph-mapping procedure again, while identifying more D-patterns, since new D-patterns might have been introduced when F-patterns were added in the previous iteration. This process is iterated, until either a desired solution is found, or no new D-pattern is found.

The core of Δ-mapping is a procedure called `Dynamic_decomp`, which is illustrated in Figure 7. This procedure is applied for each AND ugate u of a mapping graph to identify D-patterns. `Dynamic_decomp` is called from `Update`, the procedure defined in Section 3.2 for computing matches in u. More precisely, it is invoked at the beginning of `Update`, before matches are computed for the choice nodes of u. This behavior is controlled by a flag provided as an argument of `Update`, since there is a case where we want to apply `Update` without applying dynamic decomposition. In the rest of the paper, we often say that dynamic decomposition is applied for a ugate u or an AND2 a, by which we mean that `Dynamic_decomp`, or its subprocedure for a single AND2, is applied for u or a respectively.

`Dynamic_decomp` works as follows. For each polarity $p \in \{0, 1\}$, we look at each AND2 a_u of H_u^p, the set of AND2s of u for the polarity p. We check if a D-pattern matches at a_u. If this is the case, we construct the corresponding F-pattern shown in Figure 6-(b), and add it to the mapping graph. Since the functionality of the resulting F-pattern is the complement of that of the D-pattern we identified, the root AND2 of the F-pattern, denoted by a_s in Figure 6-(b), is added to $H_u^{\overline{p}}$, the set of AND2s of u whose polarity is the opposite of p. Recall the structure of an AND ugate shown in Figure 2-(c).

To add the other AND2 of the F-pattern, denoted by a_t in Figure 6-(b), we first create a new AND ugate, which consists of a_t in the positive half. The negative output of the ugate fans out to a_s. The resulting mapping graph is then reduced. We invoke `Update` for the ugate containing a_t without further applying dynamic decomposition.

In each iteration where graph-mapping is applied, we apply dynamic decomposition only once for each AND2 which exists in the mapping graph at the beginning of the iteration. It is not applied on AND2s that are newly

```
procedure Dynamic_decomp(u)
    for each polarity p ∈ {0, 1}
        for each a_u ∈ H_u^p
            if D-pattern is found at a_u
                add F-pattern to the mapping graph
                /* t is a ugate for the second AND2 a_t of F-pattern. */
                Update(t, NO_DECOMP)
```

Figure 7: Dynamic Logic Decomposition (basic flow)

introduced during the iteration.

In practice, Δ-mapping is used to speed up a timing critical region of a mapped network. Since the region is usually small and to maximize the effectiveness of dynamic decomposition, we initially collapse the region to a sum-of-products expression. Thus, the procedure is typically applied on a boolean network with a single node. However, this restriction is not necessary to characterize the space explored by the procedure, as described in the following section.

4.3 The Space Explored by Dynamic Logic Decomposition

4.3.1 Preliminaries

Let D_η denote the set of AND2/INV networks captured by applying the Δ-mapping procedure on a given boolean network η. Specifically, suppose we apply dynamic decomposition maximally on ugates of a mapping graph μ constructed for η, i.e. whenever a D-pattern is found, the corresponding F-pattern is added to μ, and this process is repeated until nothing new is found. D_η is the set of AND2/INV networks encoded in the resulting mapping graph.

The objective in this section is to claim that D_η subsumes the set of AND2/INV networks obtained by an operation usually referred to as *algebraic decomposition* [3]. Algebraic decomposition is a process which is widely used for changing the structure of a boolean network. The result is another boolean network with the same functionality. The claim is that no matter how algebraic decomposition is applied on η, every AND2/INV decomposition of the resulting boolean network is contained in D_η. We will provide a definition of algebraic decomposition in the succeeding section.

We make this claim with help of Δ_η, the closure of Λ_η under the distributive transformation defined in Section 4.1. We first claim that Δ_η is subsumed by D_η.

Lemma 4.1 *Given a boolean network η, $\Delta_\eta \subseteq D_\eta$.*

Proof: Given a set Θ of AND2/INV networks, let $\Delta(\Theta)$ denote the closure of Θ under the distributive transformation. Note that $\Delta_\eta = \Delta(\Lambda_\eta)$. We first note that for a pair of sets, say Θ_1 and Θ_2, of AND2/INV networks, $\Theta_1 \subseteq \Theta_2$ implies $\Delta(\Theta_1) \subseteq \Delta(\Theta_2)$. Let L_η be the set of AND2/INV networks encoded in the mapping graph μ, which is obtained from η by the procedure given in Section 3.3. By Theorem 3.2, L_η subsumes Λ_η. Since Δ-mapping applies the distributive transformation on the set L_η until nothing new is generated, the resulting set of AND2/INV networks D_η is equal to $\Delta(L_\eta)$. Thus, $\Delta_\eta = \Delta(\Lambda_\eta) \subseteq \Delta(L_\eta) = D_\eta$. ∎

Hence, what remains is to show that for an arbitrary boolean network $\tilde{\eta}$ obtained by applying algebraic decomposition on η, the set of AND2/INV decompositions of $\tilde{\eta}$ is contained in Δ_η.

4.3.2 Algebraic Decomposition

We now present a definition of an operation called *algebraic decomposition*.

An *algebraic expression* f is a sum-of-products expression where no product term is logically contained in another. In the sequel, we may regard f as a set of product terms.

Given algebraic expressions f and g, g is said to be an *algebraic divisor* of f if there exists an algebraic expression h such that (1) there is no variable common between g and h, and (2) for all $(p, q) \in g \times h$, $p \cdot q \in f$, where $p \cdot q$ is a product term given by the product of literals over p and q. The expression h is called a quotient of f for g. Note that h is not unique. Given a quotient h, the *remainder* is uniquely defined as the set of product terms of f not in $\{p \cdot q \mid p \in g, q \in h\}$.

We define an operation called the *algebraic division* as follows. It is defined on a pair of nodes n and m of a boolean network, and is denoted by $DIV(n, m)$. The operation first finds a quotient h of f_n for the divisor f_m, if exists, as well as the corresponding remainder r. Since h is not unique, it may choose any quotient. It then replaces f_n by $r + \sum_{q \in h} qy_m$, where qy_m denotes a product term given by multiplying the positive literal of the variable y_m associated with m to a product term q of h. The resulting boolean network has an arc from m to n.

Algebraic decomposition is defined as a three-step operation: (1) find an algebraic divisor g of f_n, (2) create or find a node m such that $f_m = g$, and (3) apply $DIV(n, m)$.

In practice, one may take the complement of g, say $\overline{g}$, and may further divide the expression of n in the resulting network by $\overline{g}$. This extension is discussed in Section 4.3.4.

4.3.3 Algebraic Decomposition and Dynamic Decomposition

Consider an arbitrary boolean network $\tilde{\eta}$ obtained by successively applying algebraic decomposition on a given boolean network η. Let $\Theta_{\tilde{\eta}}$ denote the set of AND2/INV decompositions of $\tilde{\eta}$. We claim $\Theta_{\tilde{\eta}} \subseteq \Delta_\eta$.

We first show that the statement is true if algebraic decomposition is applied only once. This is claimed by the following lemma. The proof is found in Appendix.

Lemma 4.2 *Given a boolean network η, let $\tilde{\eta}$ be a boolean network obtained by applying algebraic decomposition on η. Then an arbitrary AND2/INV decomposition of $\tilde{\eta}$ is obtained from some AND2/INV decomposition of η through the distributive transformation.*

This lemma immediately implies that if a boolean network η is transformed to $\tilde{\eta}$ by successively applying algebraic decomposition, an arbitrary AND2/INV decomposition $\tilde{\theta}$ of $\tilde{\eta}$ can be obtained from some AND2/INV decomposition θ of η through the distributive transformation. Since $\tilde{\theta}$ is arbitrary, and since $\theta \in \Delta_\eta$, it follows that $\Theta_{\tilde{\eta}} \subseteq \Delta_\eta$. Recall that Lemma 4.1 claims that Δ_η is contained in the set D_η of AND2/INV networks considered by the Δ-mapping procedure. Hence, we finally conclude the following.

Theorem 4.1 *Given a boolean network η, let $\tilde{\eta}$ be a boolean network obtained from η by successively applying algebraic decomposition. Then every AND2/INV decomposition of $\tilde{\eta}$ is contained in D_η.*

Therefore, by applying Δ-mapping, one can effectively explore the entire set of AND2/INV networks obtained from η through algebraic decomposition and AND2/INV decomposition. By Theorem 3.1, an optimal result over all such decompositions is obtained. We emphasize the point that the Δ-mapping procedure explores the space directly on a mapping graph during technology mapping, and there is no need to go back and forth between the technology independent phase and the technology dependent phase.

4.3.4 Algebraic Division with a Complemented Expression

In practice, one may generalize the algebraic decomposition defined in Section 4.3.2 so that after the third step, the expression of the node n is further divided by the complement of the divisor g. Note that in an AND2/INV decomposition of the resulting boolean network, the complemented expression $\overline{g}$ is represented as an inverter driven by an AND2/INV decomposition for g. On the other hand, using the distributive transformation, we cannot identify in general that $\overline{g}$ can be realized using an AND2/INV decomposition of g, since we cannot see that g and $\overline{g}$ are logically complement. Therefore, the set of AND2/INV decompositions for a boolean network $\tilde{\eta}$ obtained through such a generalized algebraic decomposition may not be contained in Δ_η. However, it is subsumed by D_η, the set actually captured by the Δ-mapping procedure. This is because a mapping graph is reduced, i.e. there are no two distinct choice nodes that are logically equivalent. In other words, an AND2/INV network for $\overline{g}$ is encoded in the mapping graph as an inverter driven by the choice node whose transitive fanin represents AND2/INV networks for g. Hence, every AND2/INV decomposition of $\tilde{\eta}$ is contained in D_η.

4.4 Factor-Free Ugates

Δ-mapping iteratively applies graph-mapping with dynamic decomposition. It is inefficient if the distributive transformation is applied many times on a ugate at which no new D-pattern can be found. In this section, we discuss a condition under which such an inefficient computation can be avoided.

Intuitively, dynamic decomposition can be skipped for an AND2 a if all D-patterns have been identified in the transitive fanin of a. An AND2 of a ugate is said to be *factor-free* if there is no *untransformed* D-pattern in its transitive fanin, where a D-pattern is said to be *transformed* if the corresponding F-pattern has been added to the mapping graph, and *untransformed* otherwise. A ugate is said to be *factor-free* if all of its AND2s are factor-free.

The Δ-mapping procedure identifies this property locally. It marks an AND2 a as FF (factor-free) if dynamic decomposition has been applied for a and both of its fanin ugates are marked as FF. A ugate is marked as FF if all of its AND2s are marked as FF. Initially, we mark ugates for primary inputs as FF. When `Dynamic_decomp` is invoked, we skip the procedure for an AND2 if it is marked as FF.

Now, the question is whether this marking strategy is consistent with the definition of factor-free ugates, i.e. if an AND2 is marked as FF, it is factor-free in the rest of the procedure. First, it is claimed that the statement is true in the case where Δ-mapping is applied in practice, as stated below.

Theorem 4.2 *Suppose that a boolean network η consists of a single node and the associated expression is prime and irredundant. Then an AND2 marked as FF is factor-free in the rest of the Δ-mapping procedure.*

Proof: Due to the lack of space, we briefly sketch the proof. Recall that Δ-mapping applies the graph-mapping procedure iteratively, where `Dynamic_decomp` is invoked once for each ugate that exists in the mapping graph at the beginning of the iteration. In the first iteration, it is claimed that if dynamic decomposition has been applied for an AND2 a, no new AND2 is added to any ugate in the transitive fanin of a in the rest of the first iteration. Similarly, on and after the second iteration, it is claimed that a new AND2 is added to a ugate u only if dynamic decomposition is being applied for u. This implies that once an AND2 a is marked as FF, no untransformed D-pattern is created in the transitive fanin of a in the rest of the Δ-mapping procedure. Thus, it is claimed by induction that when a is marked as FF, a is factor-free, and it remains so in the rest of the procedure. ■

Therefore, Δ-mapping with this strategy for factor-free ugates skips dynamic decomposition for some ugates without affecting the quality of the results in practice. For a more general case, however, the statement may not hold. In fact, an example is known, where a new AND2 is created and a untransformed D-pattern is introduced in the transitive fanin of an AND2 a after a has been marked as FF. In this case, we might not be able to explore the entire space D_η which can be done by the original procedure given in Figure 7.

4.5 Dynamic Decomposition with Deletion

So far, the procedure only introduces new AND2/INV networks, and never deletes existing ones. However, some of them might not lead to good results, and thus it is not worth having them around. The Δ-mapping procedure can be modified to detect such a situation, to some extent, and delete those AND2/INV networks.

The basic step of the deletion process is to delete an AND2 from the ugate it belongs to. Intuitively, it is worth having an AND2 a if there is a possibility that a is included in a mapping identified at some node of the mapping graph at which no mapping with at least equally good cost can be found without a. We want to detect this condition while processing the ugate u that a belongs to. This is difficult in general, since such a mapping might be found after u has been processed. Therefore, we modify Δ-mapping as described in the succeeding sections.

4.5.1 Partial Matches and a Subset of AND2s

The first modification is made on `Update`, the procedure shown in Figure 3. The original procedure looks at the transitive fanin of a node for finding matches. We modify the procedure, so that matches can be found only by looking at the immediate fanins.

This modification is done by using a notion called *partial matches*. Given either an inverter or an AND2, say b, a partial match at b is defined as a bijective function ϕ_p from a set of nodes of a pattern of a library gate to a subset of AND2s and inverters in the transitive fanin of b, which satisfies four properties similar to the ones given in the definition of a match in Section 3.2. In fact, the third and fourth properties are exactly same. Unlike the property (1) defined for a match, which states that the domain of the match is the set of all internal nodes of the pattern, the domain of the partial match ϕ_p is the set of all internal nodes of the transitive fanin of *any* internal node b_p of the pattern. Also, instead of the property (2) defined for a match, we need the property that $\phi_p(b_p) = b$. In other words, a match is a special type of a partial match such that b_p is the fanin of the pattern's primary output. If a partial match ϕ_p is indeed a match, we say that ϕ_p is *complete*. The domain of ϕ_p defines a subgraph of the corresponding pattern. The subgraph is an AND2/INV network with the root b_p, which we denote by T_{ϕ_p}. We say two partial matches, say ϕ_1 and ϕ_2, are *comparable* if T_{ϕ_1} and T_{ϕ_2} are isomorphic. The fanins of a partial match are defined in the same way as for a match.

Given a partial match ϕ_p, we define the cost at each leaf (node with zero in-degree) of T_{ϕ_p} and at the root. The cost of a leaf l of T_{ϕ_p} is the cost of the fanin choice node driving $\phi_p(l)$, where the cost of a choice node is defined in a moment. The cost of the root of T_{ϕ_p} is defined only if ϕ_p is complete, and is equal to the one given by the cost function used in the graph-mapping procedure, which is determined by the costs of the leaves of T_{ϕ_p} and the library gate for the pattern defined by ϕ_p. For comparable partial matches ϕ_1 and ϕ_2, we say that the cost of ϕ_1 is no worse than the cost of ϕ_2, denoted by $\phi_1 \leq \phi_2$, if the cost of each leaf and the root (if defined) of T_{ϕ_1} is no worse than the corresponding cost of T_{ϕ_2}.

Suppose the procedure `Update` is applied for an AND ugate u in graph-mapping. It first computes the set of partial matches at each AND2 a using the partial matches associated with the fanin choice nodes of a. This is done by a technique similar to the one given in [9]. Once all the AND2s have been processed, `Update` processes the choice nodes and inverters of u. Partial matches for an inverter are generated in the same way as for AND2s. For a choice node, we first take the union of the partial matches generated at the fanins of the choice node. We then delete some partial matches from the union. A partial match ϕ_1 is deleted if there exists another partial match ϕ_2 in the union such that ϕ_1 and ϕ_2 are comparable and $\phi_2 \leq \phi_1$. We associate the resulting set with the choice node. `Update` iterates this process for the choice nodes and inverters, until the sets of partial matches at both choice nodes become stable. The cost of a choice node is the least root-cost of T_ϕ over all the complete matches ϕ associated with the choice node.

Recall that we use partial matches so that we can delete some AND2s from a ugate. The key observation is that not all partial matches generated for an AND2 are retained. Namely, a partial match is deleted if there is another partial match, which is comparable and at least as good. Intuitively, we see that an AND2 could be deleted if none of its partial matches is retained.

More specifically, let $\pi(a)$ denote the set of partial matches generated at an AND2 a. For the set A of AND2s contained in a ugate u, let $\pi(A)$ be the union of $\pi(a)$ over all $a \in A$. The set of partial matches that we want to retain is the set Π such that (1) $\Pi \subseteq \pi(A)$, (2) for all $\phi_1 \in \pi(A)$, there exists $\phi_2 \in \Pi$ such that $\phi_2 \leq \phi_1$, and (3) no two distinct partial matches of Π are comparable. Let A' be a subset of AND2s of A with the property that $\Pi \subseteq \pi(A')$. In other words, all the partial matches that we want to retain can be generated from AND2s of A'. The AND2s that are not in A' are candidates for deletion. The problem of finding such an A' with the minimum cardinality is called *minimum cover*, whose decision version is known to be NP-complete [10]. Thus `Update` computes such an A' using a heuristic, once all the AND2s have been processed.

In order to identify which AND2s that are not in A' can be actually deleted, another modification is necessary. This issue is discussed in the following section.

4.5.2 Distribution-Free Ugates

Consider an AND2 a which is not in the set A' defined in the previous section. We cannot immediately conclude that a can be deleted. In fact, there are two cases where a should be retained. One case is that a new partial match could be found at a in the future. The other case is that a untransformed D-pattern containing a could be found in the future. If neither happens, a can be deleted. Since no new partial match can be obtained using a, and since the partial matches generated for a are covered by A', the quality of the result is not affected by deleting a.

We detect the first case above by using the flag FF (factor-free) defined in the previous section. Since the first case does not arise as long as a is factor-free, we check if a is marked as FF[3]. The second case above is detected by using another flag called DF (distribution-free). An AND2 a of a ugate u is said to be *distribution-free* if for each AND2 $\hat{a}$ which references u as a fanin, $\hat{a}$ is distribution-free and no untransformed D-pattern matches at $\hat{a}$. If a is distribution-free, a will not be contained in a D-pattern when dynamic decomposition is applied the next time. Since this is true for all the fanouts of a, no new D-pattern will be introduced in its transitive fanout in the rest of the procedure. In the modified Δ-mapping procedure, we traverse a mapping graph between consecutive iterations of the graph-mapping procedure, and mark an AND2 a of a ugate u as DF if for each fanout $\hat{a}$ of u, $\hat{a}$ is marked as DF and dynamic decomposition was applied on $\hat{a}$ in the previous iteration. We delete a from u if a is not in A' and it is marked as FF and DF. If a is deleted, the fanins of a might have no fanout. We delete the fanins as well if this is the case.

5 Experimental Results

The proposed procedure has been implemented in a logic synthesis system called SynFul. SynFul is developed on top of SIS [15], and inherits technology independent transformations from SIS. Operations related to technology mapping have been newly implemented.

SynFul uses a "minimum area under delay constraints" cost notion where each library gate has a fixed delay. This delay model is not only computationally convenient, but also highly accurate in a full-custom methodology [14].

SynFul shares logic using a heuristic: once a decision is taken to implement a particular function, subsequent uses of that function require little additional area. Suppose a plausible common factor is identified either during technology independent processing or from an initial mapping. The ugate output logically equivalent to the common factor is marked. During subsequent calls to graph-mapping, the area component of the cost at a marked output

[3] As stated in Section 4.4, some AND2/INV networks might not be examined in the general case. Note that this does not happen in the case where Δ-mapping is applied in practice as shown in Theorem 4.2.

is set to zero in step three of the procedure. Implementations using the common factor become appropriately less costly and more likely. Due to the cost notion used in SynFul, common factors are used only when timing constraints permit.

SynFul manipulates an extended mapping graph which encodes a set of AND2/INV networks containing latches. The graph-mapping procedure is adapted to produce mapped networks with latches. Furthermore, the dynamic decomposition procedure is extended so that a distributed pattern can be found across latches. If AND2/INV networks which differ only in latch placement are encoded, the graph-mapping algorithm effectively retimes latches [12]. Given an acyclic AND2/INV network containing latches, an mapping graph can be constructed encoding all possible retimings. Relative to encoding a single latch placement, this increases the number of nodes at most in proportion to the maximum number of latches on any primary input to primary output path.

We conducted experiments on MCNC91 benchmark examples of combinational circuits, and compared the results to SIS-1.2. The library used is the one modeled for a commercial design project in progress at the authors' affiliation. We specified delay characteristics so that both systems computed the cost in exactly the same way. We set the arrival times of the primary inputs and the required times of the primary outputs all to zero, so that the fastest possible implementation can be obtained.

The synthesis procedure used for each system is as follows. For SIS-1.2, we used a sequence of operations aimed for timing-oriented synthesis. This includes, in addition to ordinary technology independent operations, a procedure for reducing the depth of a circuit, restructuring timing-critical regions [16], and a timing-driven technology mapping [20]. The restructuring procedure was applied on a mapped network as well. We consulted SIS developers for detailed usage of these procedures to derive the best performance of the system [17].

In SynFul, we first applied the Λ-mapping procedure, where for each example, four different sets of technology independent transformations were first applied on the initial boolean network. AND2/INV decompositions for each of the resulting boolean networks were encoded into a single mapping graph μ. The rationale is that a variety of decompositions can be encoded in a mapping graph in this way. The graph-mapping procedure was then applied on μ. Once a mapped network was obtained, we further applied the Δ-mapping procedure on a timing-critical region. Such a region was identified using a heuristic suggested in [16].

The results are shown in Table 1. For **SIS-1.2** and **Λ-mapping**, **Delay** is the arrival time of the most critical primary output. The most critical output in the network generated by **Λ-mapping** was resynthesized by Δ-mapping, and the resulting arrival time is shown under **Delay** of **Δ-mapping**. The delay unit is set to the minimum delay from an input pin to the output pin over all the gates specified in the library. **Area** shows the area of the entire network obtained by each procedure, where the area unit is the smallest area among the library gates. **Time** is the CPU time in seconds measured on VAXstation 6000. The CPU time for Λ-mapping includes the time for technology independent transformations, construction of a mapping graph, and the graph-mapping procedure.

The results show that SynFul finds faster implementations than SIS-1.2 over all the examples tried. The results are remarkably better for some examples, such as x2, ttt2, and comp. It is interesting to note that SynFul often finds an implementation that is faster and smaller. See cm162a, pcle, x2, i1, for example. Even though Λ-mapping already outperforms SIS-1.2, further improvements are usually possible by using the Δ-mapping procedure. In fact, even faster implementations were found for all the examples except one. Note that Δ-mapping effectively tries all the algebraic decompositions using the real cost function in technology mapping. Therefore, we consider the procedure especially effective when the arrival times of input signals vary and a decomposition for the fastest implementation cannot be easily identified by technology independent transformations. The CPU time is reasonable.

Name	SIS-1.2		SynFul					
			Λ-mapping			Δ-mapping		
	Delay	Area	Delay	Area	Time	Delay	Area	Time
cc	12.0	167.7	9.5	192.0	16.5	8.0	193.3	3.3
cm82a	13.2	86.0	10.5	58.3	8.9	7.0	78.7	8.8
pcle	15.0	266.7	12.0	156.7	35.7	10.5	176.7	11.5
pm1	13.5	135.0	9.8	135.0	47.1	9.2	131.3	14.0
x2	17.7	151.0	9.8	109.7	58.0	9.8	109.7	24.1
cm163a	15.5	130.7	12.0	111.3	16.5	9.7	121.0	39.9
comp	32.5	401.3	26.0	502.7	193.4	21.2	454.3	43.5
cu	16.0	145.0	11.8	147.3	27.7	9.8	149.0	50.3
i1	12.7	183.0	11.5	132.3	24.2	9.8	111.0	62.3
cm151a	15.0	77.0	10.5	66.3	14.5	9.0	88.3	103.7
cht	9.2	459.7	9.2	320.7	10.7	8.2	323.7	2.8
ttt2	21.5	530.7	15.3	576.0	112.4	12.7	606.7	134.4
cm162a	12.7	164.7	12.0	111.3	21.4	9.7	122.7	154.8
term1	25.8	612.3	19.5	658.3	314.9	15.7	602.0	266.9

Table 1: Experimental Results (Timing-Driven Technology Mapping)

6 Conclusion

Conventional approaches distinguish three phases in logic synthesis: (1) technology independent processing, (2) AND2/INV decomposition, and (3) technolgy mapping. Although decisions made in the first two phases critically affect results, accurate delay, area, and loading information is not available until the third phase. Therefore, these critical decisions must be taken almost arbitrarily, producing inferior results.

Our approach allows a key part of phase(1) and all of phase (2) to be combined with mapping. The cornerstones of our approach are the mapping graph data structure and the graph-mapping algorithm. A mapping graph compactly encodes a set of AND2/INV networks. In effect, the graph-mapping algorithm applies tree-mapping to every encoded AND2/INV network and retains the best result. However, in run time, graph mapping is typically exponentially faster than this naive approach.

Built atop these pieces, the Λ-mapping procedure combines the AND2/INV decomposition phase into mapping. For boolean network η, the Λ-mapping procedure encodes Λ_η in a mapping graph and applies graph-mapping. Since Λ_η contains all AND2/INV decompositions of η, results are as good as can be obtained by applying tree-mapping to every AND2/INV decomposition of η.

The Δ-mapping procedure further combines algebraic decomposition into mapping. Every AND2/INV decomposition of an algebraic decomposition of boolean network η is contained in the set Δ_η. The Δ-mapping procedure initially encodes Λ_η in a mapping graph. Then an extended version of the graph-mapping procedure is applied which dynamically performs the distributive transformation to explore all of Δ_η. Some provably inferior AND2/INV networks may be discarded or never generated by the procedure to increase efficiency. The result is optimal over all AND2/INV decompositions of all algebraic decompositions of η.

Our experiments suggest that Λ-mapping and Δ-mapping improve results substantially over the traditional approach in which technology independent processing, AND2/INV decomposition, and mapping are separated.

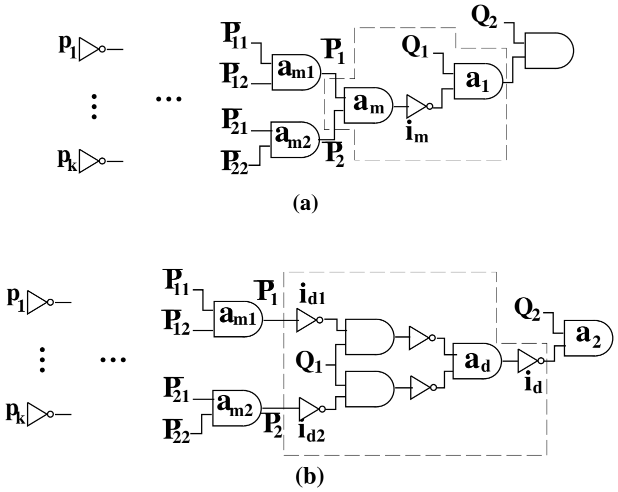

Figure 8: Reverse Transformation from $\tilde{\theta}$ to θ

Appendix

Lemma 4.2 *Given a boolean network η, let $\tilde{\eta}$ be a boolean network obtained by applying algebraic decomposition on η. Then an arbitrary AND2/INV decomposition of $\tilde{\eta}$ is obtained from some AND2/INV decomposition of η through the distributive transformation.*

Proof: Let n be the node of η on which algebraic decomposition was applied, and m be a node such that f_m is the algebraic divisor used in this operation. Namely, $DIV(n, m)$ was applied.

Recall that an AND2/INV decomposition of η is obtained by replacing each node of η by an AND2/INV decomposition of the node. Since the difference between η and $\tilde{\eta}$ arises only at the nodes n and m, the claim follows if we show that an arbitrary AND2/INV decomposition defined for the two nodes n and m in $\tilde{\eta}$ can be obtained by transforming some AND2/INV decomposition of n in η using the distributive transformation. To clarify the discussion, we denote in the sequel the nodes of $\tilde{\eta}$ corresponding to n and m by $\tilde{n}$ and $\tilde{m}$ respectively.

Let $\tilde{\theta}_n$ and $\tilde{\theta}_m$ be arbitrary AND2/INV decompositions for $\tilde{n}$ and $\tilde{m}$, respectively. Let $\tilde{\theta}$ denote the canonical AND2/INV network obtained by connecting $\tilde{\theta}_n$ and $\tilde{\theta}_m$. We show that $\tilde{\theta}$ can be transformed to an AND2/INV decomposition of n in η by applying the *reverse distributive transformation*. The reverse distributive transformation is the one which applies the distributive transformation in the other way around. Namely, it transforms an F-pattern to a D-pattern (the complement of D-pattern to be precise) in Figure 6, or $x(\overline{\overline{y}\,\overline{z}}) = x(y+z)$ to $\overline{(\overline{(xy)}\,\overline{(xz)})} = xy + xz$. Note that as illustrated in Figure 6, by adjusting the polarity of a fanin appropriately, an F-pattern can be identified whenever there are two consecutive AND2s with an inverter in-between. Note also that if one decomposition is obtained from another by the reverse distributive transformation, the latter is obtained from the former by the distributive transformation.

Let $p_1, \ldots, p_k$ denote the product terms of f_m, and $q_1, \ldots, q_l$ be the product terms of the quotient h used in this particular $DIV(n, m)$ operation. The set of product terms of f_n is given by the union of $q_j \cdot p_i$ over h and f_m, plus the product terms of the remainder r. The expression of $\tilde{n}$ is $q_1 y_m + \ldots + q_l y_m + r$. The expressions of m and $\tilde{m}$ are identical.

$\tilde{\theta}_m$ consists of a set of subtrees. First, a subtree is defined for each product term of f_m. Second, another subtree is defined for the union of the product terms of f_m. A leaf of the subtree for the union is connected to the root of the subtree of a product term through an inverter. The structure of the subtree for the union is as follows (illustrated as a part of Figure 8-(a)). The root is an inverter, say i_m, whose fanin is an AND2, say a_m. The fanins of a_m are also AND2s, say a_{m1} and a_{m2}. For each AND2, we can associate a subset of product terms of f_m with the property that the functionality of the AND2 is equal to the complement of the union of the product terms of the subset. For example, associated with a_m is the set P of all the product terms of f_m. The function at the output of a_m is equal to the complement of P, denoted by $\overline{P}$. For a_{m1} and a_{m2}, the set P can be partitioned into two non-empty disjoint subsets P_1 and P_2 such that the functions at the outputs of a_{m1} and a_{m2} are equal to $\overline{P_1}$ and $\overline{P_2}$, respectively. Similarly, the fanins of a_{m1} are also AND2s, which again correspond to non-empty subsets P_{11} and P_{12} of P_1 such that P_{11} and P_{12} are disjoint and the union of the two is P_1. As with a_{m1}, the fanins of a_{m2} correspond to P_{21} and P_{22}, which defines a disjoint partition of P_2. This structure continues, until an inverter is encountered at each fanin, which is when the corresponding subset consists of a single product term of f_m.

Now consider the subtree of $\tilde{\theta}_n$ which corresponds to a given product term $q_j y_m$ of the expression for $\tilde{n}$. Let a_1 be the AND2 whose fanin is y_m. The other fanin of a_1 corresponds to a product term made of a subset of literals of q_j, which we denote by Q_1. In $\tilde{\theta}$, y_m is connected to the root of $\tilde{\theta}_m$, i.e. the inverter i_m, as shown in Figure 8-(a). Therefore, an F-pattern is formed by a_1, i_m, and a_m, which is indicated by the dotted line in Figure 8-(a). This pattern corresponds to $Q_1 \overline{(\overline{P_1}\,\overline{P_2})} = Q_1(P_1 + P_2)$. Applying the reverse distributive transformation, we replace the F-pattern by a D-pattern with appropriate inverters added at the leaves and the root, as shown in Figure 8-(b). The resulting pattern corresponds to $\overline{(\overline{(Q_1\,P_1)}\,\overline{(Q_1\,P_2)})} = Q_1P_1 + Q_1P_2$. Let us call the three inverters at the root and the leaves of this D-pattern i_d, i_{d1}, and i_{d2} respectively, while the fanin of i_d is denoted by a_d as shown in the figure.

This D-pattern introduces new F-patterns. We are interested in three of them, identified at the boundary of the D-pattern. The first one is found at the fanout of i_d. This fanout is an AND2 of $\tilde{\theta}_n$, say a_2. The other fanin of a_2 corresponds to a product term made of a non-empty subset of literals of q_j, say Q_2. Q_2 does not intersect with Q_1. As shown in Figure 8-(b), an F-pattern is formed by a_2, i_d, and a_d, which corresponds to $Q_2(Q_1P_1 + Q_1P_2)$. The other two F-patterns are found at the fanouts of i_{d1} and i_{d2} respectively. They correspond to $Q_1\overline{(\overline{P_{11}}\,\overline{P_{12}})} = Q_1(P_{11} + P_{12}) = Q_1P_1$ and $Q_1(P_{21} + P_{22}) = Q_1P_2$, respectively. Replacing these F-patterns by D-patterns, we obtain a subtree for $Q_2(Q_1P_1) + Q_2(Q_1P_2) = Q_2(Q_1P_{11} + Q_1P_{12}) + Q_2(Q_1P_{21} + Q_1P_{22})$. The newly created D-patterns again introduce new F-patterns.

Continuing this process, we obtain a subtree corresponding to $Q_N \cdots Q_1p_1 + \ldots + Q_N \cdots Q_1p_k$, where $\{Q_1, \ldots, Q_N\}$ defines a disjoint partition of the literals of q_j (each corresponds to a fanin of an AND2 in the transitive fanout of a_1 in the subtree for the product term q_jy_m). Thus, the subtree for q_jy_m is replaced by a tree for $q_j \cdot p_1 + \ldots + q_j \cdot p_k$. Hence, $\tilde{\theta}$ is transformed to a tree for $\sum_{(q_j,p_i)\in h\times f_m} q_j \cdot p_i + r$ by the reverse distributive transformation. The resulting tree θ is an AND2/INV decomposition for the node n in η. ■

References

[1] C. Berman, J. Carter, and A. Day. The Fanout Problem: from theory to practice. In *Conference on Advanced Research in VLSI*, 1989.

[2] R. Brayton, G. Hachtel, and A. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. *Proceedings of the IEEE*, Vol. 78(No. 2), February 1990.

[3] R. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *ISCAS*, 1982.

[4] J. Darringer, W. Joyner, L. Berman, and L. Trevillyan. Logic Synthesis through Local Transformations. *IBM J. Res. Develop.*, pages 272–280, July 1981.

[5] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Technology Mapping in MIS. In *ICCAD*, 1987.

[6] J. Fishburn. A Depth-Decreasing Heuristic for Combinational Logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between. In *DAC*, 1990.

[7] J. Fishburn. LATTIS: An Interative Speedup Heuristic for Mapped Logic. In *DAC*, 1992.

[8] D. Gregory, K. Barlett, A. de Geus, and G. Hachtel. Socrates: A System for Automatically Synthesizing and Optimizing Combinational Logic. In *DAC*, 1986.

[9] A. Hoffman and M. O'Donnell. Pattern Matching in Trees. *Journal ACM*, 29(1), January 1982.

[10] R. Karp. Reducibility among Combinatorial Problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, 1972.

[11] K. Keutzer. DAGON: Technology Binding and Local Optimization by DAG Matching. In *DAC*, 1987.

[12] C. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In R. Bryant, editor, *3rd Caltech Conference on VLSI*, 1983.

[13] P. McGeer, R. Brayton, and A. Sangiovanni-Vincentelli. Performance Enhancement through the Generalized Bypass Transform. In *ICCAD*, 1991.

[14] *same authors*. A Delay Model for Logic Synthesis of Continuously-Sized Networks. submitted to *ICCAD*, 1995.

[15] E. Sentovich, K. Singh, C. Moon, H. Savoj, R. Brayton, and A. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *ICCD*, 1992.

[16] K. Singh. *Performance Optimization of Digital Circuits*. PhD thesis, U.C. Berkeley, 1992.

[17] K. Singh. *private communication*, 1995.

[18] K. Singh and A. Sangiovanni-Vincentelli. A Heuristic Algorithm for the Fanout Problem . In *DAC*, 1990.

[19] K. Singh, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Timing Optimization of Combinational Logic. In *ICCAD*, 1988.

[20] H. Touati. *Performance-Oriented Technology Mapping*. PhD thesis, U.C. Berkeley, November 1990.

[21] K. Yoshikawa, H. Ichiryu, H. Tanishita, S. Suzuki, N. Nomizu, and A. Kondo. Timing Optimization on Mapped Circuits. In *DAC*, 1991.