Optimum Functional Decomposition Using Encoding *

Rajeev Murgai Robert K. Brayton Alberto Sangiovanni-Vincentelli

Department of EECS, University of California, Berkeley, CA-94720

Abstract

In this paper, we revisit the classical problem of functional decomposition [1, 2] that arises so often in logic synthesis. One basic problem that has remained largely unaddressed to the best of our knowledge is that of decomposing a function such that the resulting sub-functions are *simple*, i.e., have small number of cubes or literals. In this paper, we show how to solve this problem optimally. We show that the problem is intimately related to the encoding problem, which is also of fundamental importance in sequential synthesis, especially state-machine synthesis. We formulate the optimum decomposition problem using encoding. In general, an input-output encoding formulation has to be employed. However, for field-programmable gate array architectures that use look-up tables, the input encoding formulation suffices, provided we use minimum-length codes. The last condition is really not a constraint, since each extra code bit means that an extra table has to be used (and that could be expensive). The unused codes are used as don't cares for simplifying the sub-functions. We compare the original implementation of functional decomposition, which ignores the encoding problem, with the new version that uses encoding while doing decomposition. We obtain an average improvement of over 20% on a set of standard benchmarks for look-up table architectures.

1 Introduction

Decomposition is a fundamental problem in logic synthesis. Its goal is to break a function into simpler functions. The first systematic study on decomposition was done by Ashenhurst [1]. He characterized the existence of a simple disjoint decomposition of a function. While being seminal, this work could not be used for functions with more than 10-15 inputs, since it required the construction of a *decomposition chart*, a modified form of the truth table for a function. Few years later, Roth and Karp proposed a technique [2] that does not require building a decomposition chart; instead, it uses a sum of products representation, which is, in general, more compact than a truth table. They, in fact, extended Ashenhurst's work by characterizing non-simple (or general) decompositions and used this characterization to determine the minimum-cost Boolean network using a library of primitive gates, each with some cost. Both of these studies did not address the problem of decomposing the function such that the resulting sub-functions are *simple*, i.e., have small number of cubes or literals. It is important that they be simple, otherwise we may lose the effect of optimizations performed thus far.

In [6], Roth and Karp decomposition was used to generate from an arbitrary network a network with fanin-constraint. The fanin-constraint restricted each function of the network to have a maximum of m inputs, where m is a constant. Such a function with at most m inputs is called an **m-feasible function**. This has direct application to look-up table (LUT) based field-programmable gate arrays where each basic block is an m-input LUT, which can implement any function of up to m inputs. This work ignored the issue of simplicity of the sub-functions. Recently, Lai *et al.* [7] used BDDs to implement functional decomposition, but they also ignored the simplicity issue.

In this work, we introduce a straightforward method of doing decomposition such that the resulting sub-functions are simple. It was previously known that an encoding step is needed to solve the problem of functional decomposition. The most popular approach was to encode *equivalence classes* (for a completely specified function)[1] generated during the decomposition [6, 7]. We show that this is not the most general formulation. Our solution is based on performing an encoding step on a certain set of input minterms. We show that the encoding formulation needed to obtain simple sub-functions is that of input-output encoding. However, for LUT architectures, the problem reduces to that of input encoding, which is easier to solve.

The paper is organized as follows. Section 2 briefly explains the encoding problem, and Section 3 describes the classical functional decomposition technique. The relationship between the two is drawn in Section 4. How the problem simplifies for LUT architectures is part of Section 5. Results on a set of benchmark examples are presented in Section 6.

2 The Encoding Problem

Many descriptions of the logic systems include variables that, instead of being 0 or 1, take values from a finite set. For example, states of a controller are initially denoted *symbolically* as $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$. Assume that the controller is in a state S_1 when it fetches the instruction "ADD R1 R2" from the memory, and then moves to a state S_2. To execute the instruction, it has to fetch the two operands from the registers R1 and R2, send a control signal to the adder to compute the sum, and enable the load signal of R1 to store the result in R1. In other words, the controller takes the present state (S_1) and external inputs (the instruction ADD and the names of the registers R1 and R2), and generates control signals (READ signal to R1 and R2, transferring their contents on the bus(ses), ADD signal to the adder, and finally LOAD signal to R1) and computes the next state (S_2). To obtain an implementation of the controller, the states need to be assigned binary codes, since a signal in a digital circuit can only take values 0 and 1. The size of the controller depends strongly on the codes assigned to the states. This gives rise to the problem of assigning binary codes to the states of the controller such that the final gate implementation after encoding and a subsequent optimization is small. It is called the **state-encoding** (or **state-assignment**) problem. Note that it entails encoding of both symbolic inputs (present state variables) and symbolic outputs (next state variables). In other words, it

*This work is supported in part by DARPA under contract number J-FBI-90-073.

[1] more generally, compatibility classes for an incompletely specified function

is an **input-output encoding** problem. The optimization after encoding may be two-level if we are interested in a two-level implementation, or multi-level, otherwise. Correspondingly, there are state-assignment techniques for two-level [10, 8, 14, 15] and for multi-level implementations [16].

Before proceeding any further, we define the concept of a multi-valued function.

Definition 2.1 *A* **multi-valued function** *with n inputs is a mapping $\mathcal{F}: P_1 \times P_2 \times \cdots \times P_n \rightarrow B$, where $P_i = \{0,1,\ldots,p_i-1\}$, p_i being the number of values that i^{th} (multi-valued) variable may take on.*

An example of a multi-valued variable is $\mathcal{S}$, the set of states of a controller. Analogous to the Boolean case, we can define the notion of a multi-valued product term and cover. Then, as in the Boolean two-level case, we have the problem of determining a minimum-cost cover of a multi-valued function. This problem is referred to as **multi-valued minimization problem**.

A problem that is simpler than state-encoding is the one where just the inputs are symbolic. For example, assigning op-codes to the instructions of a processor so that the decoding logic is small, falls in this domain. This is known as the **input encoding** problem. If the objective is to minimize the number of product terms in a two-level implementation, the algorithm first given by De Micheli *et al.* [8] can be used. It views encoding as a two phase process. In the first phase, a multi-valued minimized representation is obtained, along with a set of constraints on the codes of the values of the symbolic variables. In the second, an encoding that satisfies the constraints is determined. If satisfied, the constraints are guaranteed to produce an encoded binary representation of the same cardinality as the multiple-valued minimized representation. Details of the two phases are:

1. *Constraint generation*: The symbolic description is translated into a multi-valued description using positional cube notation. For example, let $\mathcal{S}$ be a symbolic input variable that takes values in the set $\{S_1, S_2, \ldots, S_k\}$. Let x be a binary input, and y the only (binary) output. In positional cube notation (also called 1-hot notation), a column is introduced for each S_i. A possible *behavior* of the system is: if $\mathcal{S}$ takes value S_1 or S_2, and x is 1, then y is 1. This behavior can be written as:

x	S_1	S_2	S_3	...	S_{k-1}	S_k	y
1	1	0	0	...	0	0	1
1	0	1	0	...	0	0	1

   A multi-valued logic minimization is applied on the resulting multi-valued description so that the number of product terms is minimized. The effect of multi-valued logic minimization is to group together symbols that are mapped by some input to the same output. The number of product terms is the same as the minimum number of product terms in any final implementation, provided that the symbols in each product term in this minimized cover are assigned to one *face* (or *subcube*) of a binary cube, and no other symbol is on that face. These constraints are called the **face** or **input constraints**. For example, for the behavior just described,

x	S_1	S_2	S_3	...	S_{k-1}	S_k	y
1	1	1	0	...	0	0	1

   is a product term in the minimum cover. This corresponds to a face constraint that says there should be a face with only S_1 and S_2. This face constraint can also be written as a set of **dichotomies** [14]: $(S_1S_2; S_3), \ldots, (S_1S_2; S_i), \ldots, (S_1S_2; S_k)$, which says that an encoding bit b_i must distinguish S_1 and S_2 from S_i for $3 \leq i \leq k$.

   Also, each symbol should be assigned a different code. These are known as the **uniqueness constraints**, and are handled by adding extra dichotomies. For example, to ensure that the code of S_1 is distinct from other symbols, dichotomies $(S_1; S_2), (S_1; S_3), \ldots, (S_1; S_k)$ are added.

2. *Constraint satisfaction*: An encoding is determined that satisfies all the face and uniqueness constraints. De Micheli *et al.* proposed a satisfaction method based on the constraint matrix (which relates the face constraints to the symbolic values). Yang and Ciesielski [14] proposed an alternate scheme based on dichotomies and graph coloring for solving the constraints. It was later improved by Saldanha *et al.* [11].

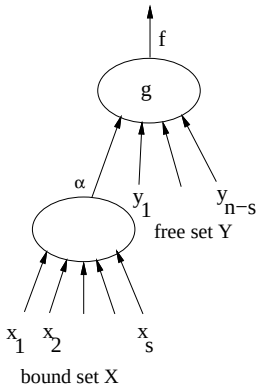

Figure 1: A simple disjoint decomposition

3 Classical Decomposition

We present briefly the classical decomposition theory due to Ashenhurst [1] and Roth & Karp [2]. Ashenhurst [1] gave necessary and sufficient condition for the existence of a simple disjoint decomposition of a completely specified function f of n variables. Given a partition of inputs of f, $X = \{x_1, x_2, \ldots, x_s\}$, $Y = \{y_1, \ldots, y_{n-s}\}$, $X \cap Y = \phi$, a **simple disjoint decomposition** of f is of the form:

$$f(x_1, x_2, \ldots, x_s, y_1, \ldots, y_{n-s}) = g(\alpha(x_1, x_2, \ldots, x_s), y_1, \ldots, y_{n-s}) \quad (1)$$

where α is a single function. In general, α could be a vector of functions, in which case the decomposition is **non-simple** (or **general**).

(1) can also be written as

$$f(X, Y) = g(\alpha(X), Y) \quad (2)$$

The representation (2) is called a **decomposition** of f; g is called the **image** of the decomposition. The set $X = \{x_1, x_2, \ldots, x_s\}$ is called the **bound set** and $Y = \{y_1, \ldots, y_{n-s}\}$ the **free set** (Figure 1). The necessary and sufficient condition for the existence of such a decomposition was given in terms of the **decomposition chart** $D(X|Y)$ for f for the partition $X|Y$ (also written $\frac{X}{Y}$ or (X,Y)). A decomposition chart is a truth-table of f where vertices of $B^n = \{0,1\}^n$ are arranged in a matrix. The columns of the matrix correspond to the vertices of $B^{|X|} = B^s$, and its rows to the vertices of $B^{|Y|} = B^{n-s}$. The

entries in $D(X|Y)$ are the values that f takes for all possible input combinations. For example, if $f(a,b,c) = abc' + a'c + b'c$, the decomposition chart for f for the partition $ab|c$ is

$\frac{ab}{c}$	00	01	10	11
0	0	0	0	1
1	1	1	1	0

Ashenhurst proved the following fundamental result, which relates the existence of a decomposition to the number of distinct columns in the decomposition chart $D(X|Y)$:

Theorem 3.1 (Ashenhurst) *The simple disjoint decomposition (2) exists if and only if the corresponding decomposition chart has at most two distinct column patterns.*

Stated differently, the decomposition (2) exists if and only if the **column multiplicity** of $D(X|Y)$ is at most 2. Note that the chart just shown has 2 distinct columns, 01 and 10.

We say that two vertices x_1 and x_2 in B^s (i.e., $B^{|X|}$) are **compatible** (written $x_1 \sim x_2$) if they have the same column patterns in $D(X|Y)$, i.e., $f(x_1, y) = f(x_2, y)$ for all $y \in B^{|Y|}$. For an incompletely specified function, a don't care entry '-' cannot cause two columns to be incompatible. In other words, two columns c_i and c_j are compatible if for each row k, either $c_i(k) = -$, or $c_j(k) = -$, or $c_i(k) = c_j(k)$. For a completely specified function f, compatibility is an equivalence relation (i.e., $x_1 \sim x_1, x_1 \sim x_2 \Rightarrow x_2 \sim x_1$, and $x_1 \sim x_2$ & $x_2 \sim x_3 \Rightarrow x_1 \sim x_3$ for all x_1, x_2, x_3), and the set of vertices that are mutually compatible (or equivalent) form an **equivalence class**. Hence the column multiplicity of the decomposition chart is the number of equivalence classes. In this paper, we will consider only completely specified functions, and so use compatibility and equivalence interchangeably.

Roth & Karp [2] extended the decomposition theory of Ashenhurst by characterizing a general (non-simple) disjoint decomposition, which is of the following form:

$$f(X,Y) = g(\alpha_1(X), \alpha_2(X), \ldots, \alpha_t(X), Y) = g(\vec{\alpha}(X), Y), \quad (3)$$

where $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_t)$. They proved that if k is the least integer such that $B^{|X|}$ may be partitioned into k equivalence classes (in other words, the column multiplicity of the decomposition chart $D(X|Y)$ is k), then there exist $\alpha_1, \alpha_2, \ldots, \alpha_t$ and g such that (3) holds if and only if $k \leq 2^t$. Hence the least t that satisfies (3) is $\lceil log_2 k \rceil$.

Suppose we have determined that there are k equivalence classes corresponding to the partition (X, Y) for the function f. The next question is how to determine sub-functions $\vec{\alpha} = (\alpha_1, \ldots \alpha_t)$ and g. We briefly review how Ashenhurst and Roth & Karp address this problem.

- Ashenhurst [1]: Given that the column multiplicity of $D(X|Y)$ is at most 2, how do we determine α and g? Since there are at most 2 equivalence classes, and a single α function for a simple decomposition, the vertices of one class are placed in the off-set of α, and of the other class in the on-set. g can then be determined by looking at each minterm in the on-set of f and replacing its bound-part (i.e., the literals corresponding to the variables in the bound set X) by either α or α', depending on whether the bound-part is in the class that was mapped to the on-set of α or the off-set. We illustrate the decomposition technique for the previous example - $f = abc' + a'c + b'c$, and partition $(X|Y) = ab|c$. $D(ab|c)$ has two distinct column patterns, resulting in the equivalence classes $C_1(a,b) = \{00, 01, 10\}$ and $C_2(a,b) = \{11\}$. Let us assign C_1 to the off-set of α and C_2 to its on-set. Then $\alpha(a,b) = ab$. Since $f = abc' + a'c + b'c$, $g(\alpha, c) = \alpha c' + \alpha' c + \alpha' c = \alpha \oplus c$.[2] The bound part of the first minterm abc' of f is ab, which yields $\alpha = 1$. So this minterm abc' generates $\alpha c'$ in g. Note that if C_1 was assigned to the on-set of α and C_2 to the off-set, the new $\widetilde{\alpha}$ would be simply α', and the new $\widetilde{g}(\alpha, c)$, $g(\alpha', c)$, which has same number of product terms as g. So irrespective of how we encode C_1 and C_2, the functions g have the same complexity. However, the situation is different if the decomposition is not simple.

- Roth & Karp [2] give conditions for the existence of $\vec{\alpha}$ functions, but do not give a method for computing them.[3] This is because they assume that a *library of primitive elements* is available from which $\vec{\alpha}$ functions are chosen. Given a choice of $\vec{\alpha}$ functions, they state the necessary and sufficient condition under which g exists as in (3).

  Proposition 3.2 (Roth & Karp) *Given f and $\vec{\alpha}$, there exists g such that (3) holds if and only if, for all $x_1, x_2 \in B^{|X|}, \vec{\alpha}(x_1) = \vec{\alpha}(x_2) \Rightarrow x_1 \sim x_2$, or equivalently, $x_1 \not\sim x_2 \Rightarrow \vec{\alpha}(x_1) \neq \vec{\alpha}(x_2)$.*

  In other words, whenever x_1 and x_2 are in different compatibility (or equivalence) classes, $\vec{\alpha}$ should evaluate differently on them. If this condition is not satisfied, then this particular choice $\vec{\alpha}$ of primitive elements is discarded, and the next one is tried. Otherwise, a valid decomposition exists, and then g is determined as follows. Each minterm in the on-set of f, written (x, y), where x is the bound-part and y the free-part, maps onto a minterm $(\widehat{\alpha_1}\widehat{\alpha_2}\ldots\widehat{\alpha_t}, y)$ in the on-set of g. Here

$$\widehat{\alpha_j} = \begin{cases} \alpha_j & \text{if } \alpha_j(x) = 1 \\ \alpha_j' & \text{if } \alpha_j(x) = 0 \end{cases} \quad (4)$$

  The entire procedure is repeated on g until it becomes equal to some primitive element.

In general, $\vec{\alpha}$ functions are not known *a priori*. For instance, this is the case when decomposition is performed during the technology-independent optimization phase, because the technology library of primitive elements is not considered. There are many possible choices for $\vec{\alpha}$ functions that correspond to a valid decomposition. For instance, given that $B^{|X|}$ may be partitioned into k classes of mutually compatible elements, and that $t \geq \lceil \log_2(k) \rceil$, each of the k compatibility classes may be assigned a unique binary code of length t, and there are many ways of doing this. Each such assignment leads to different $\vec{\alpha}$ functions. We will like to obtain that set of $\vec{\alpha}$ functions which is *simple* and which makes the resulting function g simple as well. The measure of simplicity is the size of the functions using an appropriate cost function. For instance, in the two-level synthesis paradigm, a good cost function is the number of product terms, whereas in the multi-level paradigm, it is the number of literals in the factored form. The general problem can then be stated as follows:

Problem 3.1 *Given a function $f(X,Y)$, determine sub-functions $\vec{\alpha}(X)$ and $g(\vec{\alpha}, Y)$ satisfying (3) such that an objective function on the sizes of $\vec{\alpha}$ and g is minimized.*

To the best of our knowledge, this problem has not been addressed in the past. We present an encoding-based formulation to solve it, and also show how the formulation becomes simpler for LUT architectures.

[2] $\oplus$ denotes XOR.

[3] We believe they knew how to find these functions, but not how to find simple $\vec{\alpha}$ functions.

4 Determining $\vec{\alpha}$ and g: an Encoding Problem

It seems intuitive to extend Ashenhurst's method for obtaining the $\vec{\alpha}$ functions. Ashenhurst placed the minterms of one equivalence class in the on-set of α and of the other in the off-set. In other words, one equivalence class gets the code $\alpha = 1$ and the other, $\alpha = 0$. For more than two equivalence classes, we can do likewise, i.e., assign unique $\vec{\alpha}$-codes to equivalence classes. This leads to the following algorithm:

1. Obtain a minimum cardinality partition $\mathcal{P}$ of the space $B^{|X|}$ into k compatible classes. This means that no two classes C_i and C_j of $\mathcal{P}$ can be combined into a single class $C_i \cup C_j$ such that all minterms of $C_i \cup C_j$ are mutually compatible. This means that given any two classes C_i and C_j in $\mathcal{P}$, there exist $v_i \in C_i$ and $v_j \in C_j$ such that $v_i \not\sim v_j$.

2. Then assign codes to the compatibility classes of $\mathcal{P}$. Since there is at least one pair of incompatible minterms for each pair of classes, it follows from Proposition 3.2 that each compatibility class must be assigned a unique code. This implies that all the minterms in a compatibility class are assigned the same code. We will discuss shortly how to assign codes to obtain simple $\vec{\alpha}$ and g functions.

This is the approach taken in every work (we are aware of) that uses functional decomposition, e.g., [6, 7]. However, this is not the most general formulation of the problem. To see why, let us re-examine Proposition 3.2, which gives necessary and sufficient conditions for the existence of the decomposition. It only constrains two minterms (in $B^{|X|}$ space) that are in different equivalence classes to have different values of $\vec{\alpha}$ functions. It says nothing about the minterms in the same equivalence class. In fact, there is no restriction on the $\vec{\alpha}$ values that these minterms may take: $\vec{\alpha}$ may evaluate same or differently on these minterms.

To obtain the general formulation, let us examine the problem from a slightly different angle. In Figure 2 is shown a function $f(X, Y)$ that is to be decomposed with the bound set X and the free set Y. After decomposition, the vertices in $B^{|X|}$ are *mapped into* vertices in B^t - the space corresponding to the $\vec{\alpha}$ functions. This is shown in Figure 3. This mapping can be thought of as an encoding. Assume a symbolic variable $\mathcal{X}$. Imagine that each vertex x in $B^{|X|}$ corresponds to a symbolic value of $\mathcal{X}$, and is to be assigned an $\vec{\alpha}$-code in B^t. This assignment must satisfy the following constraint: if $x_1, x_2 \in B^{|X|}$ and $x_1 \not\sim x_2$, they must be assigned different $\vec{\alpha}$-codes - this follows from Proposition 3.2. Otherwise, we have freedom in assigning them different or same codes. Hence, instead of assigning codes to classes, the most general formulation assigns codes to the minterms in the $B^{|X|}$ space.

The problem of determining simple $\vec{\alpha}$ and g can be represented as an input-output encoding (or state-encoding) problem. Intuitively, this is because the $\vec{\alpha}$ functions created after encoding are both inputs and outputs: they are inputs to g and outputs of the square block of Figure 3. Minimizing the objective for $\vec{\alpha}$ functions imposes output constraints, whereas minimizing it for g imposes input constraints.

There is, however, one main difference between the standard input-output encoding problem and the encoding problem that we have. Typically input-output encoding requires that each symbolic value be assigned a *distinct* code (e.g., in state-encoding), whereas in our encoding problem some symbols of $\mathcal{X}$ may be assigned the same code. This can be handled by a simple modification to the encoding algorithm. Let us first see how an encoding algorithm ensures that the codes are unique. A dichotomy-based algorithm [14] explicitly adds a dichotomy

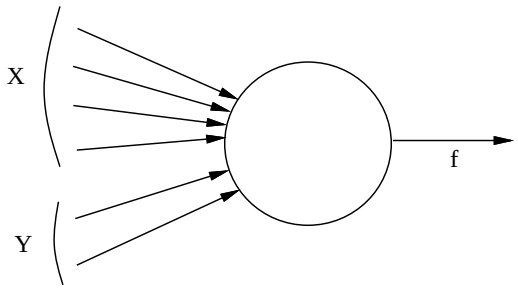

Figure 2: Function f to be decomposed with the bound set X and free set Y

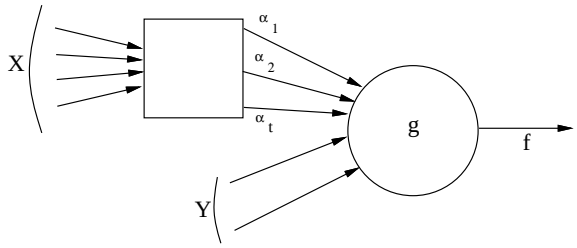

Figure 3: A general decomposition of f

$(S_i; S_j)$ for each symbol-pair $\{S_i, S_j\}$. This guarantees that the code of S_i is different from that of S_j in at least one bit. In our problem, let x_i and x_j be two symbolic values of $\mathcal{X}$. If $x_i \not\sim x_j$, we add a dichotomy $(x_i; x_j)$. Otherwise, no such dichotomy is added. This provides additional flexibility to the encoding algorithm: it may assign the same code to two or more compatible symbols if the resulting $\vec{\alpha}$ and g functions are simpler.

The encoding algorithm has to encode all the $2^{|X|}$ symbolic values of $\mathcal{X}$. If $|X|$ is large, the problem becomes computationally difficult. We can then use the approximate method of assigning codes to equivalence classes, as described at the beginning of this section.

Note that t is determined by the encoding algorithm. It is the number of bits used by the algorithm to encode the vertices in $B^{|X|}$, or the equivalence classes if the approximate method is being used. Once the codes are known, the $\vec{\alpha}$ functions can be easily computed. Then g can be determined using the procedure described in the last section. The unused codes can be used as don't cares to simplify g.

5 Application to LUT Architectures

We have shown that for a given input partition (X, Y), the general decomposition problem can be solved using an algorithm for input-output encoding. The input part is responsible for mini-

```
/* η is a network */
/* m is the number of inputs to the LUT */
functional_decomposition_for_LUT(η, m)
{
     while (nodes with support > m exist in η) do {
          n = get_an_m-infeasible_node(η);
          (X,Y) = get_input_partition(n);
          codes = encode(n, X);
          α⃗ = determine_α⃗(codes);
          g = compute_g(n, codes);
          g = simplify_g_using_DC(g, α⃗, codes);
          add α⃗ nodes to η;
          replace n by g
     }
}
```

Figure 4: Functional decomposition for LUT architectures

```
/* η is a network */
/* m is the number of inputs to the LUT */
approximate_functional_decomposition_for_LUT(η, m)
{
     while (nodes with support > m exist in η) do {
          n = get_an_m-infeasible_node(η);
          (X,Y) = get_input_partition(n);
          classes = form_compatibility_classes(n, X, Y);
          codes = encode(n, classes);
          α⃗ = determine_α⃗(classes, codes, X);
          g = compute_g(n, classes, codes, X);
          g = simplify_g_using_DC(g, classes, codes);
          add α⃗ nodes to η;
          replace n by g
     }
}
```

Figure 5: Approximate method for decomposition for LUT architectures

mizing the size of g, and the output part for minimizing the sizes of $\vec{\alpha}$. However, for LUT architectures, we are interested in a particular kind of decomposition: namely, where the bound set X is restricted to have at most m variables. Then, all the $\vec{\alpha}$ functions are m-feasible and can be realized with one m-LUT each. If $t + |Y| > m$, g needs to be decomposed further. Since an m-LUT can implement any function of up to m inputs, we do not care how large the representation of the $\vec{\alpha}$ functions is. The only concern from the output encoding part is the number of bits in the encoding. Since each extra bit means using an extra LUT, we will like to minimize the number of bits. So we use $t = \lceil log_2 k \rceil$. With this, the contribution by the $\vec{\alpha}$ functions to the objective function disappears. This removes the output encoding part of the formulation, thereby reducing it simply to an input encoding problem.

Note that if $t \geq |X|$, g will have at least as many inputs as f, and the algorithm may never terminate. So we always check for $t < |X|$.

Since LUTs impose input constraints, it is tempting to consider minimizing the support of the function g as the objective function in the encoding formulation. However, if the code-length is always chosen to be the minimum possible, the support of g is already determined (it is $t+|Y|$), and the encoding of $\vec{\alpha}$ functions do not make any difference. Hence, this objective function is not meaningful.

We show the complete algorithm in Figure 4. The approximate method, where equivalence classes are encoded, is shown in Figure 5 and is illustrated with the following example. Let

$$f(a,b,c,d,e) = ab' + ac' + ad + ae + a'e'$$

Let $m = 4$. Let us fix the bound set X to $\{a,b,c,d\}$. Then $Y = \{e\}$. Although we do not show the decomposition chart (since it is big), it has three equivalence classes C_0, C_1, and C_2. Let the corresponding symbolic representation for the on-set of g be:

e	*class*	g
1	C_0	1
1	C_1	1
0	C_2	1
0	C_0	1

Let us assume that we are minimizing the number of product terms in g. Then after a multi-valued minimization [4], we get the following cover:

e	C_0	C_1	C_2	g
1	1	1	0	1
0	1	0	1	1

This corresponds to the following face constraints:

C_0	C_1	C_2
1	1	0
1	0	1

To these, uniqueness constraints are added. These constraints are handed over to the constraint satisfier [11]. The following codes are generated:

class	$\alpha_1\alpha_2$
C_0	00
C_1	10
C_2	01

Note that C_0 and C_1 are on a face, namely $\alpha_2 = 0$. Similarly, C_0 and C_2 are on the face $\alpha_1 = 0$. Let α_1 and α_2 be the encoding variables used. Then it can be seen from the minimized multi-valued cover that

$$\begin{aligned} g &= e'(C_0 + C_2) + e(C_0 + C_1) \\ \Rightarrow g &= e'{\alpha_1}' + e{\alpha_2}' \end{aligned}$$

Also, it turns out that C_0, C_1, and C_2 are such that

$$\begin{aligned} \alpha_1 &= abcd' \\ \alpha_2 &= a' \end{aligned}$$

This simplifies to

$$\begin{aligned} g &= e'{\alpha_1}' + ea \\ \alpha_1 &= abcd' \end{aligned}$$

Had we done a $dumb$ encoding of the equivalence classes, as is the case in [6], we would have obtained the following decomposition,

$$\begin{aligned} g &= \alpha_1{\alpha_2}'e + {\alpha_1}'\alpha_2 + {\alpha_1}'e' \\ \alpha_1 &= abcd' \\ \alpha_2 &= ab' + ac' + ad, \end{aligned}$$

which uses one more function and many more literals than the previous one. This shows that the choice of encoding does make a difference in the resultant implementation.

6 Experiments

The experimental set-up is as follows. We take MCNC and ISCAS multi-level networks and optimize them by standard methods [5, 12]. We use *misII* for these experiments. There is an implementation of Roth-Karp decomposition algorithm in *misII* [6]. This implementation encodes the equivalence classes serially, that is, it assigns to an equivalence class C_j the code corresponding to the binary representation of j. To measure the encoding quality, we target LUT architectures, and our final goal is to minimize the number of Configurable Logic Blocks (CLBs) needed for a benchmark. A CLB is a basic block of the Xilinx 3090 architecture [3], which can implement either one 5-feasible function, or two 4-feasible functions with a total of at most 5 inputs. For encoding, we use the algorithm of [11], which targets two level minimization with two cost functions: number of cubes and number of literals. We go for minimum number of literals, since it is more relevant for a multi-level implementation.

The following experiments are performed:

- RK_mis-pga: Use the original Roth-Karp decomposition implementation of *misII* [5, 6]. This is followed by a *mis-pga* mapping script. The complete sequence of commands is:

  ```
  xl_k_decomp -n 5
  xl_partition -tm -n 5
  xl_merge
  ```

  The command `xl_k_decomp` invokes Roth-Karp decomposition on any node function that has greater than 5 fanins. It chooses the first input partition (X, Y) such that $|X| \leq 5$. If a disjoint decomposition is not found, the implementation switches to another decomposition method that guarantees feasibility [6].

  `xl_partition` reduces the number of nodes by collapsing them into their fanouts, without generating any nodes that have more than 5 inputs.

  `xl_merge` exploits the feature of Xilinx 3090 architecture that allows two functions to be placed on one CLB [3].

- RK_enc: Use the input encoding formulation while doing Roth-Karp decomposition. We use the approximate method, wherein the equivalence classes are encoded. The input encoding algorithm from [11] is used. `xl_k_decomp_input_encoding` is the corresponding new command. The following script is used:

  ```
  xl_k_decomp_input_encoding -n 5
  xl_partition -tm -n 5
  xl_merge
  ```

  We experiment with two options in `xl_k_decomp_input_encoding`:

  - without DC: no don't cares are used.
  - with DC: the unused codes are used as don't cares to simplify g.

Table 1 shows the results on the benchmarks. On a per example basis, RK_enc (with DC) is 16.5% better than RK_mis-pga. It helps to use unused codes as don't cares. RK_enc (with DC) is 6.6% better than RK_enc (without DC). Looking at the row **subtotal**, RK_enc (with DC) gives 21% better CLB count than RK_mis-pga. Also note that on *apex2* and *C5315*, RK_mis-pga could not complete, whereas RK_enc could.

We make another observation: most of the improvement is in the large benchmarks. This is because in small benchmarks, most of the functions are simple and do not have too many inputs. Therefore the sub-function g after applying the algorithm is m-feasible most of the time, and doing a good encoding does

example	RK_mis-pga	RK_enc	
		without DC	*with DC*
z4ml	7	7	7
misex1	10	10	10
vg2	27	23	28
5xp1	36	40	39
count	26	26	26
9symml	46	45	45
9sym	62	53	53
apex7	56	52	53
rd84	115	67	46
e64	54	54	54
C880	209	191	136
apex2	-	133	85
alu2	177	163	153
duke2	312	269	174
C499	73	68	67
rot	257	240	223
apex6	210	207	194
alu4	91	85	85
sao2	104	78	76
rd73	35	25	22
misex2	29	27	28
f51m	67	41	38
clip	114	73	54
bw	45	46	47
des	1373	1265	1151
C5315	-	442	455
b9	103	77	63
subtotal	3638	3232	2872
total	-	3807	3412

Table 1: Number of Xilinx 3090 CLBs

RK_mis-pga	Roth-Karp decomp. of *mis-pga*, followed by mapping script
RK_enc without DC	Roth-Karp decomp. with input encoding (and not using DCs), followed by mapping script
RK_enc with DC	Roth-Karp decomp. with input encoding and using unused codes as DC, followed by mapping script
-	could not finish
subtotal	sum of CLB counts for all examples except *apex2* and *C5315*
total	sum of CLB counts for all examples

not make much difference. But typically in larger benchmarks, functions have many inputs, so g is infeasible and doing a good encoding does make a difference when g is decomposed.

Although not reported here, the number of literals is also minimized in roughly the same proportion as the number of CLBs using the encoding formulation.

Note that for some benchmarks, such as *5xp1* and *bw*, the number of CLBs increases as a result of using input encoding techniques. Though counter-intuitive, it is not surprising. It just shows that the number of literals may not always be a good cost function for LUT architectures. A simple example is the following. Fix m to 5. Consider two functions f_1 and f_2:

$$\begin{aligned} f_1 &= abcdeg \\ f_2 &= abc + b'de + a'e' + c'd'. \end{aligned}$$

The representation of f_1 has 6 literals, and that of f_2 10 literals. However, f_1 requires two 5-LUTs, whereas f_2 only one. Therefore we need to come up with better cost functions for these architectures.

7 Conclusions

In this paper, we revisited the classical problem of functional decomposition. We showed how to solve the problem of decomposing a function such that the resulting sub-functions are *simple*, i.e., have small number of cubes or literals. We demonstrated that this problem is intimately related to the encoding problem. In general, an input-output encoding formulation has to be employed to solve the problem. However, for programmable gate array architectures that use look-up tables, the input encoding formulation suffices, provided we use minimum-length codes. We also use the unused codes as don't cares for simplifying the subfunctions.

Our approach gives promising results as compared to the original implementation of functional decomposition (which ignores the encoding problem) in *misII*.

The analysis presented in this paper assumes that the partition (X,Y) is known. The problem that remains unsolved is that of choosing a good input partition (X,Y).

For LUT architectures, minimizing the number of literals may not always be a good objective. We plan to come up with better objective functions in future.

Acknowledgements

We wish to thank Tiziano Villa for many helpful discussions and teaching us how to use the encoding tool [11]. We also thank Huey-Yih Wang for reading an earlier draft of this paper.

References

[1] R. L. Ashenhurst, "The Decomposition of Switching Functions", *Proc. of International Symp on Theory of Switching Functions*, 1959.

[2] J.P. Roth and R.M. Karp, "Minimization over Boolean graphs", *IBM Journal of Research and Development*, April 1962.

[3] Xilinx Inc., 2069, Hamilton Ave. San Jose, CA-95125, *The Programmable Gate Array Data Book.*

[4] R. K. Brayton, C. McMullen, G. D. Hachtel and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI synthesis*, Kluwer Academic Publishers, 1984.

[5] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD*, November 1987.

[6] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays", *Proc. 27th Design Automation Conference*, June 1990, pp. 620-625.

[7] Y. T. Lai, M. Pedram, and Sarma B. K. Vrudhula, "BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis", *Proc. 30th Design Automation Conference*, June 1993, pp. 642-647.

[8] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite state machines", *IEEE Transactions on Computer-Aided Design*, July 1985.

[9] G. De Micheli, "Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros", *IEEE Transactions on Computer-Aided Design*, Oct. 1986.

[10] S. Devadas and R. Newton, "Exact algorithms for output encoding, state assignment and four-level Boolean minimization", *IEEE Transactions on Computer-Aided Design*, Jan 1991.

[11] A. Saldanha, T. Villa, R. K. Brayton and A. Sangiovanni-Vincentelli, "A framework for satisfying input and output encoding constraints", *Proc. 29th Design Automation Conference*, June, 1992.

[12] A. Saldanha, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Multi-Level Logic Simplification using Don't Cares and Filters", *Proc. 26th Design Automation Conference*, 1989.

[13] L. Lavagno, T. Villa and A. Sangiovanni-Vincentelli, "Advances in encoding for logic synthesis", In *Progress in Computer Aided VLSI design*, G. Zobrist ed., in press, Ablex, Norwood, 1992.

[14] S. Yang and M. Ciesielski, "Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization", *IEEE Transactions on Computer-Aided Design*, January, 1991.

[15] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: State Assignment for optimal two-level logic implementations", *IEEE Transactions on Computer-Aided Design*, Sept. 1990.

[16] B. Lin, and A. R. Newton, "Synthesis of multiple level logic from symbolic high-level description languages", *Proc. of the International Conference on VLSI*, Munich, 1989.

[17] R. L. Rudell, "Logic Synthesis for VLSI Design", *UCB/ERL Memorandum M89/49*, April 1989.