# Functionally Linear Decomposition and Synthesis of Logic Circuits for FPGAs

Tomasz S. Czajkowski, *Member, IEEE*, and Stephen D. Brown, *Member, IEEE*

***Abstract*—This paper presents a novel XOR-based logic synthesis approach called Functionally Linear Decomposition and Synthesis (FLDS). This approach decomposes a logic function to expose an XOR relationship by using Gaussian Elimination. It is fundamentally different from the traditional approaches to this problem, which are based on the work of Ashenhurst and Curtis. FLDS utilizes Binary Decision Diagrams to efficiently represent logic functions, making it fast and scalable. This technique was tested on a set of 99 MCNC benchmarks, mapping each design into a network of four input lookup tables. On the 25 of the benchmarks, which have been classified by previous researchers as XOR-based logic circuits, our approach provides significant area savings. In comparison to the leading logic synthesis tools, ABC and BDS-PGA 2.0, FLDS produces XOR-based circuits with 25.3% and 18.8% smaller area, respectively. The logic circuit depth is also improved by 7.7% and 14.5%, respectively.**

***Index Terms*—Circuit synthesis, field-programmable gate arrays (FPGAs).**

## I. Introduction

THE problem of logic synthesis has been approached from various angles over the last 50 years. For practical purposes, it is usually addressed by focusing on either AND/OR, multiplexor, or XOR-based logic functions. Each of the aforementioned types of logic functions exhibits different properties, and usually a synthesis method that addresses one of these types very well does not work well for others.

The XOR-based logic functions are an important type of functions as they are heavily used in arithmetic, error correcting, and telecommunication circuits. It is challenging to synthesize them efficiently as there are seven different classes of XOR logic functions [4], each of which has distinct approaches that work well to implement logic functions of a given class. In this paper, we focus on XOR-based logic functions and show that they exhibit a property that can be exploited in an area-driven computer-aided design (CAD) flow.

A wealth of research on XOR logic decomposition and synthesis exists. Some of the early work was geared toward spectral decomposition [5], in which transforms are applied to a logic function to change it into a different domain, where analysis could be easier. It is analogous to how real time signals are analyzed in the frequency domain using tools like the Fourier transform. While at the time, the methods were exceedingly time consuming, the advent of Binary Decision Diagrams (BDDs) accelerated the computation process and showed practical gains for using spectral methods [6].

In the late 1970s, Karpovsky [7] presented an approach called linear decomposition, which decomposed a logic function into two types of blocks: linear and nonlinear. A linear block was a function consisting purely of exclusive OR (XOR) gates, while a nonlinear block represented functions that require other gates to be completely described (AND, OR, NOT). This approach was found to be effective for arithmetic, error correcting, and symmetric functions.

More recently, XOR-based decompositions were addressed using Davio expansions [8], and with the help of BDDs [9], [10]. With Davio Expansions, Reed–Muller logic equation that utilizes XOR gates can be generated for a function. On the other hand, the idea behind using BDDs was to look for x-dominators in a BDD that would indicate a presence of an XOR gate, and could be used to reduce the area taken by a logic function. Moreover, tabular methods based on the work of Ashenhurst and Curtis [1]–[3] have been used to perform XOR-based decomposition [11].

This paper addresses two limitations of the aforementioned methods. First, Davio expansions perform decomposition one variable at a time, so an XOR relationship between non-XOR functions may not necessarily be found. Second, BDD-based methods either use x-dominators to perform a nondisjoint decomposition or rely on BDD partitioning, which clings to the idea of column multiplicity introduced by Ashenhurst and Curtis [1]–[3]. The latter is inadequate for XOR-based logic synthesis as column multiplicity does not address the XOR relationship between the columns of a truth table.

In this paper, we introduce a novel approach that is based on the property of linearity. Rather than looking at linear blocks as in [7], x-dominators, or partitioning a BDD, this approach exploits a linear relationship between logic subfunctions. Thus, we define *functional linearity* to describe a decomposition of a Boolean function into the following form:

$$f(X) = \sum_i G_i(Y) H_i(X - Y) \tag{1}$$

where $X$ and $Y$ are sets of variables such that $Y \subseteq X$, while the summation represents an XOR gate. In this representation, the function $f$ is a weighted sum of functions $G_i$, where the weighting factors are defined by functions $H_i$. This approach

Manuscript received April 1, 2008; revised June 27, 2008. Current version published November 19, 2008. This work was supported in part by Altera Corporation and in part by the University of Toronto. This paper was recommended by Associate Editor S. Nowick.

T. S. Czajkowski is with the University of Toronto, Toronto, ON M5S 1A1, Canada (e-mail: czajkow@eecg.utoronto.ca).

S. D. Brown is with the University of Toronto, Toronto, ON M5S 1A1, Canada, and also with Altera Toronto Technology Centre, Toronto, ON M5S 1S4, Canada.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2008.2006144

retains both the ability to synthesize XOR logic functions using Davio and Shannon's expansions [4], as well as the ability of a BDD to find relationships between logic functions, but does not rely on the idea of column multiplicity.

Our logic synthesis method utilizes Gauss–Jordan Elimination to decompose a logic function into sets of functions and synthesizes the original function by a linear combination of these functions, as in (1). Results show that, on average, our approach produces better results for XOR-based logic circuits in comparison to the state-of-the-art synthesis tools.

An abbreviated version of this paper appeared in [19]. In addition to the work presented in [19], this paper contains additional algorithms, discussion of performance and FPGA considerations, an in-depth discussion of related works, and more detailed results on a wider set of benchmark circuits.

In this paper, we give a comprehensive description of the work as follows: Section II contains the background information for our synthesis approach. Section III details the approach for single output logic synthesis and Section IV presents the variable partitioning algorithm used in our technique. Section V discusses an algorithm to further reduce the area of a logic network. In Section VI, we extend the approach to multioutput logic synthesis, while Section VII discusses the nuances of the proposed logic synthesis approach and highlights key points required for reducing synthesis time. In Section VIII, we address some FPGA specific synthesis considerations, and in Section IX, we discuss prior publications related to this paper. In Section X, we present the results obtained using our method and compare them to ABC [18] as well as BDS-PGA 2.0 [10]. We conclude this paper and discuss the topics of future work in Section XI.

## II. Background

In this section, we focus on the description of basic concepts of logic synthesis and linear algebra used in this paper.

### A. Logic Synthesis

In the last 50 years, numerous approaches to logic synthesis emerged, which generally fall into one of the following three categories:

1) tabular methods;
2) algebraic methods;
3) BDD-based methods.

The tabular methods are based on the work of Ashenhurst and Curtis [1]–[3] in which a logic function is represented as a truth table, where both rows and columns of the table are associated with a particular valuation of input variables. Ashenhurst and Curtis chose to represent a logic function in a tabular form to exploit the idea of *column multiplicity*. This concept is shown on an example in Fig. 1.

In this example, the truth table consists of eight columns. The columns are indexed by variables $abc$, called the free set, and the rows are indexed by variables $de$, called the bound set. A closer look at the table reveals that there are four unique columns, each of which is used multiple times (2) in the table; hence, the term column multiplicity. An easy way to realize

| $de$ \ $abc$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 11 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |

Fig. 1. Column multiplicity example.

this function is to synthesize each unique column, and AND it with a function to specify in which columns it is used. For example, columns 000 and 010 are identical and implement function $h_1 = de$. The function to describe in which column it is used is $g_1 = \bar{a} \cdot \bar{c}$. If we process every unique column in this way, producing functions $h_i$ and $g_i$, we can synthesize the original function as

$$\bigcup_{i=1}^{4} h_i g_i.$$

Clearly, the fewer the number of distinct columns, the easier it is to synthesize a logic function. Examples of such methods are shown in [3] and [11].

Algebraic methods extract a subfunction of a logic expression by processing a logic equation for a given function. The basic idea is to apply tools such as the Shannon's expansion, the DeMorgan's Theorem, or the Davio expansions to reduce the complexity of a logic expression. By applying these and other logic reduction techniques, an equation can be simplified and, thus, implemented using fewer gates. Representative samples of such approaches are discussed in [3], [8], [12], and [23].

Finally, BDD-based methods use a BDD [13] to represent a logic function and manipulate it to reduce the area a logic function occupies. As the size of the BDD and the area occupied by a logic function are related to one another [24], this approach can be effective in reducing area of logic functions. In the context of XOR synthesis BDD-based approaches utilize the concept of x-dominators to determine if XOR gates are present in the logic function. An x-dominator in a BDD is a node to which a regular and a complemented edge point to. Intuitively, an x-dominator points to a node such that the function it implements can be either zero or one for the same input valuation and depends on the state of the variables above it. Thus, an XOR relationship is exposed. Examples of BDD-based approaches are discussed in [3], [9], [10], [14], and [15].

Many of the aforementioned methods have been implemented in synthesis tools readily available to researchers. Such tools include SiS [21], ABC [18], and BDS-PGA 2.0 [10].

### B. Linear Algebra

The topic of this section concerns fields, vector spaces, and Gauss–Jordan Elimination. We review these topics in detail.

In mathematics, a field F is defined as a collection of symbols S as well as summation $(+)$ and multiplication $(*)$ operators and constants 0 and 1 that satisfy the following axioms;

1) $+$ and $*$ are associative and commutative;
2) Multiplication distributes over addition;

3) 0 is an additive identity, 1 is a multiplicative identity $(0 \neq 1)$;
4) Additive and multiplicative inverses exist.

It is also important to note that the result of any operation on elements of F must produce an element that is also in F. By using elements of a field F as entries in an $n$-tuple, we can form vectors that satisfy the properties of vector spaces. Namely, we can scale and add vectors together, and the resulting vector will also have entries that belong to the field F. A collection of such vectors is then be said to form a vector space.

A very important property of a vector space is that it is spanned by a subset of the vectors it contains. We illustrate this on an example. Consider the following five vectors:

$$\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}2\\3\\0\end{bmatrix}\begin{bmatrix}6\\5\\0\end{bmatrix}\begin{bmatrix}2\\2\\0\end{bmatrix}.$$

Notice that the third entry is 0 for each vector. Further inspection reveals that each vector in the set is a weighted sum of the two leftmost vectors. We therefore say that the two leftmost vectors are a *basis* for the vector space consisting of the given set of vectors. This is a very powerful idea as, among other applications, it clearly identifies basic components that form a vector space.

While the set of basis vectors was easy to find in the aforementioned example, a general procedure to find a basis for any vector space exists. Suppose that we take six vectors and represent each vector as a column of a matrix [16]

$$\begin{bmatrix}1 & -3 & 4 & -2 & 5 & 4\\ 2 & -6 & 9 & -1 & 8 & 2\\ 2 & -6 & 9 & -1 & 9 & 7\\ -1 & 3 & -4 & 2 & -5 & -4\end{bmatrix}.$$

To find a basis for the columns of the matrix, we first apply Gaussian Elimination to the matrix.

Gaussian Elimination is a process of applying elementary row operations (addition, subtraction and scalar multiplication) to the rows of the matrix to reduce the matrix to an upper triangular matrix with leftmost elements set to 1. To do this, we first pick a row with the nonzero entry in the leftmost column. In this example, let us pick the top row. We then look through all other rows that have a nonzero entry in the same column and try to make their entries in the same column a zero. To do that, we subtract a scalar multiple of the top row from a scalar multiple of the other rows. For example, the entry in the first column of the second row can be changed to a zero by subtracting twice the first row from the second. By following this procedure, we reduce the matrix to a row-echelon form [16]:

$$\begin{bmatrix}1 & -3 & 4 & -2 & 5 & 4\\ 0 & 0 & 1 & 3 & -2 & -6\\ 0 & 0 & 0 & 0 & 1 & 5\\ 0 & 0 & 0 & 0 & 0 & 0\end{bmatrix}.$$

To obtain the basis for the column vectors, we now look at each nonzero row of the resulting matrix. For each of those rows, the column with the leading 1 corresponds to the column in the original matrix that is a basis vector for the space spanned by the column vectors [16]. In this case, these are columns one, three, and five, and therefore, the following are the basis vectors for the column vector space:

$$\begin{bmatrix}1\\2\\2\\-1\end{bmatrix}\begin{bmatrix}4\\9\\9\\-4\end{bmatrix}\begin{bmatrix}5\\8\\9\\-5\end{bmatrix}.$$

Notice that each column in the original matrix can be represented as a weighted sum of the aforementioned basis vectors.

Gaussian Elimination is sometimes carried out one step further. That is, in addition to reducing the matrix to a row-echelon form, we further reduce the matrix so that the leading one of each row is the only nonzero entry in the corresponding column. When the procedure is carried out to this point, it is referred to as Gauss–Jordan Elimination [16], and the resulting matrix is said to be in the reduced row-echelon form. This is useful for solving a system of linear equations.

For the purpose of logic synthesis of digital circuits, the mathematical operations performed here are in a modulus field, also called a Galois Field. A Galois Field with modulus $m$ is denoted as GF($m$). In particular, we are interested in GF(2), because it facilitates operations on Boolean logic functions. In this field, the additive operator is the XOR operation, while a logical AND is equivalent to multiplication.

### C. Notations

In the previous sections, we introduced background information, however, some notation and terms used in logic synthesis may appear to conflict with the well-established norms in linear algebra. Thus, we explicitly state the notation used throughout the remainder of this paper.

In this paper, matrix notation is used with the assumption that all operations are performed in GF(2). Thus, summation is equivalent to an XOR operation, and multiplication is equivalent to a logical AND operation. We use $\oplus$ to represent a sum and a $+$ to represent a logical OR operation. In addition, we will use $\uparrow$ to represent a NAND operation and $\odot$ to denote an XNOR.

A matrix is denoted by a capital letter, while a column vector is identified by a lower case bold letter. For example, $\mathrm{A}\mathbf{x} = \mathbf{b}$ indicates an equation where a matrix A is multiplied by a column vector $\mathbf{x}$, and the result is a column vector $\mathbf{b}$. To distinguish regular column vectors from basis vectors, we denote basis vectors with bold capital letters. To distinguish input variables of a logic function from matrices and vectors, input variables of a logic function, as well as the function output, are italicized (e.g., $f = abcd$).

In this paper, linear algebra is used in the context of logic synthesis, hence we often represent columns/rows of a matrix as a logic function, rather than as column vectors of 0s and 1s. For example, if a column has four rows, indexed by variables $a$ and $b$, then we index the rows from top to bottom as 00, 01, 10, and 11. Thus, a statement $\mathbf{X} = a$ indicates that the basis vector $\mathbf{X}$ is $[0011]^{\mathrm{T}}$, whereas $\mathbf{Y} = b$ indicates a basis vector $[0101]^{\mathrm{T}}$.

| $cd$ \ $ab$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 1 | 1 |
| 10 | 0 | 1 | 0 | 1 |
| 11 | 0 | 1 | 1 | 0 |

Fig. 2. Truth Table for Example 1.

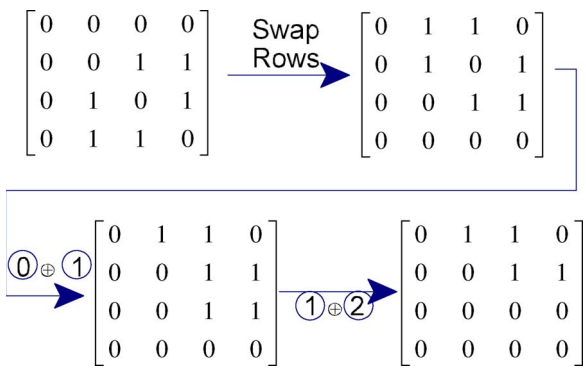


Fig. 3. Gaussian Elimination applied to Example 1.

## III. Basic Approach

Functionally Linear Decomposition and Synthesis (FLDS) is an approach that exploits an XOR relationship between logic functions. It gets its name from the way an XOR relationship is derived, by using the basis of the column space of a truth table to decompose a function into a set of subfunctions.

The basic idea behind this method is to find subfunctions that can be reused in a logic expression and, thus, reduce the size of a logic function implementation. It is analogous to Boolean and algebraic division, or kernel extraction. In these methods, we first derive a divisor, or a kernel, and then decompose a logic function with respect to it. In FLDS, we seek to achieve a similar effect, however, we take advantage of linear algebra to do it. In our approach, a basis vector is analogous to a "divisor," and a selector function to specify where a basis vector is used is analogous to a "quotient" from algebraic division. The key advantages of our method are the following: a) We can compute basis and selector functions simultaneously; b) the initial synthesis result can easily be optimized using linear algebra; and c) the method is computationally inexpensive even for large functions. In this section, we explain the basics of our approach, and then present methods to refine the synthesis results in subsequent sections.

Consider a logic function represented by a truth table in Fig. 2. The figure shows a truth table for logic function $f = ad \oplus bc$, with variables $ab$ at the top and variables $cd$ on the left-hand side. We will first decompose this function and then synthesize it.

The first step is to find the basis vectors for the truth table shown in Fig. 2. To do this, we will apply Gaussian Elimination to the aforementioned truth table, as though the truth table was a matrix in GF(2). The procedure, as it is applied to this truth table, is shown in Fig. 3.

We begin with the initial matrix that directly represents the truth table and swap the rows, such that the rows are ordered from top to bottom based on the column index of their respective leading one entries. The next step is to perform elementary row operations to reduce the matrix to the row-echelon form [16]. Thus, we replace row 1 with the sum [XOR in GF(2)] of rows 0 and 1. This causes row 0 to be the only row with a 1 in the second column. Finally, we replace row 2 with the sum of rows 1 and 2, making row 2 consist of all zeros.

In the resulting matrix, the first two rows have leading ones in the middle two columns. From linear algebra [16], we know this to indicate that the middle two columns in the original truth table, or equivalently columns $ab = 01$ and $ab = 10$, are the basis vectors for this function. Therefore, this function has two basis vectors, $\mathbf{G_1} = c$ and $\mathbf{G_2} = d$.

The next step is to express the entire truth table as a linear combination of these two vectors, which means expressing each column $C_i$ as $h_{1i}\mathbf{G_1} \oplus h_{2i}\mathbf{G_2}$, where $h_{1i}$ and $h_{2i}$ are constants. To find $h_{1i}$ and $h_{2i}$, we solve a linear equation $\mathbf{Ax} = \mathbf{b}$, where A is a matrix formed by basis vectors $\mathbf{G_1}$ and $\mathbf{G_2}$, $\mathbf{x} = [h_{1i}\, h_{2i}]^{\mathrm{T}}$, and $\mathbf{b}$ is the column, we wish to express as a linear combination of basis vectors $\mathbf{G_1}$ and $\mathbf{G_2}$. For example, to express column $C_3$, we solve the following equation:

$$\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} h_{1i} \\ h_{2i} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}. \tag{2}$$

By inspection, the solution to this equation is $h_{1i} = 1$ and $h_{2i} = 1$.

We now need to form selector functions that will identify the columns in which a given basis vector appears. To create a selector function for basis vector $\mathbf{G_1}$, we look at the columns for which $h_1$ was found to be 1. These columns are $ab = 01$ and $ab = 11$. Thus, the selector function $H_1 = b$. Similarly, the selector function for $\mathbf{G_2}$ is found to be $H_2 = a$. Finally, we can use (1) to synthesize the function $f$ as $H_1\mathbf{G_1} \oplus H_2\mathbf{G_2}$, producing logic expression $f = bc \oplus ad$.

## IV. Heuristic Variable Partitioning

The success of a logic synthesis approach depends heavily on how variables are chosen during logic decomposition. In fact, by varying the assignment of variables to rows and columns of a truth table, we can create a table with varying number of basis vectors, costs of basis, and selector functions, or both. The problem then becomes, as with most synthesis approaches, how to order, or in our case partition, variables. The problem can be demonstrated on the following example.

Consider the function $f = (a \odot c)(b \odot d)$, with variables $a$ and $b$ indexing the columns and $c$ and $d$ indexing the rows, as shown in Fig. 4. By following the procedure outlined in Section III, we find four basis vectors and synthesize it as the XOR of four minterms. To synthesize the function using XOR gates efficiently, we need to rearrange variables in the truth table such that rows, or columns, of the truth table are indexed by variables $a$ and $c$. Rearranging the variables this

| cd \ ab | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 0 |
| 01 | 0 | 1 | 0 | 0 |
| 10 | 0 | 0 | 1 | 0 |
| 11 | 0 | 0 | 0 | 1 |

Fig. 4. Truth Table for Example 2.

```
Partition_Variables(f, n, m) {
  BG(1) = {{x_i}| x_i ∈ sup(f)}
  for (k=2; k ≤ m; k=k*2) {
    G(k) = {a_i ∪ a_j | a_i, a_j ∈ BG(k/2), a_i ∩ a_j = ∅}
    BG(k) = ∅
    While(G(k)-BG(k) ≠ ∅) {
      g = {a_i ∈ G(k) | ∀a_j ∈ [G(k) − a_i], a_i ∩ a_j = ∅,
                        cost(a_i) ≤ cost(a_i) }
      BG(k) = BG(k)∪{g}
      G(k) = G(k) - {g}
    }
  }
  If m is not a power of 2 then {
    G(m) = all possible combinations of BG(2^i) that form a
      set of m variables, such that i^th bit of m = 1
    BG(m) = lowest cost grouping from G(m)
  }
  Pick Best Grouping in BG(m).
  Reorder variables in f to match the best grouping.
}
```

Fig. 5. Heuristic variable partitioning algorithm.

way will show that only one basis vector $(a \odot c)$ is present with selector $(b \odot d)$, and the resulting function will be expressed as $f = (a \odot c)(b \odot d)$.

While many variable ordering approaches exist most of them are designed for tabular methods or for BDDs. None of these directly expose basis functions of a logic expression and are thus inadequate for our purposes. Hence, we introduce a new heuristic variable partitioning algorithm that takes advantage of functional linearity. The algorithm is shown in Fig. 5. It takes as input a logic function $f$ of $n$ variables and the number of variables to be assigned to index rows $m$ (size of the bound set). It produces a variable partitioning that is suitable for the decomposition.

The algorithm works by incrementally partitioning variables into groups that have $2^k$ variables, where $2^k < m$. The algorithm begins with a setup of a set of elements $\mathbf{BG}(1)$, where each element is a set consisting of a single variable from the support of function $f$ (line 2). Then, the algorithm proceeds in a loop (lines 3–12) to create sets $\mathbf{BG}(k)$, where each element is a partition consisting of $k$ input variables. During the first pass of the loop, $k = 2$, the algorithm partitions variables into groups of two variables, such that each variables appears in at most one group. Each combination of variables is created and evaluated by determining the number of basis functions created if the given set is used to index rows as well as computing the cost of basis and selector functions. The cost of a function, basis or selector, is determined by the size of its support plus one, unless the function evaluates to zero in which case the cost is equal to zero.

Once each grouping of two variables is evaluated, the algorithm retains only $n/2$ least cost groupings, keeping in mind that each variable may only appear in one grouping. The procedure then repeats in exactly the same fashion for larger groupings, increasing the number of variables in each grouping by a factor of two on each iteration. For example, during the second pass, the algorithm creates groupings of four variables using the groupings of size two created previously as a starting point in order to reduce the overall runtime of the algorithm.

In a case where $m$ is not a power of two, the algorithm proceeds to evaluate all partitions of size $m$ by putting together input variables and groupings generated thus far (lines 13–17). For example, if $m = 5$, then for each grouping of size 4, the algorithm combines it with an input variable to make a grouping of five variables. The algorithm then evaluates the grouping and keeps it if it has lower cost than any other grouping of size 5.

The algorithm presented here is a heuristic and as such does not explore the entire solution space in the interest of reducing computation time. The number of partitions tested by our algorithm is a function of bound set size $m$ and number of variables $n$. When $m$ is a power of two, the number of partitions tested is $\sum_{i=0}^{\log_2(m)-1} \binom{2^{-i}n}{2}$, and the algorithm runtime is bounded by $O(\mathrm{n}^2)$. The number of partitions tested is larger when $m$ is not a power of two. The increase only becomes significant after $m = 15$, however, we found that after $m = 12$, the Gaussian Elimination algorithm begins to slow down significantly. As a result, we placed a limit of $m = 12$ on the algorithm to prevent Gaussian Elimination from consuming too much processing time.

To evaluate the effectiveness of this heuristic, we compared the area results obtained by FLDS when using the aforementioned algorithm to an exhaustive search and a random variable partitioning. The exhaustive search proved to generate the best results, while a random partitioning provided on average the worst results. Our proposed variable partitioning algorithm proved to be a close second to the exhaustive search for XOR based, or a symmetric, logic circuit with a distinct advantage in runtime.

## V. Basis and Selector Optimization

In the previous section, we described the variable partitioning algorithm that determines which variables should belong to the bound set and which to the free set. However, a proper variable assignment is not always sufficient to optimize area of a logic function. This is because for any given variable partitioning, there can exist many sets of basis vectors. While each set will have the same size, the cost of implementing each set of basis functions, and their corresponding selector functions, may be different.

To illustrate this point, consider the example in Fig. 6. This is the same function as in Example 1, but this time the variables are partitioned differently. Despite that, we can still synthesize this function optimally. Instead of reordering variables, we can optimize basis and selector functions.

In this particular example, we have two basis vectors: $\mathbf{G_1} = bc$ and $\mathbf{G_2} = b \uparrow c$. The selector functions corresponding to

| $bc$ \ $ad$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 1 |
| 01 | 0 | 0 | 0 | 1 |
| 10 | 0 | 0 | 0 | 1 |
| 11 | 1 | 1 | 1 | 0 |

Fig. 6. Truth Table for Example 3.

```
Optimize_Basis_and_Selectors(basis_fs, selector_fs)
    Cost = Σ_i cost(basis_fs_i) + Σ_j cost(selector_fs_j)
    repeat {
        For i=0 to m-2 do
            For j=i+1 to m-1 do
                nb = basis_fs(j) XOR basis_fs(i)
                ns = selector_fs(j) XOR selector_fs(I)
                n_cost = cost(nb)+cost(ns)
                If n_cost≤[cost(basis_fs(i))+cost(selector_fs(j))]
                    basis_fs(i) = nb
                    selector_fs(j) = ns
                else
                If n_cost≤[cost(basis_fs(j))+cost(selector_fs(i))]
                    basis_fs(j) = nb
                    selector_fs(i) = ns
                Update cost
    } until (last 3 iterations produced the same cost)
```

Fig. 7. Basis-selector optimization algorithm.

these basis functions are $H_1 = a \uparrow d$ and $H_2 = ad$. It is possible to replace one of the basis functions with $\mathbf{G_1} \oplus \mathbf{G_2}$ and still have a valid basis function. Notice that $\mathbf{G_1} \oplus \mathbf{G_2} = \mathbf{1}$ and, thus, has a smaller cost than $\mathbf{G_2}$ itself. It is therefore better to use $\mathbf{G_1} \oplus \mathbf{G_2}$ and $\mathbf{G_1}$ to represent the function in Fig. 6. By inspection, the function $f$ now has basis functions $\mathbf{G_2'} = \mathbf{G_1} \oplus \mathbf{G_2} = \mathbf{1}$ and $\mathbf{G_1} = bc$. The corresponding selector functions are now $H_2' = ad$ and $H_1 = 1$. This is because the fourth column of the truth table can now be represented as $\mathbf{1} \oplus \mathbf{G_1} = \mathbf{G_2}$. We can now synthesize function $f$ as $f = (H_1\mathbf{G_1}) \oplus (\mathbf{G_2'}H_2') = (bc(1)) \oplus (ad(1)) = ad \oplus bc$. We obtain the same result as in Section III.

The basis replacement idea is to keep the set of basis functions linearly independent of one another, while reducing the cost of basis and selector functions in the process. Clearly, the basis function pairs $\{\mathbf{G_1}, \mathbf{G_2}\}$, $\{\mathbf{G_1}, \mathbf{G_2'}\}$, $\{\mathbf{G_2'}, \mathbf{G_2}\}$ are all valid sets of basis functions, as neither element can be represented as a linear combination of the others. Moreover, each set is capable of expressing each column in the original matrix. The only consideration left is how the basis replacement affects the corresponding selector functions.

A careful analysis reveals that a basis function $\mathbf{G_2}$ can be replaced by $\mathbf{G_1} \oplus \mathbf{G_2}$, but it requires the selector function for basis $\mathbf{G_1}$ to be replaced with $H_1 \oplus H_2$. This is because we must now use basis function $\mathbf{G_1}$ wherever $\mathbf{G_2}$ was originally by itself. In addition, wherever $\mathbf{G_1}$ and $\mathbf{G_2}$ were both used, now only the new basis function $\mathbf{G_2'} = \mathbf{G_1} \oplus \mathbf{G_2}$ is used and hence basis function $\mathbf{G_1}$ should not be used.

Based on the aforementioned analysis, Fig. 7 shows the basis–selector optimization algorithm. The algorithm takes as input the set of basis functions and their corresponding selector functions and returns a modified set of basis and selector functions. The algorithm works as follows.

First, the algorithm computes the cost of each basis and selector function (line 2). The cost is determined by the support set size of each function, plus 1, with the exception of a function equal to zero, for which the cost is zero. The algorithm then proceeds to evaluate possible improvements to basis and selector functions. This is done by putting together a pair of basis–selector function pairs (lines 3–17) to determine if the resulting basis and selector function can replace one of the existing ones and reduce the basis/selector cost. If replacing an existing basis with a new one does not increase the overall cost of implementing a logic function (lines 9–15), then the algorithm replaces one of the basis and selector functions. Otherwise, the algorithm attempts to combine a different set of basis and selector functions.

The algorithm iterates until the last three iterations produce the same basis–selector cost, which indicates that further improvement is unlikely. Note that the algorithm allows for basis–selector replacement even if the cost is unchanged, because it may sometimes be necessary to go through a few intermediate steps before a lower cost basis–selector pair is found.

When this algorithm is applied to the truth table in Fig. 6, it finds that using basis function $\mathbf{G_1} \oplus \mathbf{G_2} = \mathbf{1}$, in place of $\mathbf{G_1}$ or $\mathbf{G_2}$ is preferable. An important point to notice in this example is that an all 1s column used does not appear in the truth table in Fig. 6. It shows one of the key distinctions between this method and the classical approach of Ashenhurst and Curtis [1]–[3].

## VI. Multioutput Synthesis With FLDS

The decomposition and synthesis procedure outlined in Section III addresses the issue of synthesizing a single output logic function. While it illustrates how the approach generates a two-level logic network, it is important to extend a synthesis technique to generate multilevel networks as well as the synthesis of logic functions with multiple outputs.

In this section, we discuss two aspects of multilevel and multioutput function synthesis with FLDS. First, we show how the function decomposition of a single output function may result in a multiple output logic function synthesis. We then show how different approaches to multiple output function decomposition work in the context of FLDS. Finally, we present a multioutput synthesis algorithm implement in FLDS.

### A. Multilevel Decomposition

A multilevel decomposition of a logic function is needed when the final logic expression requires more than two gates on the longest input to output path. In the context of FPGAs, the depth metric is the number of lookup tables (LUTs) on the longest input to output path. When a logic function has more than $k$ inputs, it may be necessary for more than 2 $k$-LUTs to be created on an input to output path to correctly synthesize a function.

With FLDS, there are two ways in which this situation can arise. One case is when the number of basis and selector

| | | $f$ | | | | $g$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $cd$ | | | | $ce$ | | | |
| $ab$ | | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 |
| | 00 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| | 01 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | 10 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | 11 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |

Fig. 8. Multioutput synthesis example.

functions exceeds $k/2$, in which case AND/XOR logic created to combine them will require more than one level of LUTs to implement. This is not an issue, because the synthesis technique has already addressed this by creating the AND/XOR logic, and a function will be synthesized correctly.

The second case is more interesting, and that is when either the basis or the selector function have more than $k$ inputs. Such a case requires further decomposition. It is important to notice though that in such a case, the variables for the basis functions come from a shared set (the selector functions also share variables), which presents an opportunity for further area reduction by extracting their common subfunctions.

### B. Multioutput Synthesis

With FLDS, a set of functions that share a subset of variables can be synthesized rather easily. A way to do this is to select a subset of common variables as the bound set and put the truth tables for all functions in a single table. We describe this procedure on the following example.

Consider two logic functions, $f$ and $g$, both with four input variables. Both functions share variables $a$, $b$, and $c$, while inputs $d$ and $e$ are the fourth input to functions $f$ and $g$, respectively. For this example, we chose to put variables $a$ and $b$ on the rows of the truth table, as shown in Fig. 8. If these two functions were to be synthesized separately, function $f$ would be found to have two basis vectors, while function $g$ would have three basis vectors. However, when synthesized together, there are only three basis vectors (second, third, and fifth column) for the two functions put together.

We can now apply the same steps as for single output function synthesis to further optimize the logic function. In this case, we can use basis and selector optimization algorithm to determine that a simpler function implementation can be found when an all 1s column is used instead of the fifth column.

The function $f$ is synthesized as $f = (a + b)d \oplus abc$ and shares its basis functions with $g$. For function $g$, we find that the basis function from the second column is used in column $ce = 10$, the basis function from the third column is used in columns $ce = \{01, 10\}$, while the all 1s column is used in every column of function $g$. Once we synthesize each basis function and its selector function, we can see that the synthesis of the all 1s column, and its selector function simplify to a constant 1. A constant one input to a three-input XOR gate reduces it to a two-input XNOR gate. The resulting circuit is shown in Fig. 9.

The aforementioned example showed how FLDS can be used to decompose a pair of functions that share variables to reduce their total area. While in some cases, choosing a subset

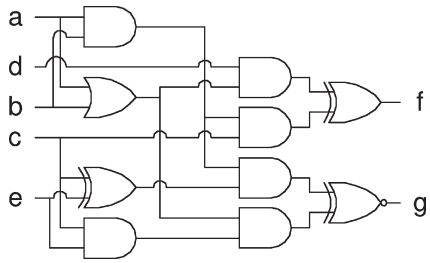


Fig. 9. Synthesized functions $f$ and $g$.

| | $C_{out}$ | | | | $S_1$ | | | | $S_0$ | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $x_0\ y_0$ | | | | $x_0\ y_0$ | | | | $x_0\ y_0$ | | | |
| $x_1 y_1$ | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 | 00 | 01 | 10 | 11 |
| 00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| 01 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 10 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 11 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |

Fig. 10. Two-bit ripple carry adder example.

of shared variables to index the rows is desirable, it is not always the case. It is also possible to use unique variables from one function to index the rows to get better results. Using this approach, we can synthesize arithmetic functions, like a ripple carry adder, efficiently. To demonstrate this, consider the example of a 2-b ripple carry adder.

Fig. 10 shows a truth table for the carry out and 2-b sum output of a 2-b ripple carry adder. The ripple carry adder has inputs $x_1$, $x_0$, $y_1$, and $y_0$, where $x_1$ and $y_1$ are the most significant bits of inputs $x$ and $y$. Notice that in this example the rows are indexed by $x_1$ and $y_1$, in contrast to the previous example, where the common variables were used to index the rows. In this case, the idea is to remove the unique variables from each functions and be left only with subfunctions that have the same support.

By applying the FLDS synthesis procedure from Section III, we find basis functions $\mathbf{G_1} = x_1 y_1$, $\mathbf{G_2} = x_1 + y_1$, and $\mathbf{G_3} = x_1 \odot y_1$. We can optimize basis and selector functions with the procedure from Section V and find that we can replace $\mathbf{G_2}$ with $\mathbf{G_1} \oplus \mathbf{G_2}$, forming $\mathbf{G'_2} = x_1 \oplus y_1$, and replace $\mathbf{G_3}$ with $\mathbf{G'_2} \oplus \mathbf{G_3}$, forming $\mathbf{G'_3} = \mathbf{1}$. Using these basis functions, we can synthesize $C_{\text{out}}$ and $S_1$ as

$$C_{\text{out}} = x_1 y_1 \oplus [(x_1 \oplus y_1) x_0 y_0]$$
$$S_1 = (x_1 \oplus y_1) \oplus x_0 y_0.$$

If we rename the selector function $x_0 y_0$ as $C_{\text{in}}$, then the circuit we obtain is a familiar full adder. The only step left to complete is to synthesize $C_{\text{in}}$ and $S_0$, which will synthesize as a familiar half adder. This example can be extended to an n-bit ripple carry adder.

### C. Multioutput Synthesis Algorithm

Examples in Section VI-B show that it is important to carefully consider which variables are assigned to the rows of the decomposition matrix when applying the FLDS algorithm.

```
Multi_Output_Synthesis(function_set, k) {
    all_vars = {x_i | (∃j)(x_i ∈ sup(function_set(j)))}
    common_vars = {x_i | (∀j)(x_i ∈ sup(function_set(j)))}
    if ||common_vars|| == ||all_vars|| then
        Assign common_vars to rows and decompose
    else
        L = function_set
        while (||L||≥1) {
            For all {f_i ∈ L | sup(f_i) ≤ k} do
                L=L-{f_i}, Synthesize f_i into a k-LUT
            For i=1 to ||L|| do Q_i={f ∈ L | x_i ∈ sup(f)}
            S = {x_i ∈ all_vars | (∀j)(||Q_i|| ≤ ||Q_j||)}
            F = {∪Q_i | x_i ∈ S}
            L = L - F
            If ||L|| == 0 then
                Decompose and synthesize functions in set F
            else {
                Decompose F into basis and selector functions
                Multi_Output_Synthesis(basis, k)
                L = L ∪ {selector functions}
            }
```

Fig. 11. Multioutput synthesis algorithm.

Based on these examples, we present a generalized multioutput synthesis algorithm.

The algorithm initially considers a decomposition for multioutput functions by using least common variables of a set of functions for indexing the rows of the truth table. By doing so, we can reduce the original set of functions to a set that depends on a common set of variables. Once the set of functions has been reduced to one where their support is common to all functions, we can then use the shared variables to index the rows of a matrix. The algorithm is shown in Fig. 11.

The first step (lines 2–3) in the algorithm determine if the functions to be synthesized share all variables, or just a subset. If all variables are shared, then we can perform a decomposition where the subset of shared variables are used to index the truth table rows (line 5). The decomposition with respect to shared variables will try to do a balanced decomposition so long as the number of basis functions is low in an effort to reduce logic circuit depth. If the number of basis functions becomes large, the number of variables indexing the rows will be reduced to decrease the number of basis functions.

The second step of the algorithm is to perform a decomposition in the case where some variables are shared by all functions and some are not (lines 7–22). We create a list $\mathbf{L}$ of these functions and perform the next set of steps while the list is not empty. The list will shrink and grow as we synthesize functions, either because of the decomposition, or because the functions with support size less than $k$ will be synthesized into $k$-LUT and removed from the list (lines 9–10).

The next set of steps focuses on finding a subset of variables $\mathbf{S}$ that are used by the fewest number of functions in the set $\mathbf{L}$ (lines 11–12). We take these functions out of the set $\mathbf{L}$ (lines 13–14). If the set $\mathbf{L}$ is empty, then all functions in $\mathbf{F}$ have the same support. We therefore run the variable partitioning algorithm, decompose and synthesize these functions as shown in Section VI-B (line 16). Otherwise, we perform a decomposition

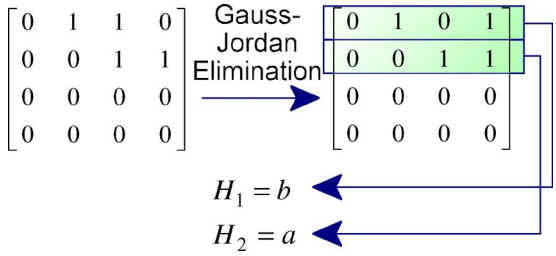


Fig. 12. Gauss–Jordan Elimination applied to Example 1.

where rows are indexed by variables in set $\mathbf{S}$ (lines 18–20). The decomposition proceeds as described in Section VI-B, but is limited to the functions in set $\mathbf{F}$. The decomposition (line 18) returns a set of basis and selector functions. The basis functions are synthesized separately by recursively calling the multioutput synthesis routine (line 19). The selector functions are placed back into set $\mathbf{L}$ (line 20) as they may still have variables in common with functions in set $\mathbf{L}$.

The motivation behind the aforementioned algorithm is to decompose a set of logic functions in such a way as to identify the subexpression that share the same support. This forces the algorithm to first find the subset of functions that heavily depend on one another, and then decompose them in as area efficient manner as it is able.

## VII. Performance Considerations

The previous sections demonstrated how FLDS can synthesize logic functions using XOR relationship between logic subfunctions. Its usefulness in CAD tools, however, depends significantly on how fast the synthesis process can be performed. Storing a truth table in memory will become problematic past 20 variables as it will take time to fill the table with 0s and 1s, and the table itself will begin to occupy increasing amount of memory. In this section, we discuss the issue of performance and show how the synthesis process can be accelerated by using BDDs and Gauss–Jordan Elimination.

BDDs introduced by Bryant [13] are an efficient structure for storing and processing logic functions. In FLDS, we have chosen to use BDDs to encode the truth table of a function, specifically by encoding each row of the truth table as a BDD. While, in extreme cases, this may not work well, in most practical cases, the BDD for each row will be small and easy to process. We used the Cudd BDD package [17] in this paper.

The second method of accelerating the synthesis process is the use of Gauss–Jordan Elimination in place of Gaussian Elimination. Recall from Example 1 in Section III that the process of Gaussian Elimination was used to determine the basis functions. After finding basis functions, we proceeded to determine the logic expression for the selector function of each basis vector. These two steps can be replaced by simply performing Gauss–Jordan Elimination.

In Fig. 12, the matrix on the left is the matrix in the row-echelon form from Example 1. By applying the Gauss–Jordan Elimination, the matrix is further reduced to a reduced row-echelon form shown on the right. In this form, the columns containing leading 1s have a single nonzero entry. By closer

inspection, we can see that the first row in the reduced row-echelon form matrix corresponds to the truth table of the selector function for the first basis vector. Similarly, the second row of this matrix corresponds to the selector function for the second basis vector.

Notice that in Section III, we found the equations for $H_1$ and $H_2$ by solving linear equations for each column of the truth table. Gauss–Jordan Elimination solves this problem for us. This is a property of the Gauss–Jordan Elimination [16], and while it is useful in solving linear equation, it is shown here that it is also important in the context of logic synthesis as the columns of the matrix are related by variables used to index them. Hence, the selector functions can be computed rapidly.

## VIII. FPGA Considerations

In the previous sections, synthesis and processing time issues were addressed by using algebraic manipulation. The above need to be augmented by considering the target platform for synthesis, which in our case are FPGAs.

In particular, a key parameter is the size of a LUT, $k$, used by an FPGA. The idea here is that the decomposition process should take into consideration the amount of space basis and selector functions will take. In our implementation, any function of $k$ or fewer variables is synthesized into a single LUT. This presents an opportunity for area recovery as it is possible for the same function to be synthesized multiple times. This can either happen by synthesizing two functions in tandem, or one after another. As such, it is important to recognize repeated functionality and reuse it. For this purpose, we use a hash table of logic functions of $k$ and fewer inputs. Every time a LUT is synthesized, it is put in a hash table, using the set of input variables $key$ entries. If another LUT is to be created, we first check if the LUT function, or its complement, has already been synthesized. If so, rather than creating a new LUT, a wire or an inverter connection is added to the logic graph thereby saving area.

Another consideration is that modern FPGAs contain more complex logic than simple LUTs and FFs. In fact, both Altera and Xilinx FPGAs contain carry chains to implement fast ripple carry adders. Because these adders contain XOR gates outside of the LUT network, it is possible to utilize these XOR gates to further reduce the area taken by a logic function. This, however, is a topic for future work.

## IX. Related Work

FLDS resembles tabular methods based on the work of Ashenhurst and Curtis [1]–[3]. However, there are key differences between these methods and FLDS.

The tabular method looks for column multiplicity in a truth table, hoping to exploit it in an effort to reduce circuit area. In the example in Fig. 1, we found four distinct columns using the tabular method, but by inspection, we can see that FLDS finds three basis functions for this truth table. A second difference between the tabular method and FLDS was illustrated in Example 3 in Section IV. Specifically, we showed that during the synthesis, FLDS can utilize columns that do not actually appear in the truth table to reduce circuit area.

| $cd$ \ $ab$ | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | 0 | 0 | 1 | 1 |
| 01 | 0 | 1 | 1 | 0 |
| 10 | 1 | 0 | 0 | 1 |
| 11 | 1 | 1 | 0 | 0 |

Fig. 13. XOR gate replacement example.

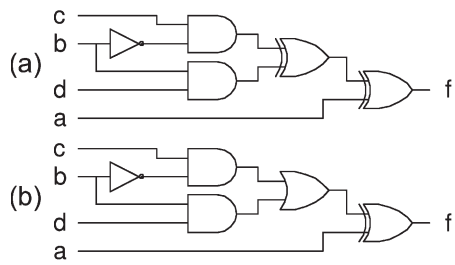


Fig. 14. Example of XOR gate replacement.

In a special case, FLDS simplifies to the tabular method. An example of such a case is the synthesis of a wide AND gate (e.g., $f = abcde$). In this particular case, we find that the number of distinct columns and the number of basis vectors are the same. In more general terms, FLDS reduces to the tabular method if one of the following is true:

1) The product of all pairs of selector functions is zero.
2) The product of all pairs of basis functions is zero.

The aforementioned relationship between FLDS and the tabular method can be exploited in non-FPGA context, where it may be advantageous to replace XOR gates with OR gates. In FLDS, it is easy to determine if an XOR gate can be replaced by an OR gate. If a product of a pair of basis functions, or their respective selector functions, is equal to zero, then the XOR gate used to sum the basis–selector products for this pair of subfunctions can be replaced by an OR gate. An example is given below.

Fig. 13 shows a logic function with three basis functions. They are $\mathbf{G_1} = c$, $\mathbf{G_2} = d$, and $\mathbf{G_3} = \mathbf{1}$. We can see that $\mathbf{G_1}$ is used in columns $ab = 00$ and 10, $\mathbf{G_2}$ is used in columns $ab = 01$ and 11, while $\mathbf{G_3}$ is used in columns $ab = 10$ and 11. Using only two-input gates, this function can be realized, as shown in Fig. 14(a). Notice however that the selector functions for $\mathbf{G_1}$ and $\mathbf{G_2}$ are complements of one another, and thus, their product is equal to zero. It is therefore unnecessary to sum these two subfunctions using an XOR gate. An OR gate can be used instead to implement the same logic expression. This is shown in Fig. 14(b).

Positive and negative Davio expansions, useful for XOR decomposition, are also a special case of FLDS. To derive them using FLDS, assign a single variable $x$ to the bound set. This will result in two basis and selector functions, and the function $f$ can be synthesized as follows:

$$f = x f_x \oplus \overline{x} f_{\overline{x}}. \tag{3}$$

By replacing one of the basis functions with $x \oplus \overline{x} = 1$, we produce one of the following equations:

$$f = f_x \oplus \overline{x}(f_x \oplus f_{\overline{x}}) \tag{4}$$

$$f = (f_x \oplus f_{\overline{x}})x \oplus f_{\overline{x}}. \tag{5}$$

Equations (4) and (5) are the negative and the positive Davio Expansions, respectively.

The consequence of the ability to decompose a logic function using (3)–(5) using a cost function is the ability to synthesize exclusive OR sum of products expressions of various classes. In particular, if FLDS is set up to decompose a logic function one variable at a time, it will generate either a Pseudo Kronecker expression or Generalized Reed–Muller expression for a logic function [4]. The choice of the expression generated depends on the cost of implementing the basis and selector functions. If during the decomposition (3)–(5) are used, then a Pseudo Kronecker expression will be formed. However, if the expansion is carried out using only (4) and (5), then the resulting expression will be in a Generalized Reed–Muller form. The choice between the two forms depends on which option produces basis and selector functions of lower cost.

Another work closely related to FLDS is called *Factor* [20]. *Factor* is an approach to logic decomposition based on Single Value Decomposition (SVD). FLDS differs from factor in several ways.

The first difference between the two approaches is the use of basis and selector optimization algorithm in FLDS. This algorithm, as shown in Section V, provides the means to further reduce the cost of logic implementation, even if the initial set of basis functions is minimal. This is not the case with factor. In fact, *Factor* requires not only a good variable partitioning, but also a good variable ordering within each partition to achieve a good result, because it does not use an algorithm similar to the basis and selector optimization.

The second difference is that the work in [20] can only perform a balanced decomposition. As we have shown earlier, FLDS is not limited to performing a balanced decomposition. In that respect, this paper generalizes the approach in [20] to cover both balanced and unbalanced decompositions.

The third difference is that the decomposition in [20] uses an algorithm of the same runtime as Gaussian Elimination several times to perform the decomposition of a single function. The goal there is to find logic expressions for each subfunction. In FLDS, we use Gauss–Jordan Elimination to achieve the same goal faster. The advantage here is that Gauss–Jordan Elimination requires the use of row operations only. This fact is used to set up data structures in an efficient manner, allowing for a fast processing time.

The fourth difference is in how FLDS and [20] handle multioutput logic functions. In [20], the decomposition can result in a linear dependence of logic subfunctions, because the algorithm picks a variable partition that minimizes the rank of each individual logic function, but it does not remove the linear dependencies between the generated logic subfunctions. Thus, a subset of functions generated by the decomposition in [20] for two functions $f$ and $g$ may be such that the subfunctions of $f$ are linearly dependent on subfunctions of $g$.

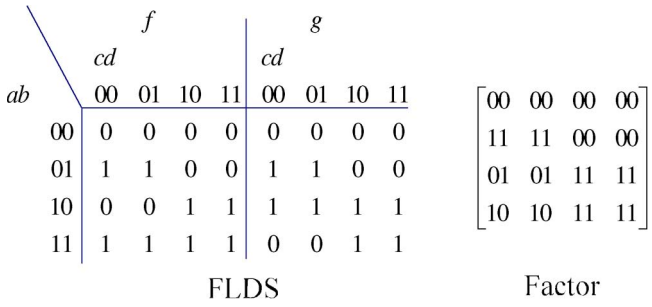


Fig. 15. Example of the differences between FLDS and *Factor.*

This is not the case with FLDS. FLDS addresses both linear dependence and subfunction minimization of multiple output functions. First, we generate linearly independent set of subfunctions for all selected functions. Any linear dependencies between the subfunctions are removed by using Gaussian Elimination. Second, a basis and selector function optimization algorithm, which runs in $O(n^2)$ for $n$ basis vectors, does a good job at finding lower cost subfunctions. As demonstrated in Section V, this helps us reduce logic area. To illustrate these differences, consider the following example.

Fig. 15 shows two functions $f$ and $g$ as they are represented by FLDS and *factor* [20]. During synthesis, the algorithm presented by Hwang *et al.* [20] uses the setup shown on the right-hand side of Fig. 15 to decompose the logic functions $f$ and $g$ into subfunctions $\mathbf{G_1} = a$ and $\mathbf{G_2} = b$ for $f$ and $\mathbf{G_3} = a \oplus b$ and $\mathbf{G_1} = a$ for $g$. However, the algorithm does not exploit the relationship $\mathbf{G_3} = \mathbf{G_1} \oplus \mathbf{G_2}$ to reduce logic area and, as a consequence, synthesizes functions $f$ and $g$ using a total of six two-input gates. In contrast, FLDS optimizes $\mathbf{G_3}$ away because of a linear dependence on $\mathbf{G_1}$ and $\mathbf{G_2}$. FLDS decomposes both $f$ and $g$ with respect to $\mathbf{G_1}$ and $\mathbf{G_2}$, synthesizing the circuit using only four two-input gates.

Finally, our approach uses a fast variable partitioning approach, whereas the work in [20] does not. A follow up work in [22] uses cubical notation to perform analysis of a logic function to better choose variable partitioning for the algorithm in [20]. However, the number of cubes can be quite large and the approach is only viable for small logic functions.

## X. Experimental Results

In this section, we present area, depth, and compilation time results and compare them to previously published works. The following sections contain the experimental methodology, summary of results, and discussion.

### A. Methodology

In this paper, the area results obtained from FLDS were compared to two academic synthesis systems: ABC [18] optimizes a circuit using structural transformations, and BDS-PGA 2.0 [10] performs Boolean optimization by utilizing BDDs.

To obtain results using ABC, each circuit was first converted to an AND-Inverter Graph (AIG) using the *strash* command. The circuit was then optimized using the *resyn2* script that operates on AIGs to reduce the circuit size, and the resulting

TABLE I
AREA RESULTS COMPARISON TABLE FOR XOR-BASED CIRCUITS

| Name | ABC | | | BDS-PGA-2.0 | | | | | | FLDS | | | | FLDS vs. ABC | | FLDS vs. BDS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LUTs | Depth | Time (s) | LUTs | Depth | Time (s) | X | H | S | LUTs | Depth | Time (s) | Cone Size | Area | Depth | Area | Depth |
| 5xp1 | 40 | 4 | 0.08 | 25 | 4 | 0.07 | X | | | 22 | 4 | 0.02 | 8 | -45.00 | 0.00 | -12.00 | 0.00 |
| 9sym | 114 | 6 | 0.08 | 58 | 6 | 0.54 | | X | | 13 | 4 | 0.031 | 20 | -88.60 | -33.33 | -77.59 | -33.33 |
| 9symml | 85 | 5 | 0.08 | 19 | 6 | 0.17 | X | | | 13 | 4 | 0.051 | 12 | -84.71 | -20.00 | -31.58 | -33.33 |
| alu2 | 150 | 9 | 0.09 | 208 | 12 | 0.82 | | X | X | 102 | 7 | 0.082 | 24 | -32.00 | -22.22 | -50.96 | -41.67 |
| C1355 | 75 | 4 | 0.14 | 77 | 6 | 3.87 | X | X | X | 80 | 6 | 0.036 | 8 | 6.25 | 33.33 | 3.75 | 0.00 |
| C1908 | 107 | 8 | 0.11 | 122 | 9 | 0.91 | | X | X | 167 | 13 | 0.077 | 8 | 35.93 | 38.46 | 26.95 | 30.77 |
| C3540 | 363 | 12 | 0.22 | 418 | 14 | 2.99 | | | | 613 | 17 | 0.236 | 8 | 40.78 | 29.41 | 31.81 | 17.65 |
| C499 | 77 | 5 | 0.13 | 80 | 4 | 3.42 | | | | 77 | 6 | 0.035 | 8 | 0.00 | 16.67 | -3.75 | 33.33 |
| C880 | 112 | 9 | 0.14 | 123 | 9 | 1.21 | X | X | X | 167 | 11 | 0.067 | 8 | 32.93 | 18.18 | 26.35 | 18.18 |
| cordic | 304 | 8 | 0.16 | 161 | 18 | 1501 | | | | 24 | 5 | 9.735 | 24 | -92.11 | -37.50 | -85.09 | -72.22 |
| count | 42 | 5 | 0.06 | 39 | 5 | 0.12 | | X | X | 52 | 5 | 0.036 | 8 | 19.23 | 0.00 | 25.00 | 0.00 |
| dalu | 255 | 6 | 0.16 | 326 | 9 | 2.34 | | X | X | 363 | 9 | 0.128 | 8 | 29.75 | 33.33 | 10.19 | 0.00 |
| des | 1340 | 6 | 0.67 | 1338 | 7 | 6.03 | | | | 1155 | 8 | 5.456 | 16 | -13.81 | 25.00 | -13.68 | 12.50 |
| f51m | 41 | 4 | 0.06 | 32 | 5 | 0.12 | | | | 18 | 4 | 0 | 12 | -56.10 | 0.00 | -43.75 | -20.00 |
| inc | 51 | 4 | 0.06 | 45 | 4 | 0.1 | | | | 51 | 4 | 0.035 | 8 | 0.00 | 0.00 | 11.76 | 0.00 |
| my-adder | 32 | 16 | 0.08 | 33 | 16 | 0.89 | X | X | X | 35 | 16 | 0.051 | 24 | 8.57 | 0.00 | 5.71 | 0.00 |
| rd53 | 10 | 3 | 0.05 | 12 | 3 | 0 | | X | X | 7 | 2 | 0 | 12 | -30.00 | -33.33 | -41.67 | -33.33 |
| rd73 | 63 | 4 | 0.09 | 15 | 4 | 0.09 | | X | | 11 | 3 | 0.015 | 12 | -82.54 | -25.00 | -26.67 | -25.00 |
| rd84 | 106 | 6 | 0.09 | 25 | 5 | 0.4 | | | | 15 | 3 | 0.031 | 8 | -85.85 | -50.00 | -40.00 | -40.00 |
| sqrt8 | 26 | 4 | 0.06 | 22 | 4 | 0.06 | | | | 12 | 3 | 0 | 12 | -53.85 | -25.00 | -45.45 | -25.00 |
| sqrt8ml | 20 | 5 | 0.08 | 26 | 6 | 0.07 | | X | X | 12 | 3 | 0.02 | 16 | -40.00 | -40.00 | -53.85 | -50.00 |
| squar5 | 18 | 3 | 0.06 | 18 | 3 | 0.03 | | X | X | 15 | 2 | 0.015 | 24 | -16.67 | -33.33 | -16.67 | -33.33 |
| t481 | 135 | 6 | 0.11 | 70 | 6 | 22.81 | | X | X | 5 | 2 | 0.078 | 24 | -96.30 | -66.67 | -92.86 | -66.67 |
| xor5 | 2 | 2 | 0.05 | 2 | 2 | 0 | | | | 2 | 2 | 0.015 | 8 | 0.00 | 0.00 | 0.00 | 0.00 |
| z4ml | 7 | 3 | 0.05 | 6 | 3 | 0.01 | | X | X | 8 | 3 | 0.02 | 8 | 12.50 | 0.00 | 25.00 | 0.00 |
| **Total/Average** | | | 2.96 | | | 1548 | | | | | | 16.27 | | -25.26 | -7.68 | -18.76 | -14.46 |
| **Ratio** | | | 1 | | | 523 | | | | | | 5.497 | | | | | |

logic was mapped into 4-LUTs using ABC's *fpga* command. The time measurement included the *strash, resyn2*, and *fpga* steps.

To obtain results using BDS-PGA 2.0, a few more steps had to be taken, because the BDS-PGA 2.0 system contains various flags that guide its synthesis process. The flags of particular interest are:

1) xhardcore ($\mathbf{X}$)—a flag to enable x-dominator-based XOR decompositions;
2) sharing ($\mathbf{S}$)—a flag used to perform sharing extraction prior to decomposition;
3) heuristic ($\mathbf{H}$)—a flag used to enable a heuristic variable ordering algorithm.

Depending on the settings of these flags, a circuit synthesized with BDS-PGA 2.0 may have a different final area. Thus, we performed a sweep across all combinations of these three parameters and recorded the best result, based first on area, then depth, and, finally, on processing time. The resulting network was then mapped into 4-LUTs using the ABC system.

FLDS results were obtained by running the FLDS algorithm on each circuit, varying only the cone size parameter. This parameter is used within FLDS in the preprocessing step to create cones for synthesis. It specifies the maximum number of inputs to which a cone can be grown. To obtain our result the cone size was set to 8, 12, 16, 20, and 24 inputs. The resulting circuit was mapped into 4-LUTs using the ABC system. ABC was then used to verify the logical equivalence of each circuit synthesized with FLDS. The processing time for our method includes preprocessing to generate the logic cones of specified size as well as the decomposition and synthesis time used to restructure each logic circuit.

### B. Results

Table I presents the results obtained for 25 XOR-based MCNC circuits [25]. The name of each circuit is given in the first column. The following three columns list the area (LUT-wise), depth, and processing time results when using ABC. Columns 5 through 7 list the area, depth, and processing time results for BDS-PGA 2.0. The setting used to obtain the result for BDS-PGA 2.0 is given in columns 8 through 10, which corresponds to a particular BDS-PGA 2.0 flag ($\mathbf{X}$, $\mathbf{H}$, or $\mathbf{S}$) as described in the previous section. For a given circuit, an "X" appears in these columns if the flag was turned on to generate the given result.

In columns 11 through 13, area, depth, and processing time results obtained by our FLDS system are shown. For each circuit, column 14 states the cone size used to generate cones for logic synthesis. The final four columns compare results obtained with FLDS to ABC (columns 15 and 16) and to BDS-PGA 2.0 (columns 17 and 18). The results for non-XOR-based circuits are presented in a similar format in Table II.

To compare the results obtained using various tools, we used the following equation:

$$\%\text{difference} = 100\frac{(\text{final} - \text{initial})}{MAX(\text{initial}, \text{final})}. \tag{6}$$

This equation ensures that both gains and losses presented in the tables of results are weighted equally.

Overall, our results show that XOR-based logic functions can be significantly reduced in size and logic depth, 18.8% and 14.5%, respectively, as compared to logic minimization tools like BDS-PGA 2.0. The cost of this optimization is the increase in cost for non-XOR-based logic circuit, both in area

TABLE II
AREA RESULTS FOR NON-XOR-BASED LOGIC CIRCUITS

| Name | ABC | | | BDS-PGA-2.0 | | | | | | FLDS | | | | FLDS vs. ABC | | FLDS vs. BDS | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LUTs | Depth | Time (s) | LUTs | Depth | Time (s) | X | H | S | LUTs | Depth | Time (s) | Cone Size | Area | Depth | Area | Depth |
| apex3 | 706 | 6 | 0.31 | 808 | 13 | 17.12 | X | X | X | 843 | 8 | 0.971 | 8 | 16.25 | 25.00 | 4.15 | -38.46 |
| apex6 | 252 | 5 | 0.11 | 252 | 6 | 0.64 | | | | 337 | 7 | 0.593 | 24 | 25.22 | 28.57 | 25.22 | 14.29 |
| apex7 | 73 | 5 | 0.08 | 82 | 6 | 0.18 | | X | X | 103 | 9 | 0.092 | 24 | 29.13 | 44.44 | 20.39 | 33.33 |
| b1 | 4 | 1 | 0.03 | 4 | 1 | 0 | | | | 4 | 1 | 0.02 | 20 | 0.00 | 0.00 | 0.00 | 0.00 |
| b12 | 29 | 3 | 0.06 | 34 | 4 | 0.04 | | | | 36 | 4 | 0.02 | 12 | 19.44 | 25.00 | 5.56 | 0.00 |
| b9 | 38 | 3 | 0.08 | 40 | 3 | 0.04 | | X | | 57 | 4 | 0.035 | 8 | 33.33 | 25.00 | 29.82 | 25.00 |
| bw | 105 | 4 | 0.08 | 79 | 5 | 0.29 | X | X | X | 68 | 4 | 0.035 | 12 | -35.24 | 0.00 | -13.92 | -20.00 |
| C17 | 2 | 1 | 0.05 | 2 | 1 | 0 | X | X | | 2 | 1 | 0.02 | 12 | 0.00 | 0.00 | 0.00 | 0.00 |
| C432 | 63 | 11 | 0.06 | 166 | 16 | 1.33 | | | | 123 | 15 | 0.123 | 12 | 48.78 | 26.67 | -25.90 | -6.25 |
| c8 | 37 | 3 | 0.06 | 37 | 3 | 0.06 | | X | | 38 | 4 | 0.015 | 8 | 2.63 | 25.00 | 2.63 | 25.00 |
| cc | 25 | 2 | 0.05 | 24 | 2 | 0.01 | X | | | 26 | 3 | 0.0 | 20 | 3.85 | 33.33 | 7.69 | 33.33 |
| cht | 38 | 2 | 0.05 | 41 | 2 | 0.13 | X | X | X | 48 | 3 | 0.0 | 24 | 20.83 | 33.33 | 14.58 | 33.33 |
| clip | 102 | 5 | 0.09 | 52 | 5 | 0.29 | | | | 37 | 4 | 0.082 | 12 | -63.73 | -20.00 | -28.85 | -20.00 |
| cm138a | 10 | 2 | 0.06 | 10 | 2 | 0.01 | | X | | 10 | 2 | 0.015 | 8 | 0.00 | 0.00 | 0.00 | 0.00 |
| cm150a | 13 | 4 | 0.05 | 13 | 4 | 0.03 | | | | 25 | 6 | 4.39 | 24 | 48.00 | 33.33 | 48.00 | 33.33 |
| cm151a | 7 | 3 | 0.03 | 8 | 3 | 0.01 | | X | | 11 | 4 | 0.02 | 8 | 36.36 | 25.00 | 27.27 | 25.00 |
| cm152a | 6 | 3 | 0.03 | 6 | 3 | 0 | | X | | 6 | 3 | 0.035 | 24 | 0.00 | 0.00 | 0.00 | 0.00 |
| cm162a | 12 | 3 | 0.05 | 18 | 3 | 0.01 | | X | | 18 | 4 | 0.02 | 8 | 33.33 | 25.00 | 0.00 | 25.00 |
| cm163a | 11 | 3 | 0.05 | 11 | 3 | 0.01 | | X | | 13 | 4 | 0.0 | 8 | 15.38 | 25.00 | 15.38 | 25.00 |
| cm42a | 10 | 1 | 0.05 | 10 | 1 | 0.01 | X | X | X | 10 | 1 | 0.02 | 8 | 0.00 | 0.00 | 0.00 | 0.00 |
| cm82a | 4 | 2 | 0.06 | 4 | 2 | 0 | | X | X | 5 | 2 | 0.0 | 20 | 20.00 | 0.00 | 20.00 | 0.00 |
| cm85a | 12 | 3 | 0.05 | 11 | 3 | 0.05 | X | X | | 12 | 4 | 0.015 | 12 | 0.00 | 25.00 | 8.33 | 25.00 |
| cmb | 16 | 3 | 0.05 | 19 | 4 | 0.04 | X | X | | 11 | 2 | 0.015 | 20 | -31.25 | -33.33 | -42.11 | -50.00 |
| comp | 30 | 4 | 0.06 | 33 | 6 | 2.51 | X | X | | 30 | 5 | 0.895 | 20 | 0.00 | 20.00 | -9.09 | -16.67 |
| con1 | 6 | 2 | 0.03 | 5 | 2 | 0 | X | X | | 7 | 2 | 0.0 | 16 | 14.29 | 0.00 | 28.57 | 0.00 |
| cu | 17 | 3 | 0.05 | 20 | 3 | 0.07 | X | X | X | 19 | 3 | 0.015 | 12 | 10.53 | 0.00 | -5.00 | 0.00 |
| decod | 20 | 2 | 0.03 | 20 | 2 | 0.02 | | X | | 24 | 2 | 0.015 | 20 | 16.67 | 0.00 | 16.67 | 0.00 |
| duke2 | 192 | 5 | 0.11 | 227 | 6 | 0.35 | X | | | 268 | 6 | 0.077 | 8 | 28.36 | 16.67 | 15.30 | 0.00 |
| e64 | 226 | 4 | 0.08 | 233 | 5 | 0.62 | | X | X | 280 | 5 | 0.067 | 16 | 19.29 | 20.00 | 16.79 | 0.00 |
| ex5p | 882 | 7 | 0.41 | 871 | 8 | 14.04 | | X | X | 173 | 4 | 0.208 | 16 | -80.39 | -42.86 | -80.14 | -50.00 |
| example2 | 121 | 4 | 0.08 | 125 | 5 | 0.18 | | | | 140 | 6 | 0.036 | 8 | 13.57 | 33.33 | 10.71 | 16.67 |
| frg1 | 39 | 6 | 0.06 | 55 | 9 | 0.29 | | X | | 70 | 8 | 0.035 | 8 | 44.29 | 25.00 | 21.43 | -11.11 |
| frg2 | 279 | 6 | 0.16 | 265 | 6 | 0.96 | X | X | X | 355 | 9 | 0.097 | 8 | 21.41 | 33.33 | 25.35 | 33.33 |
| i1 | 19 | 3 | 0.05 | 17 | 3 | 0.01 | | X | | 19 | 3 | 0.0 | 16 | 0.00 | 0.00 | 10.53 | 0.00 |
| i2 | 73 | 5 | 0.05 | 86 | 6 | 0.74 | | | | 76 | 7 | 0.035 | 8 | 3.95 | 28.57 | -11.63 | 14.29 |
| i3 | 46 | 3 | 0.05 | 46 | 3 | 0.28 | | | | 46 | 3 | 0.067 | 12 | 0.00 | 0.00 | 0.00 | 0.00 |
| i4 | 82 | 5 | 0.08 | 85 | 5 | 0.87 | X | | | 97 | 6 | 0.083 | 8 | 15.46 | 16.67 | 12.37 | 16.67 |
| i5 | 101 | 6 | 0.08 | 101 | 6 | 0.15 | | X | X | 115 | 6 | 0.015 | 16 | 12.17 | 0.00 | 12.17 | 0.00 |
| i6 | 144 | 2 | 0.08 | 121 | 2 | 0.21 | X | X | | 108 | 2 | 0.02 | 12 | -25.00 | 0.00 | -10.74 | 0.00 |
| i7 | 181 | 2 | 0.08 | 167 | 2 | 0.34 | X | | | 167 | 3 | 0.067 | 16 | -7.73 | 33.33 | 0.00 | 33.33 |
| i8 | 447 | 5 | 0.2 | 414 | 7 | 2.33 | X | X | X | 450 | 9 | 0.404 | 12 | 0.67 | 44.44 | 8.00 | 22.22 |
| i9 | 262 | 5 | 0.13 | 211 | 5 | 0.86 | | X | X | 300 | 8 | 0.097 | 8 | 12.67 | 37.50 | 29.67 | 37.50 |
| lal | 32 | 3 | 0.05 | 30 | 3 | 0.04 | | | | 33 | 3 | 0.036 | 8 | 3.03 | 0.00 | 9.09 | 0.00 |
| ldd | 35 | 3 | 0.05 | 34 | 3 | 0.09 | X | X | X | 41 | 4 | 0.035 | 8 | 14.63 | 25.00 | 17.07 | 25.00 |
| majority | 3 | 2 | 0.03 | 3 | 2 | 0 | | | | 3 | 2 | 0 | 20 | 0.00 | 0.00 | 0.00 | 0.00 |
| misex1 | 21 | 3 | 0.05 | 20 | 3 | 0.01 | X | X | | 19 | 3 | 0.02 | 8 | -9.52 | 0.00 | -5.00 | 0.00 |
| misex2 | 40 | 3 | 0.05 | 43 | 4 | 0.07 | X | X | X | 46 | 4 | 0.015 | 16 | 13.04 | 25.00 | 6.52 | 0.00 |
| misex3c | 440 | 6 | 0.19 | 570 | 12 | 34.49 | | | | 291 | 7 | 5.816 | 24 | -33.86 | 14.29 | -48.95 | -41.67 |
| mux | 13 | 4 | 0.06 | 13 | 4 | 0.04 | | | | 20 | 5 | 0.036 | 8 | 35.00 | 20.00 | 35.00 | 20.00 |
| pair | 534 | 7 | 0.22 | 529 | 9 | 3.76 | X | X | X | 736 | 9 | 0.268 | 8 | 27.45 | 22.22 | 28.13 | 0.00 |
| parity | 5 | 2 | 0.06 | 5 | 2 | 0.01 | | X | | 5 | 2 | 0 | 8 | 0.00 | 0.00 | 0.00 | 0.00 |
| pcle | 21 | 3 | 0.05 | 20 | 3 | 0.03 | | | | 23 | 4 | 0.035 | 8 | 8.70 | 25.00 | 13.04 | 25.00 |
| pcler8 | 31 | 4 | 0.06 | 32 | 4 | 0.06 | X | X | X | 34 | 4 | 0.02 | 8 | 8.82 | 0.00 | 5.88 | 0.00 |
| pdc | 3302 | 8 | 1.56 | 3966 | 13 | 18.74 | X | | | 693 | 9 | 44.14 | 20 | -79.01 | 11.11 | -82.53 | -30.77 |
| pm1 | 19 | 2 | 0.06 | 18 | 2 | 0.04 | | X | X | 20 | 3 | 0 | 24 | 5.00 | 33.33 | 10.00 | 33.33 |
| rot | 246 | 8 | 0.11 | 272 | 10 | 17.59 | X | X | X | 328 | 11 | 0.097 | 8 | 25.00 | 27.27 | 17.07 | 9.09 |
| sao2 | 82 | 4 | 0.08 | 96 | 8 | 0.35 | | X | | 32 | 3 | 0.076 | 12 | -60.98 | -25.33 | -66.67 | -62.5 |
| sct | 23 | 3 | 0.06 | 23 | 3 | 0.01 | X | X | | 24 | 4 | 0.02 | 8 | 4.17 | 25.00 | 4.17 | 25.00 |
| spla | 2634 | 8 | 1.45 | 334 | 8 | 22.44 | | | | 398 | 9 | 0.925 | 16 | -84.89 | 11.11 | 16.08 | 11.11 |
| table3 | 393 | 7 | 0.17 | 445 | 9 | 3.55 | X | X | X | 537 | 9 | 0.144 | 8 | 26.82 | 22.22 | 17.13 | 0.00 |
| table5 | 412 | 6 | 0.17 | 462 | 10 | 7.73 | X | X | X | 550 | 9 | 0.144 | 8 | 25.09 | 33.33 | 16.00 | -10.00 |
| tcon | 16 | 1 | 0.06 | 16 | 1 | 0 | | X | | 16 | 1 | 0.0 | 16 | 0.00 | 0.00 | 0.00 | 0.00 |
| term1 | 61 | 5 | 0.08 | 68 | 5 | 0.29 | X | X | X | 93 | 8 | 0.036 | 8 | 34.41 | 37.50 | 26.88 | 37.50 |
| too-lrg | 141 | 6 | 0.09 | 176 | 8 | 0.54 | X | | | 226 | 10 | 0.051 | 8 | 37.61 | 40.00 | 22.12 | 20.00 |
| ttt2 | 49 | 4 | 0.06 | 56 | 5 | 0.14 | | | | 53 | 5 | 0.035 | 12 | 7.55 | 20.00 | -5.36 | 0.00 |
| unreg | 48 | 2 | 0.06 | 37 | 2 | 0.06 | | X | | 40 | 3 | 0.016 | 8 | -16.67 | 33.33 | 7.50 | 33.33 |
| vda | 260 | 5 | 0.13 | 273 | 7 | 1.41 | | X | X | 364 | 8 | 0.3 | 8 | 28.57 | 37.50 | 25.00 | 12.50 |
| vg2 | 33 | 4 | 0.06 | 42 | 5 | 0.15 | | X | X | 68 | 9 | 0.051 | 24 | 51.47 | 55.56 | 38.24 | 44.44 |
| x1 | 120 | 4 | 0.08 | 128 | 5 | 0.23 | | | | 154 | 5 | 0.036 | 8 | 22.08 | 20.00 | 16.88 | 0.00 |
| x2 | 19 | 3 | 0.03 | 17 | 4 | 0.03 | | | | 18 | 4 | 0 | 12 | -5.26 | 25.00 | 5.56 | 0.00 |
| x3 | 231 | 5 | 0.11 | 248 | 5 | 0.91 | | X | X | 294 | 7 | 0.143 | 12 | 21.43 | 28.57 | 15.65 | 28.57 |
| x4 | 123 | 4 | 0.08 | 158 | 4 | 0.24 | | | | 143 | 5 | 0.051 | 20 | 13.99 | 20.00 | -9.49 | 20.00 |
| **Total/Average** | | | 8.83 | | | 158.18 | | | | | | 61.27 | | 6.20 | 16.48 | 4.78 | 6.19 |
| **Ratio** | | | 1 | | | 17.9 | | | | | | 6.94 | | | | | |

and in depth by 4.8% and 6.2%, respectively. As compared to structural optimization techniques, such as those used in ABC, FLDS synthesizes XOR-based circuits using 25.3% less area and reduces logic depth by 7.7%. For non-XOR-based logic circuits, ABC produces circuits with 6.2% lower area results and 16.5% lower depth on average than FLDS.

### C. Discussion of Individual Circuits

As shown in Table I, significant area savings are found using FLDS. The largest area savings for XOR-based circuits were observed for *9sym, cordic, sqrt8ml*, and *t481*.

*9sym* is a circuit with a single nine input logic function. The circuit itself consists of a number of XOR gates. Interestingly enough, *9symml* represents the same function, except with a different initial representation, was synthesized much better than *9sym*. In *9sym*, BDS-PGA 2.0 found the circuit to consist of nine logic functions and hence was unable to find a compact implementation for it, although it faired 30% better than ABC. However, for *9symml*, BDS-PGA found it to be a single output nine-input function and created a single BDD for it. Despite finding a single cone of logic for *9symml*, BDS-PGA produced a 30% worse result than FLDS. This example shows that functionally linear decomposition and x-dominator-based decompositions are quite different. While both strive to utilize XOR gates, searching for an x-dominator in a BDD clearly does not guarantee finding a better decomposition.

Another interesting circuit is *cordic*. It consists of two functions sharing 23 inputs. In this particular case, the results we obtained come from two sources: the ability to synthesize large functions and synthesizing multioutput functions. FLDS found the functions to be reasonably easy to synthesize using a balanced decomposition.

In contrast, neither ABC nor BDS-PGA found there to be only two cones of logic. Instead, many cones were created to represent this circuit. In the case of BDS-PGA, 32 logic functions were created. While in Table I, the synthesis flags for this circuit do not include x-dominator decompositions usage, the results with the same flag turned on were identical, except for the longer processing time.

*Sqrt8ml* is an arithmetic circuit that computes a square root of a binary number. It consists of four logic functions, with two, four, six, and eight inputs. The two functions that fit in a single LUT were synthesized separately, but the six and eight input functions were synthesized in tandem. The algorithm found five basis vectors shared by these two functions. In the worst case scenario, this would cause 15 LUTs to be created to implement these two functions, but due to our basis and selector function optimization routine, this was not the case. Specifically, the basis and selector function optimization led to the realization of the basis and selector functions using fewer variables. Thus, the mapping stage of the CAD flow could successfully map both functions into a total of ten LUTs. The result we obtained was nearly half the size of the one produced by ABC, and less than half of the area taken by the same circuit synthesized with BDS-PGA.

The *t481* circuit is a 16-input single-output logic function, which has a compact representation in the Fixed-Polarity Reed–Muller form. When synthesized correctly, it takes only five four-LUTs, which is the minimum for this circuit. In FLDS, the function was synthesized in 0.125 s, finding only two basis functions during balanced decomposition. In this particular case, the variable partitioning algorithm was successful in finding the correct assignment of variables to the bound and the free set, which allowed for minimum area synthesis of the logic function.

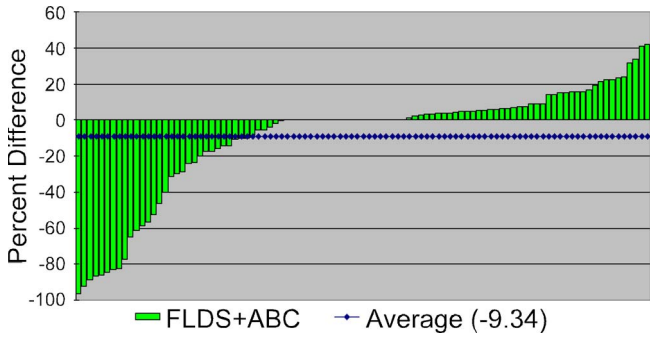


Fig. 16. Distribution of area savings compared to ABC.

The success of our method is not limited to XOR-based logic functions. Table II shows the results for non-XOR-based logic circuits from the MCNC benchmark set. The columns in Table II have the same meaning as in Table I. As shown in the table, numerous logic circuits that are considered not to be XOR-based have benefited significantly as well. Examples of such circuits are *ex5p, pdc, sao2*, and *ex1010*.[1] In these circuits, *pdc* and *ex1010* in particular, the success is attributed to synthesis of multioutput functions, rather than synthesizing each cone of logic separately. The *pdc* circuit did take FLDS 2.5 times longer to synthesize than when using BDS-PGA 2.0, but the achieved result is nearly six times smaller and with a 30% lower logic depth.

The results produced by FLDS can be further improved by applying existing logic synthesis techniques on top of FLDS. In this paper, we investigated the effect of applying the *resyn2* script of ABC on top of FLDS results to determine if it can further reduce logic area. We found that if ABC is used on top of FLDS and the results compared to those obtained using ABC alone for the same set of circuits, then we obtain a 24.2% smaller area and a 16.2% lower depth for XOR-based circuits. For non-XOR-based circuits area was 4.25% smaller with only a 1% depth increase. Averaged over both XOR-based and non-XOR-based circuits, FLDS together with ABC can reduce an average circuit size by 9.3% and reduce the depth by 3.3%, as compared to what ABC can do alone. The distribution of results is shown in Fig. 16.

## XI. Conclusion

The FLDS technique was found to be very effective in reducing area taken by logic functions heavily dependent on XOR gates. It achieved a 25% reduction in area as compared

[1]This circuit failed to synthesize with BDS-PGA 2.0 and, hence, is excluded from average/total computation. FLDS produced a result in 13.94 s, resulting in 1063 LUTs and a depth of seven. In contrast, ABC produced a result in 1.52 s containing 4094 LUTs with depth eight.

to ABC [18], as well as an 18.8% reduction in comparison to BDD-based approaches used in BDS-PGA 2.0 [10]. We also showed that FLDS provides a depth reduction of 7.7% and 14.48%, respectively, compared to these approaches. For non-XOR intensive logic circuits, the technique suffers only a minor area and depth penalty. For both of these groups of functions, FLDS performed synthesis rapidly.

The technique presented here can be further improved. In the short term, the most promising is the extension of this technique to cover nondisjoint decompositions. A challenge here will be to modify the variable partitioning technique to select the correct variables to overlap between the bound and the free sets.

Another extension to consider is to apply the proposed approach to multivalued logic decompositions. Since this approach is quite generic in terms of the underlying mathematical field used, it will be valid for any valid GF($m$). However, the addition, subtraction, and multiplication operations may prove more complex for $m > 2$, mostly because the final implementation is in terms of Boolean logic.

## ACKNOWLEDGMENT

The authors would like to thank Dr. D. Singh and Dr. V. Manohararajah from Altera Corporation for suggestions during the course of this paper. The authors would also like to thank Prof. Z. G. Vranesic and Prof. J. Zhu from the University of Toronto for their feedback throughout the course of this paper.

## REFERENCES

[1] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. Theory Switch*ing, Apr. 2–5, 1957, pp. 74–116.
[2] H. A. Curtis, *A New Approach to the Design of Switching Circuits*. Princeton, NJ: Van Nostrand, 1962.
[3] M. Perkowski and S. Grygiel, "A survey of literature on function decomposition," in "A Final Report for Summer Faculty Research Program, Wright Laboratory," Air Force Office Sci. Res., Bolling Air Force Base, Wright Lab., Washington, DC, Sep. 1994.
[4] T. Sasao, *Switching Theory for Logic Synthesis*. Norwell, MA: Kluwer, 1999.
[5] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral Techniques in Digital Logic*. London, U.K.: Academic, 1985.
[6] E. M. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," in *Proc. 30th DAC*, Jun. 1993, pp. 54–60.
[7] M. G. Karpovsky, "Harmonic analysis over finite commutative groups in linearization problems for systems of logical functions," *Inf. Control*, vol. 33, no. 2, pp. 142–165, Feb. 1977.
[8] C. Tsai and M. Marek-Sadowska, "Multilevel logic synthesis for arithmetic functions," in *Proc. 33rd DAC*, Jun. 1996, pp. 242–247.
[9] C. Yang, M. Ciesielski, and V. Singhal, "BDS: A BDD-based logic optimization system," in *Proc. 37th ICCAD*, 2000, pp. 92–97.
[10] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," *ACM Trans. Des. Autom. Electron. Devices*, vol. 7, no. 4, pp. 501–525, Oct. 2002.
[11] W. Wan and M. A. Perkowski, "A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to FPGA mapping," in *Proc. Eur. Des. Autom. Conf.*, 1992, pp. 230–235.
[12] T. Sasao and J. T. Butler, "A design method for look-up table type FPGA by pseudo-Kronecker expansion," in *Proc. 24th Int. Symp. Multi-Valued Logic*, 1994, pp. 97–106.
[13] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.
[14] Y. T. Lai, M. Pedram, and S. Sastry, "BDD based decomposition of logic functions with application to FPGA synthesis," in *Proc. 30th Des. Autom. Conf.*, 1993, pp. 642–647.
[15] C. Yang, V. Singhal, and M. Ciesielski, "BDD decomposition for efficient logic synthesis," in *Proc. Int. Conf. Comput. Des.*, 1999, pp. 626–631.
[16] H. Anton and C. Rorres, *Elementary Linear Algebra, Applications Version*, 7th ed. Hoboken, NJ: Wiley, 1994.
[17] F. Somenzi, *CUDD: CU Decision Diagram Package, Release 2.4.1*. [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD
[18] Berkeley Logic Synthesis Group, *ABC: A System for Sequential Synthesis and Verification*, Dec. 2005. [Online]. Available: http://www.eecs.berkeley.edu/~alanmi/abc
[19] T. S. Czajkowski and S. D. Brown, "Functionally linear decomposition and synthesis of logic circuits for FPGAs," in *Proc. IEEE 45th DAC*, Jun. 2008, pp. 18–23.
[20] T. T Hwang, R. M. Owens, and M. J. Irwin, "Exploiting communication complexity for multilevel logic synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 9, no. 10, pp. 1017–1027, Oct. 1990.
[21] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, *SIS: A System for Sequential Circuit Synthesis*. Berkeley, CA: Dept. EECS, Univ. California, 1992.
[22] T. T Hwang, R. M. Owens, and M. J. Irwin, "Efficiently computing communication complexity for multilevel logic synthesis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 11, no. 5, pp. 545–554, May 1992.
[23] A. Bernasconi, V. Ciriani, and R. Cordone, "The optimization of kEP-SOPs: Computational complexity, approximability and experiments," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 13, no. 2, pp. 1–31, 2008. art. no. 35.
[24] C. L. Berman, "Ordered binary decision diagrams and circuit structure," in *Proc. IEEE Int. Conf. Comput. Des.*, Cambridge, MA, Oct. 1989, pp. 392–395.
[25] S. Yang, "Logic synthesis and optimization benchmarks User Guide 3.0," Microelectron. Center North Carolina, Research Triangle Park, NC, 1991. Tech. Rep.

**Tomasz S. Czajkowski** (M'02) received the B.A.Sc., M.A.Sc., and Ph.D. (exploring a relatively recent approach to field-programmable gate array (FPGA) computer-aided design flow called physical synthesis) degrees from the Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada, in 2001, 2004, and 2008, respectively.

He is currently with the University of Toronto. His work focused on logic implementation and optimization of circuits as it pertains to FPGA devices, looking in particular at how various logic structures can be utilized more effectively. In this context, he is mostly interested in the synthesis and physical implementation of logic circuits, taking into account logic structures available on FPGAs, circuit performance, area, and power dissipation.

**Stephen D. Brown** (M'89) received the Ph.D. and M.A.Sc. degrees in electrical engineering from the University of Toronto, Toronto, ON, Canada, and the B.A.Sc. degree in electrical engineering from the University of New Brunswick, Fredericton, NB, Canada.

He was with the University of Toronto faculty in 1992, where he currently holds the rank of Professor with the Department of Electrical and Computer Engineering. He also holds the position of Architect with the Altera Toronto Technology Centre, Toronto, a world-leading research and development site for computer-aided design (CAD) software and field-programmable gate array architectures, where he is involved in research activities and is the Director of the Altera University Program. His research interests include field-programmable very large scale integration technology, CAD algorithms, and computer architecture. He is a coauthor of more than 60 scientific research papers and three textbooks: *Fundamentals of Digital Logic with Verilog Design, Fundamentals of Digital Logic with VHDL Design*, and *Field-Programmable Gate Arrays*.

Dr. Brown won the Canadian Natural Sciences and Engineering Research Councils 1992 Doctoral Prize for the best Ph.D. thesis in Canada. He has won multiple awards for excellence in teaching electrical engineering, computer engineering, and computer science courses.