CORE
Metadata, citation and similar papers at core.ac.uk
Provided by PDXScholar

Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

Summer 7-21-2017

Synthesis of Linear Reversible Circuits and EXOR-AND-based Circuits for Incompletely Specified Multi-Output Functions

Ben Schaeffer
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the VLSI and Circuits, Embedded and Hardware Systems Commons
Let us know how access to this document benefits you.

Recommended Citation

Schaeffer, Ben, "Synthesis of Linear Reversible Circuits and EXOR-AND-based Circuits for Incompletely Specified Multi-Output Functions" (2017). *Dissertations and Theses.* Paper 3783.
https://doi.org/10.15760/etd.5667

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Synthesis of Linear Reversible Circuits and EXOR-AND-based Circuits for Incompletely Specified Multi-Output Functions

by

Ben Schaeffer

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computing Engineering

Dissertation Committee:
Marek Perkowski, Chair
Douglas Hall
Steven Bleiler
Yih-Chyun Jenq
Zeshan Chishti

Portland State University
2017

© 2017 Ben Schaeffer

Abstract

At this time the synthesis of reversible circuits for quantum computing is an active area of research. In the most restrictive quantum computing models there are no ancilla lines and the quantum cost, or latency, of performing a reversible form of the AND gate, or Toffoli gate, increases exponentially with the number of input variables. In contrast, the quantum cost of performing any combination of reversible EXOR gates, or CNOT gates, on n input variables requires at most $O(n^2/\log_2 n)$ gates. It was under these conditions that EXOR-AND-EXOR, or EPOE, synthesis was developed.

In this work, the GF(2) logic theory used in EPOE is expanded and the concept of an EXOR-AND product transform is introduced. Because of the generality of this logic theory, it is adapted to EXOR-AND-OR, or SPOE, synthesis. Three heuristic spectral logic synthesis algorithms are introduced, implemented in a program called XAX, and compared with previous work in classical logic circuits of up to 26 inputs. Three linear reversible circuit methods are also introduced and compared with previous work in linear reversible logic circuits of up to 100 inputs.

Acknowledgments

I would like to thank my advisor Marek Perkowski, the graduate office representative Steven Bleiler, committee members Douglas Hall, Yih-Chyun Jenq, and Zeshan Chishti, and members of the Portland Quantum Logic Group for their time and feedback. I would like to thank Addy Gronquist for his copyediting efforts and work in creating parallel-processing versions of the predecessor to XAX, as well as his conversations with Milena Stanković and Radomir S. Stanković that advanced this work to its present state. They all have my gratitude.

Table of Contents

Abstract i

Acknowledgements ii

List of Tables v

List of Figures vii

1. Introduction 1

2. Theory 3

   2.1 Definitions 3

   2.2 Theorems 10

   2.3 Matrix Inversion 11

3. Linear Reversible Circuit Synthesis 14

   3.1 Overview 14

   3.2 Subrow Elimination-based Methods for Inverting GF(2) Matrices 17

   3.3 Alternating Elimination 21

   3.4 The Multiple CNOT Gate Method 24

   3.5 Variants of Gauss-Jordan Elimination 25

   3.6 Experimental Results 34

   3.7 Conclusion 40

4. Linear Reversible Circuit Synthesis 42

   4.1 Overview 42

   4.2 Theoretical Comparisons with Other Approaches 48

   4.3 Introduction to XAX 49

4.4 XAX EPOE Synthesis of Completely Specified Functions 51

4.5 XAX EPOE Synthesis of Incompletely Specified Functions 55

4.6 XAX SPOE Synthesis of Incompletely Specified Functions 58

4.7 Postprocessing 61

4.8 Complexity 62

4.9 Experimental Results 63

4.10 Conclusion 77

5. Summary 79

References 81

List of Tables

Table 3.1. CNOT gate truth table 14

Table 3.2. Comparisons of linear reversible circuit synthesis methods (average CNOT gates) 36

Table 3.3. Comparisons of quantum linear reversible circuit synthesis methods (average quantum gates) 37

Table 3.4. Best performing minimum density subcolumn parameter (*dmin*) frequencies corresponding to quantum gate counts produced by the GJCO4 method 40

Table 4.1. Total gate inputs (#TI) for XAX EPOE and ESPP synthesis of selected MCNC benchmark functions 69

Table 4.2. Total gate inputs (#TI) of both XAX EPOE synthesis methods of selected incompletely specified MCNC and Espresso benchmark functions 70

Table 4.3. Total CMOS gate inputs (#TI) for EXORCISM4, EXORCISM4 with quality=16, and XAX EPOE/EPOE with don’t cares using multi-output synthesis of selected Espresso benchmark functions 72

Table 4.4. Total CMOS gate inputs (#TI) for XAX EPOE default mode, greedy mode, and random mode syntheses of selected MCNC benchmark functions 73

Table 4.5. Total CMOS gate inputs (#TI) for XAX SPOE default mode, greedy mode, and random mode syntheses of selected MCNC benchmark functions 74

Table 4.6. Literals for XAX SPOE using a single postprocessing stage, SPPk where $k = 0$ (SPP0), and Exact SPP synthesis of selected Espresso benchmark functions 75

Table 4.7. Total CMOS gate inputs (#TI) for exact-minimum SOP, XAX SPOE, and Exact SPP synthesis of selected Espresso benchmark functions 76

Table 4.8. Total CMOS gate inputs (#TI) for XAX SPOE and ORAX synthesis of selected Espresso benchmark functions 77

List of Figures

Figure 2.1. The pseudocube matrix corresponding to the pseudoproduct $(x_0 \oplus x_1)(x_2 \oplus x_3)$ 5

Figure 3.1. The reversible circuit schematic of the CNOT gate 14

Figure 3.2. The Matrix representation of an adjacent CNOT-down gate 15

Figure 3.3. The GF(2) Gaussian Elimination-based synthesis output of function M_{F1} 17

Figure 3.4. The "Algorithm 1" synthesis output of function M_{F1} using $m = 2$ 19

Figure 3.5. A 4×4 linear function, the corresponding "Algorithm 1" circuit realization (middle), and an exact minimum circuit realization (right) 20

Figure 3.6. A CNOT gate-based circuit and two logically equivalent quantum gate-based circuits 20

Figure 3.7. The 16-gate, GJCO1 linear reversible circuit realization of the function in (13) 32

Figure 3.8. The 30-gate GJCO1 quantum linear reversible circuit realization of the function in (13) 32

Figure 3.9. The 15-gate, A1CO linear reversible circuit realization of the function in (13) 33

Figure 3.10. The 35-gate, A1CO quantum linear reversible circuit realization of the function in (13) which contains two redundant gates 33

Figure 3.11. The 15-gate, A1CO linear reversible circuit realization of the function in (13) after gate reordering 33

Figure 3.12. The 33-gate, A1CO quantum linear reversible circuit realization of the function in (13) after gate reordering 33

Figure 3.13. The average CNOT gate counts of synthesizing 100 randomized linear reversible circuits with multiple methods 38

Figure 3.14. The average quantum gate counts of synthesizing 100 randomized linear reversible circuits with multiple methods 39

Figure 4.1. A logic function used to discuss redundant cubes in SOP minimization 42

Figure 4.2. The Karnaugh maps of alternative ESOP syntheses of the function F_1: a) for expression $cef \oplus ac'def$ and b) for expression $ef \oplus c'ef \oplus ac'def$ 44

Figure 4.3. The Karnaugh maps of alternative ESOP syntheses of function F_2: a) for expression $ef \oplus c'd'ef \oplus a'b'c'def$ and b) for expression $ef \oplus c'ef \oplus c'def \oplus a'b'c'def$ 45

Figure 4.4. The Karnaugh map and corresponding spectral coefficients of an example switching function 47

1

Introduction

A number of different quantum computing architectures, such as quantum dot, scalable ion-trap, and linear optical quantum computing [1, 2], have been proposed and used experimentally. There are many challenges in realizing a quantum computer and it remains an open question which approach will be the most commercially viable. A parallel open question is how best to convert irreversible Boolean functions into a quantum form (or quantum circuit). A commonly used quantum computing model for logic synthesis is one in which ancilla lines are unavailable, the quantum cost or latency of performing a reversible form of the AND gate (known as the multiple-controlled Toffoli gate) increases exponentially with the number of input variables, and the quantum cost of performing a reversible form of the EXOR gate (which is known as the CNOT gate) is one [3]. It was under these conditions that the EXOR-AND-EXOR (or EPOE) logic synthesis methodology presented herein was developed [4]. Specifically, the original goal for EPOE logic synthesis was to realize reversible circuits which would use a minimal number of Toffoli gates at a cost of increasing the number of CNOT gates. This Toffoli-gate-CNOT-gate tradeoff strategy is general and can lead to multi-level logic forms significantly more complicated than the three-level forms presented herein.

In this work the two fundamental problems in EPOE logic synthesis, Linear Reversible Circuit Synthesis and EXOR-AND-based logic synthesis, are covered in separate sections. Previous work on linear reversible circuit synthesis used the Four Russians Method for Inverting a GF(2) matrix to achieve an upper-bound of $O(n^2/$

$\log_2 n$) CNOT gates per circuit [5, 6]. Compared to that work, the methods presented herein translate more efficiently into both reversible gates and quantum gates.

The closest previous work to EPOE synthesis was an Exclusive-OR sum of pseudoproducts (or ESPP) approach developed by Ishikawa et al. [7]. Their experiments were limited to circuits of 10 inputs with no synthesis times reported, they had no optimizations for multi-output functions, and they did not make the connection to quantum computing. In contrast there have been many papers on EXOR-AND-OR synthesis formulated as a sum of pseudoproducts (or SPP) which was introduced by Luccio and Pagli [8]. Herein a technology-independent approach to EXOR-AND-based logic synthesis is presented and compared with previous work. For compatibility with reversible circuit notation, the presented EXOR-AND logic synthesis algorithms use a GF(2) linear transform formulation employed in describing error correction codes and logic decomposition by Meinel and Theobald [9], and Günther and Drechsler [10], and Karpovsky, Stanković, and Astola [11]. Comparisons are made between the GF(2) linear transform formulation and the SPP/pseudoproduct formulation in order to bridge these two approaches.

The presented EXOR-AND-based logic synthesis algorithms differ from the majority of logic synthesis algorithms in two fundamental ways. First, they rely on EXOR-AND product transformations in a postprocessing stage to increase similarity across EXOR-AND expressions, and second, they employ a Hadamard transform spectral approach to generate logic expressions. This spectral approach improves the speed of synthesis but at a cost of increased variability in the results compared to other methods.

2

Theory

2.1 Definitions (extending [12])

Previously defined terms relevant to logic synthesis [13, 14, 15, 16]

The following conventions are used herein:

1. Logical sums are denoted by terms separated by the "$\vee$" operator.
2. Logical products are denoted by both "$\prod x_i$" and concatenated expressions.
3. Boolean expressions that contain products of Exclusive-OR sums will be expressed in algebraic normal form, e.g., as $(x_i \oplus 1)(x_j \oplus x_k \oplus 1)$ rather than $\overline{x}_i(\overline{x}_j \oplus x_k)$, $\overline{x}_i(x_j \oplus \overline{x}_k)$, or $\overline{x}_i\overline{(x_j \oplus x_k)}$.
4. Where noted Galois Field 2 (GF(2)) matrix algebra will be used. In GF(2) algebra, values are restricted to either 0 or 1, multiplication is equivalent to logical AND, and addition is modulo two.

A completely specified $n \times 1$ *switching function*, or equivalently a *two-valued logic function*, is a Boolean mapping of the form $f: \mathbb{B}^n \rightarrow \mathbb{B}$. The n input variables of a switching function are specified herein in one of two ways: either as an n-bit binary variable in the form $x_0 x_1 x_2 \dots x_{n-1}$, or as an $n \times 1$ binary vector in the form $\vec{x} = [x_0, x_1, \dots, x_{n-1}]^T$. Given a constant $n \times 1$ binary vector $\vec{a}$ in the domain of switching function f, the expressions $f\big|_{a_0 a_1 a_2 \dots a_{n-1}} \equiv f(a_0, a_1, a_2, \dots a_{n-1}) \equiv f(\vec{a})$ denote the particular value in the range of f to which f maps $\vec{a}$. When the domain of a switching function f is restricted by holding a collection of input variables constant, the restricted

functions will be denoted similarly. For instance, the expressions $f|_{a_0} \equiv f|_{x_0=a_0} \equiv f(a_0, x_1, x_2, \dots x_{n-1})$ denote the subfunction of f where $x_0 = a_0$; $f|_{a_0a_1} \equiv f|_{x_0=a_0,x_1=a_1} \equiv f(a_0, a_1, x_2, \dots x_{n-1})$ denote the subfunction of f where $x_0 = a_0$ and $x_1 = a_1$, etc. In the logic synthesis literature these restricted functions are often called cofactors. The specification of a switching function can be denoted in multiple ways, such as a vector of 2^n binary constants in the form $[f|_{0\dots000}, f|_{0\dots001}, f|_{0\dots010}, \dots, f|_{1\dots111}]$, or as a set of all binary vectors $\vec{x}$ such that $f(\vec{x}) = 1$. The latter specification is sometimes referred to as the *ON-set* of f. Further, a given ON-set can be denoted either as a list of unsigned binary values or as a list of integers via the $\mathbb{B}^n \to \mathbb{Z}$ mapping $\vec{x} \to 2^{n-1}x_0 + 2^{n-2}x_1 + 2^{n-3}x_2 \dots + 2^0x_{n-1}$. The *OFF-set* of f is the Boolean complement in the domain of f of the ON-set for f, i.e., the set of all $\vec{x}$ such that $f(\vec{x}) = 0$.

In an *incompletely specified switching function*, binary vectors in the domain of a switching function belong to either the *ON-set* of f, the *OFF-set* of f, or the *don't-care set* (*DC-set*) of f which is the set of all binary vectors $\vec{x}$ for which $f(\vec{x}) = 0$ or $f(\vec{x}) = 1$ is acceptable. Each of these sets can be represented as an independent switching function, i.e., f_{ON}, f_{OFF}, and f_{DC}.

Given the n×1 switching function f, we define the following:

The expression $|f|$ denotes the number of elements in the ON-set of f.

A *literal* l_i is an input variable in either direct or negated form, e.g., l_i where $l_i \in \{\overline{x_i}, x_i\}$.

A *minterm* is a product of exactly *n* literals in which each input variable appears precisely once, e.g., $\prod_{i=0}^{n-1} l_i$ where $l_i \in \{\overline{x_i}, x_i\}$. A minterm can be regarded either as a *1-minterm* when $f_{ON}(l_0, l_1, l_2, \ldots l_{n-1}) = 1$, or a *0-minterm* when $f_{OFF}(l_0, l_1, l_2, \ldots l_{n-1}) = 1$, or a *DC-minterm* when $f_{DC}(l_0, l_1, l_2, \ldots l_{n-1}) = 1$.

A *cube* is either the Boolean value 1, a single literal, or the product of two or more literals, e.g., $c = \prod_{i=0}^{n-1} m_i$ where $m_i \in \{\overline{x_i}, x_i, 1\}$. The cubes c_0 and c_1 are *disjoint* when $c_0 c_1 = 0$ for all $\vec{x}$, otherwise they *intersect* each other.

A *sum of products* (SOP) expression for the function f is a logical sum of cubes which corresponds to the ON-set of f.

The *canonical* SOP expression for the function f is the logical sum of unique minterms which corresponds to the ON-set of f. For example, the canonical SOP expression for the 3×1 function $f_1 = x_0 x_1 \vee x_2$ is $\overline{x}_0 \overline{x}_1 x_2 \vee \overline{x}_0 x_1 x_2 \vee x_0 \overline{x}_1 x_2 \vee x_0 x_1 \overline{x}_2 \vee x_0 x_1 x_2$.

A *disjoint sum of products* (DSOP) expression for the function f is an SOP expression which is comprised of disjoint cubes. Note that all canonical SOP expressions are DSOP expressions. An advantage of DSOP expressions is that they can be treated flexibly as SOP expressions or ESOP expressions.

x_0	x_1	x_2	x_3
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0

Figure 2.1. The pseudocube matrix corresponding to the pseudoproduct $(x_0 \oplus x_1)(x_2 \oplus x_3)$.

Previously Luccio and Pagli defined the terms *pseudocube* and canonical expression of a pseudocube or *pseudoproduct*, [8] which are summarized here briefly. A pseudoproduct such as $(x_0 \oplus x_1)(x_2 \oplus x_3)$ is a product of Exclusive-OR sums of literals (or equivalently a product of *EXOR-factors*) which is algorithmically constructed from a specific type of matrix of minterms called a pseudocube. A pseudoproduct expression will be either 1, an Exclusive-OR sum of one or more literals, or a product in which each factor is either a literal or an Exclusive-OR sum of literals for which the number of literals is bounded. Specifically, in pseudoproduct expressions consisting of p EXOR-factors, the maximum number of literals in each EXOR-factor is $n - p + 1$. Further, a pseudocube is a binary encoded matrix representation of 2^k minterms, where $0 \leq k \leq n$, that satisfies the following requirements: all rows are unique; all rows are sorted in ascending order; the entries in each column are either all 0, all 1, or half 0's and half 1's; and, the upper half of each column is equal to either the lower half of the same column or the Boolean complement of the lower half of the same column. Further, these requirements must continue to hold for all submatrices created from recursively dividing a pseudocube matrix into two submatrices corresponding to its upper-half rows and lower-half rows. For example, the pseudoproduct $(x_0 \oplus x_1)(x_2 \oplus x_3)$ will be constructed from the pseudocube matrix in Figure 2.1.

In its verb form the term *cover* denotes either an expression representing an entire function or an inclusion relationship. With regard to the latter, an example would be as

follows: given the cubes $c_1 = x_0x_1$ and $c_2 = x_0x_1x_2$, since c_1 covers c_2 it follows that $c_1c_2 = c_2$. Conversely c_2 is an *implicant* of c_1.

Given the two nonzero switching functions f_1 and f_2, we define the following:

The function f_1 *approximately covers* or *fractionally covers* f_2 when the ON-set of f_2 is not a subset of the ON-set of f_1 and $|f_1f_2| \neq 0$. Here it follows that $|f_1f_2| < |f_2|$. For example, the function $f_1 = x_0x_1 \vee x_2$ approximately covers the function $f_2 = x_0$. Conversely the function f_2 is an *approximate implicant* of the function f_1.

An *approximate cover criterion* or *fractional cover criterion* is a real number between ½ and 1, typically denoted as *w*, used in the approximate covering criterion equation $|f_1f_2| > w \times |f_2|$. For example, given $w = 0.57$ and the 3×1 functions $f_1 = x_0x_1 \vee x_2$ and $f_2 = x_0$, the equation $3 > 0.57 \times 4$ holds and therefore f_2 is an acceptable approximate implicant of f_1. The approximate covering criteria equation for incompletely specified switching functions is $|f_1^{ON}f_2| > w \times |(f_1^{ON} \vee f_1^{OFF})f_2|$; i.e., minterms in f_1^{DC} are excluded from consideration.

The *Hadamard transform*, or *Walsh-Hadamard transform*, is a self-inverse spectral transform which herein is used to perform the $\mathbb{Z}^n \rightarrow \mathbb{Z}^n$ mapping: multiplication of an input vector of length 2^n with the matrix $(½)^{n/2} \otimes_1^n H_i$ where $H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$. For convenience, elements of this input vector will be restricted to $\{0,1\}$ and the scalar $(½)^{n/2}$ will be omitted; consequently a correcting scalar of $(½)^n$ must be used when computing inverses.

Given an invertible GF(2) $n \times n$ matrix M, an input vector $\vec{x} = [x_0, x_1, x_2, \ldots, x_{n-1}]^T$, a constant vector $\vec{b} = [b_0, b_1, \ldots, b_{n-1}]^T$, and the output vector

$\vec{y} = [y_0, y_1, y_2, \dots, y_{n-1}]^T$, when $\vec{b} = [0, \dots, 0,0,0]^T$ the mapping $\vec{y} = M\vec{x} + \vec{b}$ denotes a GF(2) *linearly transformed vector* or a *linear transform* of $\vec{x}$. When $\vec{b}$ can take any constant value the mapping $\vec{y} = M\vec{x} + \vec{b}$ denotes a GF(2) *affine-linearly transformed vector* or an *affine-linear transform* of $\vec{x}$. In these transforms the matrix M is restricted to invertible matrices, thus preventing elements of $\vec{y}$ from attaining values such as $0x_0$, x_1x_1, $\overline{x}_1x_1$, and $x_0(x_0 \oplus x_1 \oplus 1)x_1$.

The above transformations can be extended to switching functions. For example, linearly transforming all elements of a function's ON-set, OFF-set, and DC-set into a transformed ON-set, transformed OFF-set, and transformed DC-set creates a linearly transformed function.

A *product of Exclusive-OR sums* (POE) is a product of two or more unique elements of $\vec{y}$. For example, given $M = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, $\vec{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$, and $\vec{y} = M\vec{x} + \vec{b} = \begin{bmatrix} x_0 \oplus x_1 \\ x_1 \\ x_2 \oplus x_3 \\ x_3 \end{bmatrix}$, the expression $y_0y_2 = (x_0 \oplus x_1)(x_2 \oplus x_3)$ is a POE expression.

A *product cube of Exclusive-OR sums* (POE cube) is either 1, a component of $\vec{y}$, or the product of two or more unique elements of $\vec{y}$.

A *product minterm of Exclusive-OR sums* (POE minterm) is a POE cube that is a product of all *n* elements of $\vec{y}$. Further, each element of $\vec{y}$ may be in negated form.

A *sum of products of Exclusive-OR sums* (SPOE) expression is a logical sum of POE cubes, e.g., $(x_0 \oplus x_1)x_2 \vee (x_0 \oplus x_1)x_3$. Each POE cube in an SPOE expression may correspond to a different affine-linear transform.

An *Exclusive-OR sum of products of Exclusive-OR sums* (EPOE) expression is an Exclusive-OR sum of POE cubes, e.g., $x_0 \oplus x_1 \oplus (x_0 \oplus x_1)(x_2 \oplus 1)(x_3 \oplus 1)$. Each POE cube in an EPOE expression may correspond to a different affine-linear transform.

A *disjoint sum of products of Exclusive-OR sums* (DPOE) expression for the function f is an SPOE expression comprised of disjoint POE cubes, e.g., $(x_0 \oplus x_1)(x_2 \oplus x_3) \vee (x_0 \oplus x_1)x_2x_3$.

Given rows $m_{k,*}$ and $m_{l,*}$ in matrix M and $k, l \in \{0,1,2, \dots n-1\}$ such that $k \neq l$, *elementary row addition* over GF(2) is the operation $m_{k,*} \oplus m_{l,*} \rightarrow m_{k,*}$. Given corresponding vector elements b_k and b_l in the vector $\vec{b}$, the *linear combination complement* (LCC) operation is the operation $m_{k,*} \oplus m_{l,*} \rightarrow m_{k,*}$, $b_k \oplus b_l \oplus 1 \rightarrow b_k$. When applying LCC operations the property $\prod_{i=0}^{n-1} y_i^{old} = \prod_{i=0}^{n-1} y_i^{new}$ holds. For example, performing the LCC operation $m_{0,*} \oplus m_{2,*} \rightarrow m_{0,*}$, $b_0 \oplus b_2 \oplus 1 \rightarrow b_0$ on the previous matrix M and vector $\vec{b}$ results in $M = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ and $\vec{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$. Further, because of the Boolean identity $y_j y_k = y_j(y_j \oplus y_k \oplus 1)$, it holds that the POE minterm $(x_0 \oplus x_1)x_1(x_2 \oplus x_3)x_3$, associated with the previous $M\vec{x} + \vec{b}$ expression, is equal to the POE minterm $(x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus 1)x_1(x_2 \oplus x_3)x_3$, associated with the resultant $M\vec{x} + \vec{b}$

expression. *Product transformation* is computation of these equivalent EXOR-AND expressions.

When $f\,|_{x_i} = f\,|_{\bar{x}_i}$ holds for one or more input variables of a switching function, it is known as a *degenerate function*. A degenerate function can be linearly transformed, and the linearly transformed function may or may not be degenerate. Conversely Bernasconi et al. define a *k-autosymmetric function* (paraphrasing, using the notation defined herein) as a function which can be linearly transformed into a new function consisting of $n - k$ input variables which are elements of $\vec{y}$ [17, 18]. Alternatively this new function can be viewed as an n-input function which is degenerate in k input variables.

2.2 Theorems

Theorem 1. Given the three unique forms of the POE expression $c = y_k y_l = y_k(y_k \oplus y_l \oplus 1) = (y_k \oplus y_l \oplus 1)y_l$ where the number of input variables n is evenly divisible by three, the number of input variables appearing in the first EXOR factor of c added to the number of input variables appearing in the second EXOR factor of c is at most $\frac{4}{3}n$ in at least one of three unique forms. **Proof.** Let r denote the number of input variables in the vector component y_k. Considering the number of input variables in the vector component y_l and expression $y_k \oplus y_l$, the equation $Min(variables(y_l), variables(y_k \oplus y_l))$ reaches a maximum when y_l and $y_k \oplus y_l$ both consist of $n - r$ input variables not appearing in y_k plus $\frac{1}{2}r$ input variables shared with y_k. As a result, the number of input variables is $h_1(r) = r + \left(n - r + \frac{1}{2}r\right) = n + \frac{1}{2}r$ for

the forms $y_k y_l$ and $y_k(y_k \oplus y_l \oplus 1)$; the number of input variables is $h_2(r) = 2 \times \left(n - r + \frac{1}{2}r\right) = 2n - r$ for the form $(y_k \oplus y_l \oplus 1)y_l$. The equation $Min(h_1, h_2)$ reaches a maximum when $r = \frac{2}{3}n$; therefore the number of input variables is at most $n + \frac{1}{2} \times \frac{2}{3}n = \frac{4}{3}n$ in one of the three unique forms.

Example. Let $n = 6$, $y_k = x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus 1$, and $y_l = x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$. The POE expression $y_k y_l$ collectively has nine input variables, which is greater than $\frac{4}{3}6$=8. From Theorem 1 at least one of the alternative POE expression forms of $y_k y_l$ will not exceed eight input variables. The form $y_k(y_k \oplus y_l \oplus 1)$, which collectively has seven input variables, is the most compact form.

2.3 Matrix Inversion

Given an invertible matrix M_F consisting of real entries, Gaussian Elimination computes M_F^{-1} as the product of three types of elementary row operation matrices: those which scale a row, those which swap two rows, and those which either add one row to or subtract one row from another row. Performing each elementary row operation creates a new remainder matrix as shown in (1). As these remainder matrices evolve they typically become sparser and increasingly similar to the identity matrix. An inverse matrix decomposition produced by Gaussian Elimination can be used to solve for the original function as in (2).

$$M_k \dots (M_3(M_2(M_1 M_F))) = I \qquad (1)$$

$$M_F = M_1^{-1} M_2^{-1} M_3^{-1} \dots M_k^{-1} \qquad (2)$$

In the typical application of Gaussian Elimination on real-valued matrices, elementary row operations are employed first to solve for an upper triangle matrix and then to solve for the identity matrix. This two-phase process can be visualized as solving lower-triangle matrix entries from the left-most column to the right-most column, then solving upper-triangle entries from the right-most column to the left-most column. The entries in a column are individually processed through elementary row operations called *forward substitution* and *backward elimination*. When elementary row operations are employed, forward substitution is defined as the operation $row_i + row_j \rightarrow row_i$ given that matrix entry a_{ii} is zero and a_{ji} is nonzero, and backward elimination is defined as the operation $row_j - row_i \rightarrow row_j$ given that matrix entry a_{ii} is nonzero and $a_{ji} = a_{ii}$. Gaussian Elimination can also be performed with elementary column operations. When elementary column operations are employed, forward substitution is defined as the operation $column_i + column_j \rightarrow column_i$ given that matrix entry a_{ii} is zero and a_{ij} is nonzero, and backward elimination is defined as the operation $column_j - column_i \rightarrow column_j$ given that matrix entry a_{ii} is nonzero and $a_{ij} = a_{ii}$. In the first phase of Gaussian Elimination, solving for a lower triangle matrix could be performed using either elementary row or elementary column operations.

A GF(2) variant of Gaussian Elimination can be used to compute the inverse of an invertible Boolean matrix. In GF(2) Gaussian Elimination, matrix multiplication and addition operations become vector-based AND and EXOR operations. Considering elementary row operations with GF(2) matrices, there is no operation analogous to scaling a row, nor is there a difference between row subtraction operations and row

addition operations. Two rows can be swapped via three modulo-2 (i.e., EXOR) row addition operations. Consequently GF(2) Gaussian Elimination can be performed by using only elementary row addition operations. Because GF(2) elementary row addition operations are self-inverse, any GF(2) function inverse decomposition can be simplified as follows.

$$M_F = M_1^{-1} M_2^{-1} M_3^{-1} \dots M_k^{-1} = M_1 \; M_2 \; M_3 \; \dots M_k \qquad (3)$$

3

Linear Reversible Circuit Synthesis

3.1 Overview

Expanding [20, 21],

… reversibility in a gate or circuit means that the gate or circuit implements a bijective correspondence between the input and output sets. In particular, the two-input, two-output controlled-NOT (CNOT) gate, illustrated in Figure 3.1 and defined in Table 3.1, does precisely this for the exclusive or (EXOR) operation. When the implemented correspondence is in fact linear, then the implementing gate or circuit is defined as a linear reversible gate or circuit. Note that linear reversible circuits form a natural subclass of the class of reversible circuits.

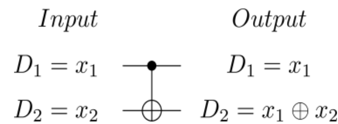

Figure 3.1. The reversible circuit schematic of the CNOT gate.

Table 3.1. CNOT gate truth table.

Input		Output	
D_1	D	D_1	D_2
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

When primitive swap gates are unavailable, linear reversible circuits can be GF(2) synthesized solely from CNOT gates which will be the treatment herein. Moreover, linear reversible circuits often play a bridging role in more complex reversible circuits. For

example, linear reversible circuits can realize permutation functions or can prepare affine-linear functions used on the control lines of a Toffoli gate.

Given a Boolean input vector $\vec{x}$ of length *n,* an invertible GF(2) matrix M of dimension $n \times n$, and Boolean output vector $\vec{y}$ of length n, the general equation for Boolean linear reversible functions is $\vec{y} = M\vec{x}$. The matrix *M* represents the GF(2) linear function to be performed by a linear reversible circuit, with each row of *M* corresponding to a reversible circuit line whose output value is the EXOR sum of corresponding input variables. For instance, the $GF(2)$ product of the input vector $\vec{x} = [x_1, x_2, \dots, x_n]^T$ and the matrix in Figure 3.2 is $\vec{y} = [x_1, x_1 \oplus x_2, x_3, x_4, \dots, x_n]^T$. This matrix is an example of an elementary row addition matrix. Elementary row addition matrices are similar to the identity matrix except that they contain one off-diagonal entry with a value of 1. Each elementary row addition matrix synthesizes to one CNOT gate with a control value equal to the column of the off-diagonal entry and a target value equal to the row of the off-diagonal entry. Multiplication with the matrix in Figure 3.2 performs the elementary row addition operation $Row_1 \oplus Row_2 \rightarrow Row_2$; this matrix is synthesized as a CNOT gate with a control line of one and a target line of two. For convenience this type of gate will be referred to as an adjacent CNOT-down gate, and if the control line is immediately below the target line it will be referred to as an adjacent CNOT-up gate.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

Figure 3.2. The matrix representation of an adjacent CNOT-down gate.

When elementary row operations are restricted to modulo-2 row addition, Gaussian Elimination can be adapted to synthesize linear reversible circuits over GF(2). As Gaussian Elimination of a function evolves, synthesis is performed by converting each modulo-2 row addition to an ordered pair representing CNOT gate control and target values. These values are stored in an output-side CNOT gate list, as applying the CNOT gate list to a reversible circuit in forward order executes the linear reversible function M^{-1}. Applying a Gaussian Elimination-generated CNOT gate list to a reversible circuit in reverse order executes the linear reversible function M.

Alternative syntheses can be generated several ways. In the typical case where $M^{-1} \neq M$, M^{-1} can be synthesized to an input-side CNOT gate list which may be shorter than the original. Similarly, alternative syntheses can be generated from the transposed matrix M^T and its inverse. Due to the linear algebra property $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$, a CNOT gate list resulting from transposed matrix synthesis must have the target and control values of each gate swapped before it can be used [6].

Each elementary row addition operation is denoted as CNOT(*control*, *target*) where $control = row_{modifying}$ and $target = row_{modified}$ for each M_i. For example, in (4) the matrix M_{F1} represents the function $f([a, b, c, d]^{\mathrm{T}}) = [a, a \oplus b, a \oplus b \oplus c, d]^{\mathrm{T}}$; Gaussian Elimination generates the CNOT gate list CNOT(1, 2), CNOT(1, 3), and CNOT(2, 3).

$$M_{F1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, M_{F1}^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (4)$$

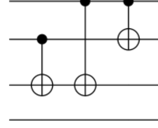

Figure 3.3. The GF(2) Gaussian Elimination-based synthesis output of function M_{F1}.

Performing GF(2) Gaussian Elimination with elementary column operations leads to the decomposition in (5) and (6). In this case GF(2) Gaussian Elimination generates a CNOT gate list from input to output, with each elementary column operation corresponding to a CNOT(*control*, *target*) gate where $control = column_{modified}$ and $target = column_{modifying}$ for each M_i. Under the convention that rows are processed from bottom to top in the first phase, column operation-based Gaussian Elimination of M_{F1} synthesizes the gates CNOT(2, 3), CNOT(1, 3), and CNOT(1, 2). While this resulting gate list is identical to the prior list produced with row operations, the list will typically be different in larger and more complicated functions.

$$(((M_F M_1)M_2)M_3) \dots M_k = I \quad (5)$$

$$M_F = M_k^{-1} \dots M_3^{-1} M_2^{-1} M_1^{-1} = M_k \dots M_3 \; M_2 \; M_1 \quad (6)$$

3.2 Subrow Elimination-based Methods for Inverting GF(2) Matrices

The Four Russians Method for inverting a GF(2) matrix (4RMI) [5] and its related linear reversible circuit synthesis method “Algorithm 1” [6] are variants of GF(2) Gaussian Elimination. While GF(2) Gaussian Elimination iteratively processes entries in a single column, 4RMI and “Algorithm 1” use *subrow elimination* to iteratively process entries in *m* adjacent columns. In our implementation of “Algorithm 1” the subrow width variable *m* is computed as $\lfloor ½ log_2 n + ½ \rfloor$. As a result *m* tends to have small values such as two, three, or, when synthesising linear reversible circuits with 128 through 511 lines,

four. The key idea “Algorithm 1” inherits from 4RMI is that equivalent subrows are guaranteed to occur whenever $2^m < n - i + 1$, given that elementary row operations are used and the left-most unprocessed lower-triangle entries are in the range a_{ii} through a_{ni}. For dense matrices where $2^{m+1} < n - i + 1$ frequently holds, it is likely that the majority of elementary row operations needed to process m adjacent columns can be performed in one $O(n)$ pass. This leaves each column except the last with an $O(1)$ forward substitution and all columns with $O(log_2 n)$ backward eliminations. These substitution and elimination operations can be performed via GF(2) Gaussian Elimination or a table lookup when m is small.

The “Algorithm 1” method uses a strategy of iteratively processing n/m sets of m adjacent columns to achieve an upper-triangle matrix, transposing the matrix, and again iteratively processing m columns to achieve an identity matrix. The corresponding linear reversible circuit is synthesized from the function decomposition derived in (7), (8), and (9).

$$M_k \dots (M_{j+3}(M_{j+2}(M_{j+1}(M_j \dots (M_3(M_2(M_1 M_F))))^T)))) = I \quad (7)$$

$$M_F = M_1^{-1} M_2^{-1} M_3^{-1} \dots M_j^{-1} (M_k^{-1})^T \dots (M_{j+3}^{-1})^T (M_{j+2}^{-1})^T (M_{j+1}^{-1})^T \quad (8)$$

$$M_F = M_1 \ M_2 \ M_3 \ \dots M_j \ M_k^T M_{k-1}^T M_{k-2}^T \dots M_{j+3}^T M_{j+2}^T M_{j+1}^T \quad (9)$$

In “Algorithm 1” subrow elimination, operations are performed through a table with 2^m entries. This table operates similarly to the tag storage component of a direct-mapped cache, with each entry consisting of either an *INVALID* marker or a valid row index. The table position of each valid row index is computed by treating each corresponding subrow as an m-bit number. Because zero subrows are treated as solved, it

is unnecessary to have or use table index 0. Subrow elimination is performed on each column by first setting all table entries to *INVALID*, then processing subrows [$a_{(i,\, i)}$, $a_{(i,\, i+1)}$, $a_{(i,\, i+2)}$, ... $a_{(i,\, i+m-1)}$] through [$a_{(n,\, i)}$, $a_{(n,\, i+1)}$, $a_{(n,\, i+2)}$, ... $a_{(n,\, i+m-1)}$] as follows:

```
For each j from i to n
  If table_entry[a(j, i), a(j, i + 1), a(j, i + 2), ... a(j, i + m - 1)] = INVALID
    table_entry[a(j, i), a(j, i + 1), a(j, i + 2), ... a(j, i + m - 1)] := j
  Else
    CNOT(table_entry[a(j, i), a(j, i + 1), a(j, i + 2), ... a(j, i + m - 1)], j)
```

Performing "Algorithm 1" on function M_{F1} using $m = 2$ results in row_1 being stored in the table, row_2 being stored in the table, a subrow elimination with row_2 modifying row_3, and a backward elimination with row_1 modifying row_2. This yields the linear reversible circuit shown in Figure 3.4, going from output toward input, a CNOT(2, 3) gate and a CNOT(1, 2) gate.

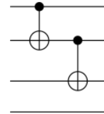

Figure 3.4. The "Algorithm 1" synthesis output of function M_{F1} using $m = 2$.

Linear reversible circuit synthesis via Gaussian Elimination has an upper bound of $O(n^2)$ CNOT gates, which typically is not minimal. Previously Patel, et. al. [6] established that the "Algorithm 1" method has an upper bound of $O(n^2/\log_2 n)$ CNOT gates. While "Algorithm 1" is "asymptotically optimal up to a multiplicative constant"

[6], i.e., bounded, it is too simplistic to find an exact minimum solution to the function $y = [x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, x_1 \oplus x_2 \oplus x_3 \oplus x_4]^T$ illustrated in Figure 3.5 [22].

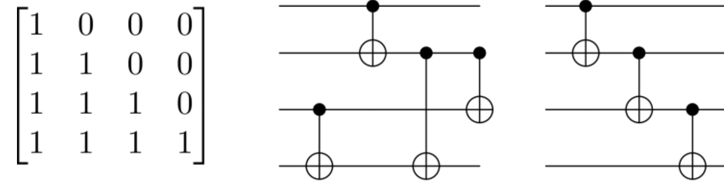

Figure 3.5. A 4×4 linear function, the corresponding “Algorithm 1” circuit realization (middle), and an exact minimum circuit realization (right).

As an alternative to “Algorithm 1”, three linear reversible circuit synthesis methods are introduced herein: the Alternating Elimination with Cost Minimization method (AECM), the Multiple CNOT Gate method (MCG), and several variants of the Gauss-Jordan Elimination with Column Operations (GJCO) method. Both the AECM and MCG methods were developed to reduce CNOT gate counts in small-sized to medium-sized circuits. The GJCO method was developed to reduce quantum gate counts.

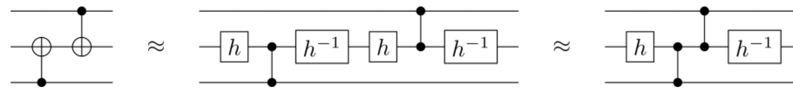

Figure 3.6. A CNOT gate-based circuit and two logically equivalent quantum gate-based circuits.

The mechanism by which the GJCO method reduces quantum gate counts is illustrated in Figure 3.6 and reasoned as follows: In many quantum technologies CNOT gates are implemented by groups of pseudo-Hadamard, controlled-Z, and inverse pseudo-Hadamard gates. When adjacent CNOT gates use the same target lines, pseudo-

Hadamard-inverse/pseudo-Hadamard gate pairs result; these gate pairs can be eliminated from the circuit. Since a series of backward eliminations using column operations synthesizes groups of CNOT gates that use the same target lines, synthesis methods that use column operations are of interest.

In comparison with the “Algorithm 1,” the AECM, MCG, and GJCO methods solve rows and columns in a data dependent, nondeterministic order which provides the greatest cost reduction. The main heuristic cost function depends on the GF(2) linear function M and its inverse (10). When synthesis begins, these matrices correspond to the input function specification, and as CNOT gates are synthesized, these matrices correspond to a remainder function more closely resembling the identity matrix. An alternative cost function (11) is based on the sparseness of a matrix and its inverse. The alternative cost function facilitates synthesis of a permutation of an input linear reversible function specification [22].

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\left(\left(M_{(i,j)}\oplus I_{(i,j)}\right)+\left(M_{(i,j)}^{-1}\oplus I_{(i,j)}\right)\right) \quad (10)$$

$$-2n+\sum_{i=1}^{n}\sum_{j=i}^{n}\left(M_{(i,j)}^{-1}+M_{(i,j)}\right) \quad (11)$$

3.3 Alternating Elimination

The AECM method is built on the Alternating Elimination method [21]. The Alternating Elimination method extends the Gaussian Elimination approach of forward substitution and backward elimination to iteratively process diagonal entries. Since there are $n!$ possible orderings of diagonal matrix entries, Alternating Elimination can generate

a large number of functionally equivalent circuit solutions which have a range of CNOT gate counts.

Because AECM is based on Alternating Elimination, it will always converge to a solution for any invertible GF(2) matrix given as input [21] and therefore cannot be trapped in a local minimum. The AECM method iteratively compares $O(n)$ matrix diagonalizations and then commits to the diagonalization which produces the greatest cost reduction per CNOT gate ratio. It is possible, though rare, that for one iteration all matrix diagonalizations will result in a negative cost reduction per CNOT gate.

The AECM diagonalization function has four stages: preparation, forward substitution, row-based backward elimination, and column-based backward elimination. In each stage the changes in cost of choosing different CNOT gates are compared. The preparation stage performs preprocessing through an $O(n)$ search to find row and column forward substitutions which reduce cost by two or more. Using a cost reduction of at least two is based on testing which showed that in over half the syntheses examined it produced lower CNOT gate counts than using a cost reduction of at least one or skipping the preprocessing stage. Using a cost reduction of at least three was in some instances superior and in other instances inferior to using at least two. Each CNOT gate synthesized in this stage replaces two or more CNOT gates which would have been synthesized in the third and fourth stages.

If the diagonal matrix entry associated with the current iteration is a 0, then a forward substitution stage is used. In this stage either a row or column forward substitution is chosen through an $O(n)$ search to find the CNOT gate which establishes a

1 on the diagonal and results in the lowest-cost remainder function. When it is necessary to perform a forward substitution, a check is made to ensure that the forward substitution CNOT gate was not synthesized in the first stage. This situation is unusual but possible. In these cases the CNOT gate list can be rearranged to detect pairs of identical CNOT gates. Because CNOT gates are self-inverse, all detected identical CNOT gate pairs can be erased.

In the row-based backward elimination stage $O(n)$ eliminations are performed to process column entries which are equal to 1. Unlike Gauss-Jordan Elimination which performs eliminations using the diagonally intersecting row, here each row elimination employs an $O(n)$ search to find the lowest-cost backward elimination operation. Similarly in the column-based backward elimination stage $O(n)$ eliminations are performed to process row entries which are equal to 1, each employing an $O(n)$ search to find the lowest-cost backward elimination operation.

Performing one row or column addition with a cost difference computation takes $O(n)$ time. Therefore the entire AECM diagonalization function takes $O(n) \cdot (O(n) + O(n) + O(n^2) + O(n^2) \approx O(n^3)$ time. Since the outer AECM loop requires $O(n)$ iterations through $O(n)$ comparisons, the total time is $O(n) \cdot O(n) \cdot O(n^3) \approx O(n^5)$.

In order to support partial syntheses the AECM algorithm uses the parameter `threshold`. Using AECM with `threshold` = 0 causes a complete synthesis to be performed. Using AECM with larger `threshold` values causes synthesis to terminate when the cost of the remainder function `c1` goes below `threshold`. In the outermost loop of the algorithm, CNOT gate selection is performed by comparing `gain3` with

gain2. These gain values are computed as $(\texttt{Cost}(f_{k-1}) - \texttt{Cost}(f_k))/(\texttt{Gates}(f_k) - \texttt{Gates}(f_{k-1}))$ for remainder function f at iteration k. The AECM algorithm can be extended to handle occurrences in which these ratios are equal, thus facilitating algorithm extensions such as recursion and probabilistic gate selection.

3.4 The Multiple CNOT Gate Method

The MCG synthesis method synthesizes linear reversible circuits as a collection of linear subfunctions, and each of these subfunctions may correlate to two or more CNOT gates. Subfunction selection is based solely on cost minimization, and consequently the MCG method is completely outside the Gaussian Elimination family of methods. Herein the MCG synthesis method is restricted to two-CNOT-gate subfunctions. This is a practical choice, as larger subfunctions require significant increases in computation time.

The two-CNOT-gate subfunctions can be categorized as one of three types: 1) subfunctions of two elementary row operations corresponding to two CNOT gates synthesized from output toward input; 2) subfunctions of two elementary column operations corresponding to two CNOT gates synthesized from input toward output; 3) subfunctions of one elementary row operation and one elementary column operation representing one CNOT gate synthesized from output toward input and another synthesized from input toward output. The MCG method iteratively compares the cost of applying all possible two-CNOT-gate subfunctions and commits to the pair of CNOT gates which produces the greatest cost reduction. In the event that the cost reaches a local minimum, synthesis temporarily switches to AECM until the cost drops below the local

minimum. In this situation a flag is set indicating that MCG failed to converge and MCG synthesis resumes. In each iteration, MCG retrieves a two-CNOT-gate subfunction from an $O(n^4)$ length list. Performing elementary row or column operations and cost difference computations on each two-CNOT-gate subfunction requires $O(n)$ time. Since the maximum cost is $2n^2$, the smallest cost 0, and the minimum cost reduction is 1 at each iteration, the two-CNOT-gate function search takes at most $O(n^2)$ outermost loop iterations. Therefore the total time is $O(n^4) \cdot O(n) \cdot O(n^2) \approx O(n^7)$.

Like AECM, MCG can be extended to perform more sophisticated gate selections in iterations where multiple minimum-cost alternatives exist. This will be demonstrated later using a probabilistic gate selection. Also, the speed of the MCG algorithm can be improved by using precalculated two-CNOT-gate subfunctions. In the above MCG algorithm all possible CNOT gate sequences are generated, and many will be redundant. For instance, the two-CNOT-gate function CNOT(1, 2) followed by CNOT(3, 4) is equivalent to CNOT(3, 4) followed by CNOT(1, 2). If MCG is extended to use three-CNOT-gate subfunctions, a greater variety of redundant sequences will be generated.

3.5 Variants of Gauss-Jordan Elimination

There are multiple ways to create variants of the Gauss-Jordan Elimination algorithm for linear reversible synthesis. The first variant, named GJCO1, will employ the most rudimentary form of Gauss-Jordan; i.e., performing $O(n)$ iterations of $O(1)$ forward substitutions followed by $O(n)$ backward eliminations. Consequently the GJCO1 method has an upper bound of $O(n^2)$ CNOT gates. Although it is possible to process matrix rows and columns in any order, for simplicity all Gauss-Jordan Elimination

variants will use a convention of processing matrix elements from top to bottom and from left to right.

The remaining Gauss-Jordan Elimination variants introduced herein use combinations of the following seven strategies. A variant of the “Algorithm 1” method which is optimized for quantum linear reversible circuit synthesis will be included for comparison.

Strategies for Quantum Linear Reversible Circuit Synthesis

1. *Use elementary column operations*. Using elementary column operations with GF(2) Gaussian Elimination may require up to $n - 1$ column elimination operations per row. This corresponds to the synthesis of up to $n - 1$ CNOT gates with the same target line per row, minimizing quantum gate counts.

2. *Use a Gauss-Jordan Elimination approach for substitution and elimination operations*. Gauss-Jordan Elimination-based linear reversible circuit synthesis of parallel circuits was introduced in [23], but herein it will be used in the context of serial circuits. Gauss-Jordan Elimination has the advantage of being a single-phase algorithm and, in linear reversible circuit synthesis using elementary column operations, this creates sequences of up to $n - 1$ CNOT gates with the same target line being grouped together.

3. *Use variable-width subcolumn eliminations before processing each row, searching from the longest possible subcolumns to the shortest subcolumns.* This strategy is illustrated using matrix M_{F3} in (12).

$$M_{F3} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ a_{61} & a_{62} & a_{63} & a_{64} & a_{65} & a_{66} \end{bmatrix} \quad (12)$$

The first and second rows of M_{F3} have already been processed and are equal to their corresponding identity matrix rows. Before performing substitution and elimination operations on matrix cell a_{33}, subcolumn eliminations are performed on equal subcolumns of width four, three, and finally two. The width-four subcolumn searches begin with a comparison of $[a_{33}, a_{43}, a_{53}, a_{63}]^T$ with all subcolumns to the left of the third column, $[a_{32}, a_{42}, a_{52}, a_{62}]^T$ and $[a_{31}, a_{41}, a_{51}, a_{61}]^T$. If $[a_{33}, a_{43}, a_{53}, a_{63}]^T$=$[a_{32}, a_{42}, a_{52}, a_{62}]^T$ then the subcolumn elimination $C_2 \oplus C_3 \rightarrow C_2$ is performed and the gate CNOT(2, 3) is added to the input-side gate list. Similarly if $[a_{33}, a_{43}, a_{53}, a_{63}]^T$=$[a_{31}, a_{41}, a_{51}, a_{61}]^T$ then the subcolumn elimination $C_1 \oplus C_3 \rightarrow C_1$ is used and the gate CNOT(1, 3) is added to the input-side gate list. Next $[a_{34}, a_{44}, a_{54}, a_{64}]^T$ is compared with all subcolumns to the left of the third column, followed by $[a_{35}, a_{45}, a_{55}, a_{65}]^T$ and $[a_{36}, a_{46}, a_{56}, a_{66}]^T$ using the same comparison. Because a column cannot be equal to any other column in an invertible matrix, at this stage no comparisons are necessary between subcolumns on and to the right of the third column.

The width-three subcolumn searches begin with a comparison of $[a_{33}, a_{43}, a_{53}]^T$ with all subcolumns to the left and right of the third column. Next is a comparison of $[a_{34}, a_{44}, a_{54}]^T$ with all subcolumns to the left of the third column and to the right of the fourth column, then a comparison of $[a_{35}, a_{45}, a_{55}]^T$ with all subcolumns to the left of the third column and to the right of the fifth column, and lastly a comparison of $[a_{36}, a_{46}, a_{56}]^T$

with all subcolumns to the left of the third column. The width-two subcolumn searches are similar, beginning with a comparison of $[a_{33}, a_{43}]^T$ with all subcolumns to the left and right of the third column, then a comparison of $[a_{34}, a_{44}]^T$ with all subcolumns to the left of the third column and to the right of the fourth column, then a comparison of $[a_{35}, a_{45}]^T$ with all subcolumns to the left of the third column and to the right of the fifth column, and lastly a comparison of $[a_{36}, a_{46}]^T$ with all subcolumns to the left of the third column. In all subcolumn searches only columns on and to the right of the diagonally intersecting column are permitted to modify other columns, thus ensuring solved rows are not disturbed.

There are two qualifying requirements for subcolumns to be considered as acceptable for use in subcolumn elimination. First, the uppermost subcolumn entry must be equal to 1 to avoid disturbing solved columns. Second, the number of entries equal to 1 in a subcolumn must be greater than or equal to a specified minimum subcolumn density parameter (*dmin*). Using a minimum subcolumn density parameter maximizes the effectiveness of isolated subcolumn eliminations. Synthesising the same function with different minimum subcolumn density parameter values can give a range of results depending on the dimensions of the circuit and whether CNOT gate count or quantum gate count is of interest.

The algorithm for variable-width subcolumn elimination is as follows:

```
//i is the index of the row currently being processed
//M is an n by n linear reversible function
//Arrays use 1-based indexes
//dmin is the minimum subcolumn density parameter
```

```
n := M.columns
IF i = n THEN return
FOR c1 FROM i TO n BY 1
  IF (M(i, c1) = 1 AND M.Subcolumn(c1, i, n) has at least
  dmin entries = 1) THEN
    FOR c2 FROM i - 1 TO 1 BY -1
      IF M.Subcolumn(c1, i, n) = M.Subcolumn(c2, i, n) THEN
        DoColumnCNOT(M, c1, c2)
r := n - 1 //Second row from bottom index
IF dmin <= 2 THEN
  boundaryrow := i
ELSE
  boundaryrow := i + dmin - 2
WHILE r > boundaryrow
  FOR c1 FROM i TO n BY 1
    IF (M(i, c1) = 1 AND M.Subcolumn(c1, i, r) has at least
    dmin entries = 1) THEN
      FOR c2 FROM i - 1 TO 1 BY -1
        IF M.Subcolumn(c1, i, r) = M.Subcolumn(c2, i, r) THEN
          DoColumnCNOT(M, c1, c2)
      FOR c3 FROM c2 + 1 TO n BY 1
        IF M.Subcolumn(c1, i, r) = M.Subcolumn(c3, i, r) THEN
          DoColumnCNOT(M, c1, c3)
  r := r – 1
```

4. *Use a cost minimization approach in subcolumn elimination*. During subcolumn elimination, when two or more equivalent subcolumns are discovered on or to the right of a subcolumn being processed, a series of partial syntheses using different column orderings is performed. The partial synthesis with the lowest-cost result is chosen. The cost function in equation (10) is defined as the sum of differences between a given remainder function and the identity matrix added to the sum of differences between the inverse of a given remainder function and the identity matrix. For simplicity and effective quantum gate count minimization, these partial syntheses use one subcolumn to eliminate all equivalent subcolumns. In instances where the subcolumn $[a_{(i,\,i)}, a_{(i\,+\,1,\,i)}, a_{(i\,+\,2,\,i)}, \ldots\, a_{(i\,+\,m\,-\,1,\,i)}]^T$ is one of the matching subcolumns, it is used to modify all other columns.

5. *Use a cost minimization approach in forward substitution.* When multiple columns can be used to perform forward substitution, a partial synthesis is performed using each available column. The partial synthesis works through all elementary row operations for the current row being processed, and then computes the circuit cost. The partial synthesis with the lowest cost is selected.

6. *Use multiple passes.* For the "Algorithm 1" variant which uses column operations, seven syntheses will be performed using subcolumn widths between 2 and 8. For the new Gauss-Jordan Elimination-based methods, seven syntheses will be performed using minimum subcolumn density parameter values between 1 and 7.

7. *Use gate reordering*. Following experimentation, an iterative three-stage gate reordering approach was adopted to postprocess CNOT gate lists from input to output: 1)

when two separate groups of CNOT gates with the same target line are detected, an attempt is made to move the first group toward the output so it is adjacent to the second group; 2) if 1) fails, an attempt is made to move the second, third, fourth,… and *n*th groups of CNOT gates with the same target line toward the input so they are adjacent to the first group; and 3) if 2) fails, an attempt is made to individually relocate each CNOT gate from the second, third, fourth,… and *n*th groups mentioned previously into groups as close to the input as possible. This method is applied *n* times to synthesized CNOT gate lists, once per each possible target line.

The new "Algorithm 1" and Gauss-Jordan Elimination-based methods are organized as follows:

A1CO: employs strategies 1, 6, and 7.

GJCO1: employs strategies 1, 2, and 7.

GJCO2: employs strategies 1, 2, 3, 6, and 7.

GJCO3: employs strategies 1, 2, 3, 4, 6, and 7.

GJCO4: employs strategies 1, 2, 3, 5, 6, and 7.

GJCO5: employs strategies 1, 2, 3, 4, 5, 6, and 7.

With regard to the last strategy, the function in equation (13) [6] will be used to illustrate gate reordering using GJCO1 and A1CO. The synthesis output of GJCO1 is 16 CNOT gates, shown in Figure 3.7, and 30 quantum gates, shown in Figure 3.8. The synthesis output of A1CO is 15 CNOT gates, shown in Figure 3.9, and 35 quantum gates, shown in Figure 3.10. Gate reordering on the GJCO1 circuit results in no change. After

gate reordering, the A1CO circuit in Figure 3.11 shows the 14th CNOT gate has moved to the 10th position, resulting in 33 quantum gates in Figure 3.12.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad (13)$$

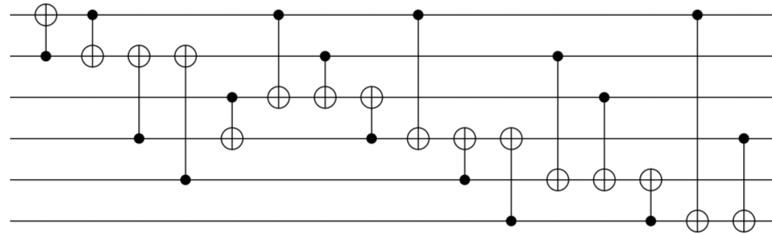

Figure 3.7. The 16-gate, GJCO1 linear reversible circuit realization of the function in (13).

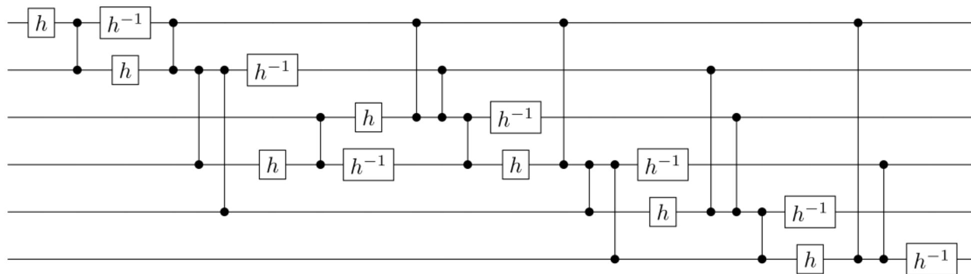

Figure 3.8. The 30-gate GJCO1 quantum linear reversible circuit realization of the function in (13).

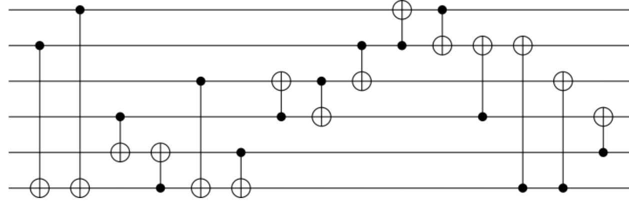

Figure 3.9. The 15-gate, A1CO linear reversible circuit realization of the function in (13).

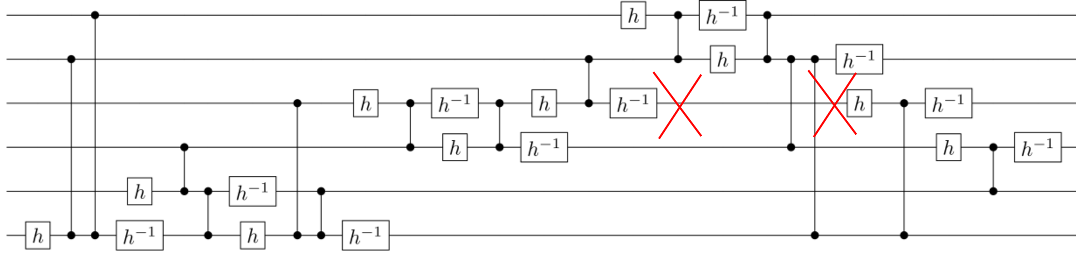

Figure 3.10. The 35-gate, A1CO quantum linear reversible circuit realization of the function in (13) which contains two redundant gates.

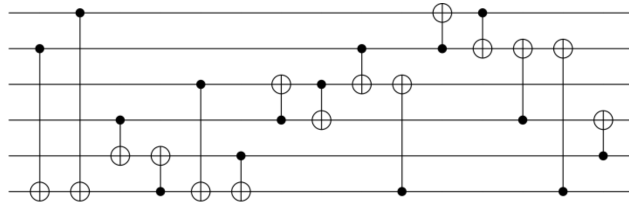

Figure 3.11. The 15-gate, A1CO linear reversible circuit realization of the function in (13) after gate reordering.

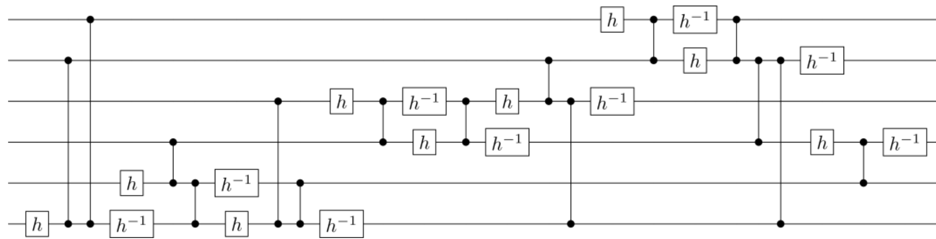

Figure 3.12. The 33-gate, A1CO quantum linear reversible circuit realization of the function in (13) after gate reordering.

For comparison the above methods will be tested with probabilistic variants of the AECM method and MCG method, named AECMP and MCGP respectively. In the AECM method and MCG method, whenever there are two or more subfunctions which yield equivalent cost reductions selection defaults to the first subfunction encountered. In the probabilistic variants subfunction selection is random.

3.6 Experimental Results

A set of tests similar to that used in [20] was performed on circuits with 8 to 100 lines; each set consisted of synthesising 100 randomized linear reversible circuits with multiple methods. The n-line circuit randomization function used $2n^2$ operations on the identity matrix, and each of these operations represented either a random distant CNOT gate or a random distant SWAP gate. AECMP and MCGP were run seven times per function to attempt to make a more fair comparison with the new methods which used multiple passes. In addition, MCGP testing was restricted to 24 or fewer lines due to long run times and known performance drop-off above this circuit dimension. All results were recorded in terms of both CNOT gate counts and, after gate reordering, quantum gate counts. Compared with the new methods, "Algorithm 1", AECMP, and MCGP were expected to produce lower CNOT gate counts but higher quantum gate counts.

The results for average CNOT gate counts and quantum gate counts are shown in Table 3.2 and Table 3.3 respectively. Figure 3.13 and Figure 3.14 illustrate the best performing methods using a normalized gate count. Examining the new methods in both CNOT gate counts and quantum gate counts, A1CO and GJCO1 performed poorly whereas GJCO2, GJCO3, GJCO4, and GJCO5 performed better. In circuits with more

than 40 lines GJCO2 was the most likely method to find the lowest CNOT gate count. In circuits with more than 16 lines GJCO4 was the most likely method to yield the lowest quantum gate count, although by 100 lines GJCO2 achieved similar performance. Table 3.4 shows the best-performing minimum subcolumn density parameters of GJCO4 for quantum gate counts. As the number of lines increased, GJCO4 performed best with a minimum subcolumn density parameter value of either four or five. Similar results were found in GJCO2, GJCO3, and GJCO5.

The quantum gate count reductions due to gate reordering ranged from 0 to 19.7%. Another benefit of gate reordering was that all pairs of identical CNOT gates that could be placed adjacent to each other were identified and deleted. This is a known issue with AECM-based methods (and indirectly MCG-based methods) [22]. No pairs of identical CNOT gates were discovered in the syntheses of any of the other methods.

Both AECMP and MCGP produced low CNOT gate counts for small circuits but also comparatively low quantum gate counts (following gate reordering) in circuits with 8 through 20 lines. This last result was unanticipated and indicates a possible direction for developing synthesis methods designed specifically for small quantum linear reversible circuits.

Table 3.2. Comparisons of linear reversible circuit synthesis methods (average CNOT gates).

Lines	A1CO	GJCO1	GJCO2	GJCO3	GJCO4	GJCO5	A1	AECMP	MCGP
8	27.87	31.5	25.56	25.39	25.33	25.29	28.01	20.3	19.38
12	60.37	71.55	54.83	55.02	54.03	54.57	62	42.84	39.47
16	103.32	129.46	93.22	93.39	91.71	92.58	107.97	74.54	68.75
20	158.72	197.68	138.71	140.81	136.81	138.73	164.64	115	107.66
24	222.4	286.1	193.8	195.75	192.28	193.54	234.16	166.47	160.33
28	294.2	391.94	254.61	257.81	253.63	256.41	317.06	230.33	
32	377.81	511.26	323.48	328.54	323.32	326.48	375.05	303.65	
36	469.87	646.85	399.24	404.76	400.27	405.09	469.57	392.37	
40	568.57	799.67	482.2	488.82	481.99	491.91	570.37	493.38	
44	679.37	969.66	573.8	580.09	578.56	583.6	681.85	606.02	
48	799.93	1154.9	669.28	676.56	672.7	682.97	800.54	731.67	
52	927.36	1353.6	772.15	781.86	777.84	789.66	929.18	875.63	
56	1066	1568.9	881.05	888.81	889.43	902.11	1068.7	1033.9	
60	1208.3	1794.9	991.91	1008.1	1003.5	1018.4	1216.1	1200.9	
64	1360.1	2045	1114.2	1128.7	1132.7	1147.6	1371.4	1380.5	
68	1516.9	2312.3	1241.5	1261.1	1260.6	1281	1542.2	1570.8	
72	1683.4	2590.8	1378	1394.5	1397.7	1420.8	1716.2	1790.1	
76	1854	2885.5	1512.8	1537.8	1541.7	1563.5	1901.4	2021.5	
80	2035.3	3202.4	1658	1680	1687.4	1708.1	2096.3	2254.6	
84	2220	3526.6	1808.4	1835.2	1845.3	1871.6	2300.3	2507.9	
88	2412.8	3875	1965	1990	2005.3	2042.2	2513.7	2774.7	
92	2615.9	4230.8	2129.2	2158	2168	2208	2737.1	3057.2	
96	2825.3	4611.8	2291.8	2326.3	2341.3	2383.7	2971.3	3351.4	

100	3041.6	5000.2	2466.4	2498.5	2521.1	2559.5	3210.9	3653.8

Table 3.3. Comparisons of quantum linear reversible circuit synthesis methods (average quantum gates).

Lines	A1CO	GJCO1	GJCO2	GJCO3	GJCO4	GJCO5	A1	AECMP	MCGP
8	59.14	53.54	51.45	51.13	50.7	50.53	63.53	50.43	48.99
12	116.72	106.07	102.05	101.54	98.74	98.92	130.98	100.62	97.12
16	189.64	175.64	168.47	166.74	161.67	161.66	215.61	169.11	164.43
20	281.19	255.52	243.85	244.21	235.79	237.59	315.52	254.34	251
24	383.59	356.32	337.59	340.48	325.7	327.3	433.4	359.54	364
28	496.23	474.62	442.07	445.96	427.4	430.34	569.82	492.64	
32	626.14	605.76	557.18	564.49	540.85	545.54	697.37	640.09	
36	763.09	752.61	684.49	695.69	666.03	670.9	862.11	815.55	
40	914.68	917.49	823.1	839.15	802.75	812.27	1031.9	1016.9	
44	1071	1100.8	972.61	988.4	954.24	961.99	1223.5	1243.6	
48	1245.2	1296.6	1131.1	1152.2	1109.2	1123.7	1412.4	1491.8	
52	1421.3	1507.5	1302.5	1324	1278.8	1294.5	1629.1	1771.8	
56	1615.6	1735.4	1480.5	1504.9	1458.6	1471.7	1854.8	2085.7	
60	1812.9	1973.2	1666.1	1700.3	1643.2	1666.4	2090.2	2416.1	
64	2017.3	2234.6	1864.5	1899.9	1844.9	1870.3	2334.5	2765.1	
68	2229.1	2514.4	2078.3	2118.2	2055.2	2084.1	2611.1	3141.1	
72	2456.7	2805.9	2296.8	2337.3	2272.9	2305.9	2889.2	3576.8	
76	2690.5	3112.2	2522	2567.6	2502.9	2535.9	3181.6	4030	
80	2932.9	3440.4	2758.1	2807.6	2738.2	2772.7	3477.1	4485.9	
84	3183.9	3776.8	3005.3	3056.6	2993.7	3031.4	3801.1	4981.7	
88	3449.2	4136.7	3266	3322.5	3249.8	3299.8	4133.2	5505.1	

92	3724.6	4503.7	3528.3	3589.1	3511.1	3563.9	4471.1	6055.5
96	4003.6	4898.9	3799.5	3866.3	3790.1	3850.2	4842.6	6628.9
100	4289.6	5297.4	4077	4150.4	4074	4132.6	5192.6	7217.4

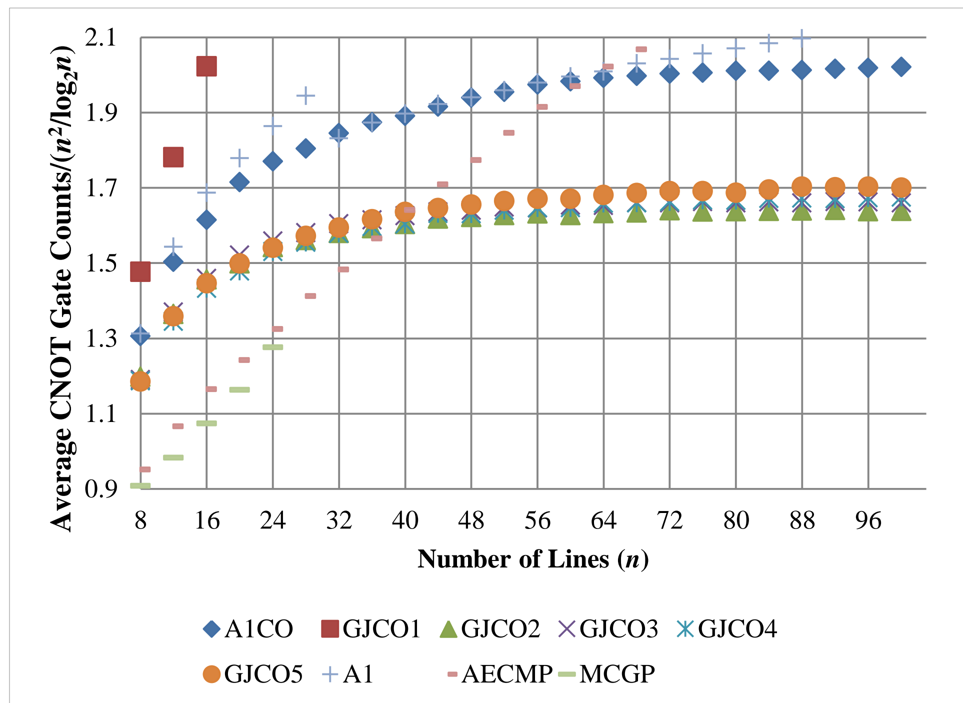

Figure 3.13. The average CNOT gate counts of synthesizing 100 randomized linear reversible circuits with multiple methods.

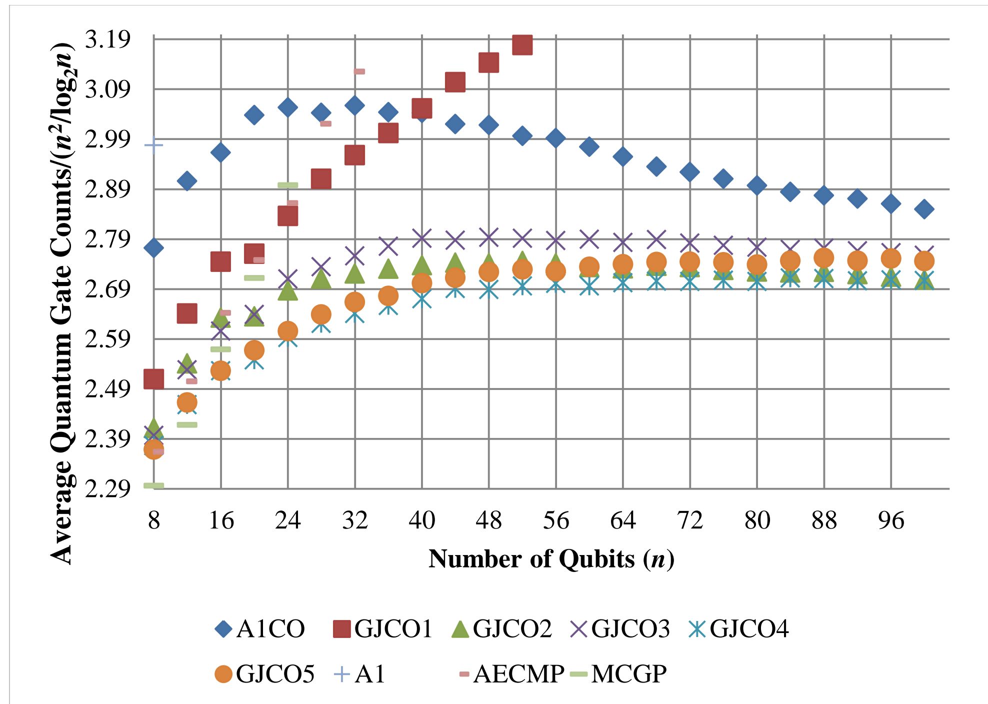

Figure 3.14. The average quantum gate counts of synthesizing 100 randomized linear reversible circuits with multiple methods.

Table 3.4. Best performing minimum density subcolumn parameter (*dmin*) frequencies corresponding to quantum gate counts produced by the GJCO4 method.

Lines	*dmin=1*	*dmin=2*	*dmin=3*	*dmin=4*	*dmin=5*	*dmin=6*	*dmin=7*
8	29	33	19	14	5	0	0
12	12	21	30	18	12	6	1
16	0	19	37	20	17	7	0
20	0	9	25	33	18	15	0
24	0	3	27	35	20	12	3
28	0	0	21	47	23	8	1
32	0	1	13	50	29	6	1
36	0	0	20	47	28	3	2
40	0	0	15	47	29	9	0
44	0	0	7	65	25	2	1
48	0	0	3	63	33	1	0
52	0	0	4	64	30	1	1
56	0	0	6	61	27	6	0
60	0	0	2	59	34	5	0
64	0	0	2	55	39	4	0
68	0	0	2	64	29	5	0
72	0	0	1	63	35	1	0
76	0	0	1	66	32	1	0
80	0	0	0	54	46	0	0
84	0	0	0	61	37	2	0
88	0	0	1	63	36	0	0
92	0	0	0	68	32	0	0
96	0	0	0	69	31	0	0
100	0	0	0	60	39	1	0

3.7 Conclusion

Strategies for improving both CNOT and quantum gate counts in linear reversible circuits were introduced and tested in several synthesis methods. Best performance depended on the number of lines, and whether CNOT or quantum gate counts were of chief importance. For circuits of 40 lines or more, the best performing methods used

elementary column operations, a Gauss-Jordan Elimination approach, variable-width subcolumn eliminations, multiple passes, and gate reordering. For circuits near 100 lines, the best results asymptotically approached approximately $1.64n^2/\log_2 n$ CNOT gates and $2.71n^2/\log_2 n$ quantum gates. The results also indicated that the Algorithm 1 and A1CO methods, which are based closely on 4RMI, were regularly outperformed by other methods; this is likely a consequence of using fixed-width subrow/subcolumn eliminations and permitting subrow/subcolumn eliminations on matrix columns which are sparse.

Among the tested methods GJCO4 produced the minimum quantum gate count most often. The GJCO3 and GJCO5 methods selected subcolumn eliminations based on cost, but they often produced slightly higher CNOT and quantum gate counts than the simpler GJCO2 method produced. This seems to parallel previous results [22] in which cost-based methods performed worse as n increased.

4

EXOR-AND-Based Logic Synthesis

4.1 Overview

The idea for EPOE synthesis came from analysis of the following function:

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

Figure 4.1. A logic function used to discuss redundant cubes in SOP minimization.

This function has been used frequently in Portland State University Digital Electronics I courses as a means to discuss redundant cubes. Often students synthesize the above function as the five–cube expression $bd \vee abc \vee a\bar{c}d \vee \bar{a}b\bar{c} \vee \bar{a}cd$. However, the four–cube expression $abc \vee a\bar{c}d \vee \bar{a}b\bar{c} \vee \bar{a}cd$ is a cover for the function, and the cube bd is redundant.

Another issue about this function is how to efficiently synthesize an expression for quantum computing. Because the expression $abc \vee a\bar{c}d \vee \bar{a}b\bar{c} \vee \bar{a}cd$ is a DSOP expression, it can be used as an ESOP expression. Considering that Toffoli quantum costs go up exponentially as the number of inputs increases, the author used an ad hoc greedy-search approach to generate the expression $d \oplus (a \oplus c \oplus 1)(d \oplus b \oplus 1)$. This latter expression is a type of factored ESOP expression which can be realized in a reversible circuit without ancilla lines; more specifically, it can be realized using one Toffoli gate

and several low-cost CNOT gates. Further, the latter expression consists of five literals which makes it intriguing for classical logic synthesis as well. Thus began an investigation into practical ways to synthesize logic functions into these types of expressions.

In order to formalize the greedy-search approach, the concept of an approximate cover (or fractional cover) and its converse the approximate implicant (or fractional implicant) was developed. For simplicity of implementation and to guarantee convergence, in EPOE synthesis the approximate covering criterion equation restricts POE expressions to those which cover more ON-set minterms that OFF-set minterms. Arguably this approach is a compromise. While it is true that ESOP expressions can include cubes which are completely in the OFF-set, the concern in EPOE synthesis was that permitting such flexibility would lead to a significant increase in the search space.

There are two challenges in using an approximate cover criterion. The first challenge is how best to set the value of w. For EPOE synthesis the following analysis yielded a "best guess" value of $w = {}^{2}/_{3}$ [13].

> … two similar functions, F_1 and F_2 shown in Figure 4.2 and Figure 4.3 respectively, will each be synthesized twice and judged strictly on AND gate counts. The function F_1 is synthesized in Figure 4.2a using a greater than 2/3 fractional criterion. Under this restriction the most efficient covering selection is comprised of disjoint groups cef and $ac'def$, realized by one three-input AND gate and one five-input AND gate. This results in the disjoint ESOP $F_1 = cef \oplus ac'def$. In contrast in Figure 4.2b the 2/3 covering criterion is violated.

Consequently the synthesis in Figure 4.2b has more steps and is realized by one two-input AND gate, one three-input AND gate, and one five-input AND gate. This results in the non-disjoint ESOP $F_1 = ef \oplus c'ef \oplus ac'def$. Violating the 2/3 covering criterion in the choice of product ef leads to an additional AND gate.

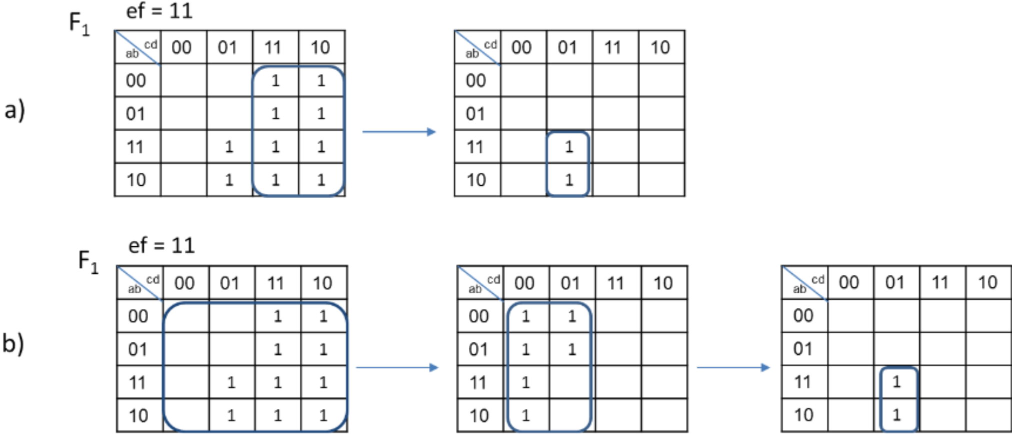

Figure 4.2. The Karnaugh maps of alternative ESOP syntheses of the function F_1: a) for expression $cef \oplus ac'def$ and b) for expression $ef \oplus c'ef \oplus ac'def$.

The function F_2 is synthesized in Figure 4.3a using a greater than 2/3 covering criterion. Under this restriction the most efficient covering selection is comprised of non-disjoint groups ef, $c'd'ef$, and $a'b'c'def$ which requires one two-input AND gate, one four-input AND gate, and one six-input AND gate. This results in the non-disjoint ESOP $F_2 = ef \oplus c'd'ef \oplus a'b'c'def$. In contrast in Figure 4.3b the 2/3 covering criterion is violated. Consequently the synthesis in Figure 4.3b has more steps and is realized by one two-input AND gate, one three-input AND gate, one four-input AND gate, and one six-input AND gate. This results in the

non-disjoint ESOP $F_2 = ef \oplus c'ef \oplus c'def \oplus a'b'c'def$. Again, violating the 2/3 covering criterion in the choice of product $c'ef$ leads to an additional AND gate.

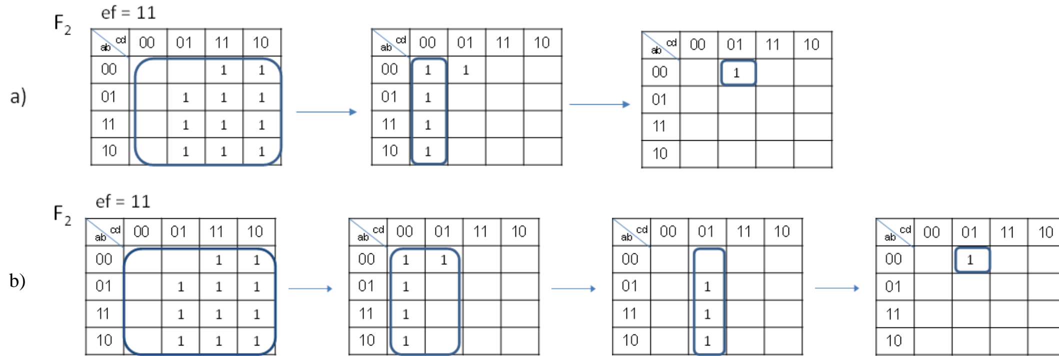

Figure 4.3. The Karnaugh maps of alternative ESOP syntheses of function F_2: a) for expression $ef \oplus c'd'ef \oplus a'b'c'def$ and b) for expression $ef \oplus c'ef \oplus c'def \oplus a'b'c'def$.

Based on the above examples, the threshold value for the fractional covering criterion occurs at the point when the AND gate with the smallest number of inputs performs an asserting role and all subsequent AND gates, each of which covers one-fourth the number of minterms as compared to the prior gate, perform an inhibiting role. Under these conditions the total number of asserted minterms is $m = 2^k - (2^{k-2} + 2^{k-4} + \cdots + 2^0)$ for some even integer k. Based on this threshold, expressions must satisfy a $lim_{k\to\infty}\left(\frac{m}{2^k}\right) = \frac{2}{3}$ covering criterion in order to be acceptable.

In practice, the value $w = {}^{2}/_{3}$ is more likely to produce efficient synthesis of reversible circuits for quantum computing than classical digital circuits. This appears to be due to the increased complexity of real-world functions as compared to the above functions.

The second challenge in using an approximate cover criterion is how to efficiently group together sets of ON-set minterms for analysis. A spectral approach was adopted for the synthesis methods described herein. Using the Hadamard transform permits quick analysis of the number of minterms in restricted subfunctions $|(f|_{\bar{x}_i})|$ and $|(f|_{x_i})|$, but more importantly it permits quick analysis of the number of minterms in restricted subfunctions $|(f|_{\bar{y}_i})|$ and $|(f|_{y_i})|$ where y_i is an affine-linear function of one or more x_i variables. Excluding the functions $f = 0$ and $f = 1$, all functions will have some form of a bias (i.e., $|(f|_{\bar{x}_i})| \neq |(f|_{x_i})|$ for at least one x_i input variable). POE expressions are generated by iteratively selecting spectral coefficients with the largest magnitude, using the sign of these coefficients to select restricted subfunctions [12], and then recomputing spectral coefficients until an approximate implicant is detected.

The following example illustrates spectral coefficient selection for the four-minterm function in Figure 4.4. There are two coefficients with the same maximum absolute value: the value $S_{c\oplus d} = -4$ indicates $|(f|_{c\oplus d})| = ½(S_0 - S_{c\oplus d}) = ½(6 - (-4)) = 5$, and the value $S_{a\oplus b\oplus c\oplus d} = 4$ indicates $|(f|_{a\oplus b\oplus c\oplus d\oplus 1})| = ½(S_0 + S_{a\oplus b\oplus c\oplus d}) = ½(6 + 4) = 5$. Iterating the same analysis on the restricted subfunctions $|(f|_{a\oplus b\oplus c\oplus d\oplus 1})|$ and $|(f|_{c\oplus d})|$ leads to one of the following equivalent POE expressions: $(a \oplus b \oplus c \oplus d \oplus 1)(c \oplus d)$, $(a \oplus b \oplus c \oplus d \oplus 1)(a \oplus b)$, or $(c \oplus d)(a \oplus b)$. One weakness of this approach is that some potentially good coefficients, such as $S_{a\oplus b}$, are initially overlooked.

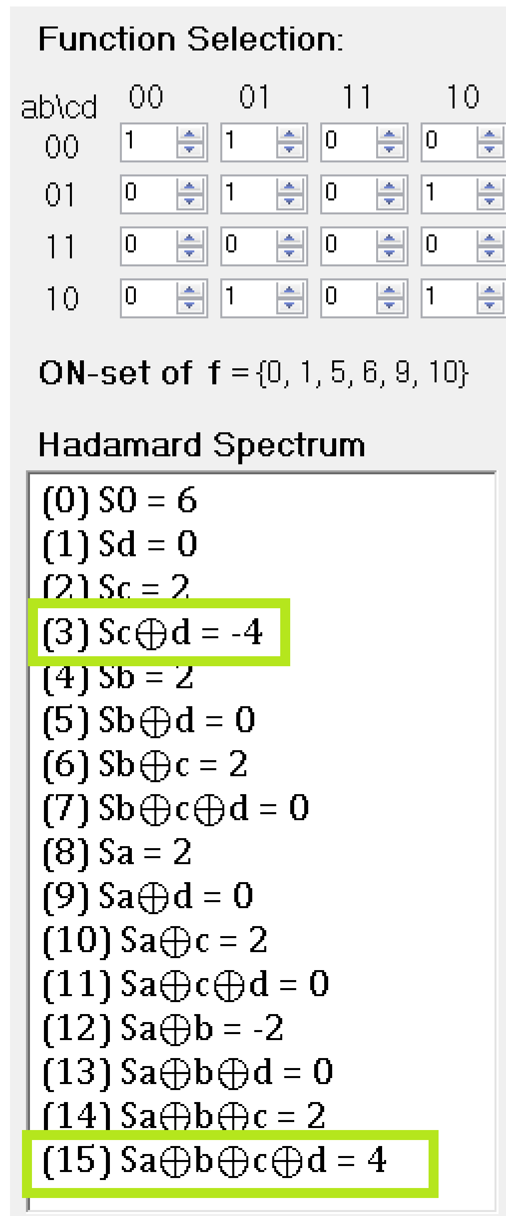

Figure 4.4. The Karnaugh map and corresponding spectral coefficients of an example 4×1 switching function.

4.2 Theoretical Comparisons with Other Approaches

A GF(2) matrix representation for POE expressions was chosen because of its applicability to reversible circuits. The idea was to use CNOT gates and NOT gates intensively and reduce the number of Toffoli gates, and then determine if it had applications in classical logic. The mathematical approach essentially is the same one employed by Meinel and Theobald [9], Günther and Drechsler [10], and Karpovsky, Stanković, and Astola [11] in their works for classical logic.

Later it was discovered that the expressions were compatible with the pseudoproduct EXOR-AND formulation of Luccio and Pagli [8]. However, their formulation is less directly applicable to the synthesis of reversible and quantum circuits. For instance, a pseudoproduct is defined as the canonical expression of a pseudocube, a pseudocube being an explicit list of ON-set minterms with particular properties. A POE expression is defined quite differently [12], and has no requirement for examination of an explicit list of minterms. Furthermore, in EPOE synthesis a POE expression can be an approximate implicant of a function which cannot be constructed from an explicit list of ON-set minterms. Another fundamental difference is the concept of canonical and noncanonical variables within a pseudoproduct. This concept does not have a parallel in POE expressions. Lastly, unlike pseudoproducts, POE expressions are thought of as having a number of circuit realizations (i.e., $0.29 \times 2^{p^2}/p!$ different circuit realizations of any POE expression c which is a product of p entries of $\vec{y}$ [12]). In quantum logic having a large number of circuit realizations facilitates the synthesis of groups of linear reversible circuits separated by Toffoli gates.

Further research into the pseudoproduct formulation clarified that Ishikawa et al. first synthesized EXOR-AND-EXOR (ESPP) circuits [7]. They used an algorithm similar to cube calculus methods used in ESPRESSO-II [14], EXMIN2 [24], and EXORCISM4 [25]. Their results were superior for a number of small functions. Unfortunately they did not publish their synthesis times, so it is unknown what the performance-to-time tradeoff of their algorithm is. Further, Ishikawa et al. did not investigate reversible and quantum computing applications of their work.

4.3 Introduction to XAX

A proof-of-concept program named XAX developed by the author employs three algorithms, the first for EPOE synthesis of completely specified functions, the second for EPOE synthesis of incompletely specified functions, and the third for SPOE synthesis of incompletely specified functions [12]. The second and third algorithms employ a second spectrum in order to assess the number of don't-care minterms in a restricted subfunction; i.e., the spectrum S_{DC} which is the Hadamard transform of f_{DC}, and spectrum $S_{ON \vee DC}$ which is the Hadamard transform of $f_{ON} \vee f_{DC}$. Considering the third algorithm for SPOE synthesis used on completely specified functions, i.e., incompletely specified functions with an empty DC-set, the second spectrum is still required to both ensure that POE expressions may intersect one another and that each POE expression covers as many ON-set minterms as possible.

All algorithms compute POE expressions through spectral coefficients in restricted subfunctions; i.e., $f\big|_{y_0 y_1 y_2 \ldots}$ expressions in which each y_i expression corresponds to a spectral coefficient which has a maximum absolute value. When there

are two or more spectral coefficients with the same maximum absolute value, the first coefficient found is selected by default. Alternatively selection can be performed randomly or based on the coefficient associated with the fewest input variables (which is known as greedy selection in XAX). These selection modes can be combined with a policy of excluding y_i expressions from consideration when they contain more than a user-specified number of input variables.

The coefficient selection mode is one of three algorithmic decisions which affect POE expression construction. The second algorithmic decision is the choice to discard all unused spectral information at each iteration, i.e., discarding the complement spectra corresponding to $f\big|_{\bar{y}_0}, f\big|_{y_0\bar{y}_1}, f\big|_{y_0 y_1 \bar{y}_2 \ldots} \ldots$. Discarding complement spectra simplifies the algorithm; although retaining and using this information may be viable, it is beyond the scope of the methods presented herein. The third algorithmic decision is the selection of which input variable to exclude in restricted subfunctions whenever y_i contains more than one x_i input variable. Depending on the function, changing this x_i-exclusion policy may have no effect on synthesis or change the cost by a small percentage. By default, the input variable with the highest index is selected. These algorithmic decisions would appear to pose a considerable challenge to developing variants of these algorithms which perform backtracking.

For ease of programming [12], these algorithms encode vector elements in little endian format; i.e., a format in which the least significant bit in mathematical notation is stored at the lowest order memory location. Consequently a minterm is encoded as $\vec{x} \rightarrow 2^{n-1}x_{n-1} + 2^{n-2}x_{n-2} + 2^{n-3}x_{n-3} \ldots + 2^0 x_0$; spectral coefficients are of the form

$[S_0, S_{x_0}, S_{x_1}, S_{x_0 \oplus x_1} \dots, S_{x_0 \oplus \dots x_{n-3} \oplus x_{n-2} \oplus x_{n-1}}]$. In these algorithms the vector $\vec{y}$ is decomposed into three components: rows of the matrix M as $y.m[]$, elements of the vector $\vec{b}$ as $y.b[]$, and number of p factors in $y_0 y_1 y_2 \dots y_{p-1}$ as $y.p$. For example, the three elements $y.m[0] = 0011_2$ and $y.m[1] = 1100_2$, $y.b[0] = 0$ and $y.b[1] = 1$, and $y.p = 2$ represent the POE expression $y_0 y_1 = (x_0 \oplus x_1)(x_2 \oplus x_3 \oplus 1)$.

It is possible to reduce EPOE and SPOE synthesis time for degenerate and autosymmetric functions [17, 18]. Specifically, using spectral methods it is possible to detect if there exists one or more y_i expressions such that $f\big|_{y_i} = f\big|_{\overline{y_i}}$. In the worst case, the search for degenerate and autosymmetric functions will yield nothing except an increase in synthesis time. Consequently detection of degenerate and autosymmetric functions is not integrated into XAX.

4.4 XAX EPOE Synthesis of Completely Specified Functions

The XAX EPOE synthesis program generates one EPOE structure per switching-function output, and this EPOE structure consists of a polarity bit and a list of POE cubes. Synthesis proceeds as follows: First, a single-output switching function f and approximate cover criterion w in the range $0.5 < w < 1$ are passed to the synthesis function. Prior to synthesis, if $|f| < w \times 2^n$ is satisfied then the remainder function f_{rem} is assigned as $f_{rem} \leftarrow f, polarity \leftarrow 0$; otherwise $f_{rem} \leftarrow f \oplus 1, polarity \leftarrow 1$. If the function is incompletely specified, all DC-minterms are treated as OFF-set minterms. Then the main synthesis subprogram, shown below, is called.

$S \leftarrow \mathrm{FHT}(f_{rem})$ //Fast Hadamard Transform

while $S[0] > 1$

$y.p \leftarrow 0$ //prepare to compute y_0

CalcGate1(S, w, y) //i.e., $y_0\ y_1\ y_2 \dots\ y_{p-1}$

EPOEList.Add(y)

$f_{rem} \leftarrow f_{rem} \oplus y$

$S \leftarrow$ FHT(f_{rem})

if $S[0] = 1$ then

EPOEList.Add(CalcMintermGate(f_{rem}))

Extending [12], the CalcGate1 function begins by searching for the largest magnitude spectral coefficient, denoted as $S[z]$ where z is an integer in the range $0 < |z| < 2^n$. Once z is selected, the value y_p is set to z according to the sign of z; i.e., $y.m[y.p] \leftarrow z$ and either $y.b[y.p] \leftarrow 1$ when $S[z] > 0$ (meaning the restricted function $f\big|_{\overline{y}_p}$ covers the most minterms) or $y.b[y.p] \leftarrow 0$ when $S[z] \leq 0$ (meaning the restricted function $f\big|_{y_p}$ covers the most minterms). Then the variable $y.p$ is incremented. This process creates an implicit POE cube $c = y_0 y_1 y_2 \dots y_{p-1}$. If c covers $w \times 2^{n-1}$ minterms then the CalcGate1 function returns, otherwise a new spectrum S' is computed. The spectrum of the restricted function, S', corresponds to a switching function which has been reduced to dimension $(n-1)\times 1$. Using a reduced dimension function reduces computation time but requires that variables be remapped from S to S'. The CalcGate1 function is called recursively, and after each recursion all variable mappings in $y_{i+1}, y_{i+2}, y_{i+3}$ ... are restored through bitwise manipulations. In the following CalcGate1

algorithm the operators ∨, ∧, and ⊕ denote bitwise-OR, bitwise-AND, and bitwise-Exclusive-OR operations respectively:

Function CalcGate1(S, w, y):

$z \leftarrow 1$ //little endian encoding for $y_i = x_0$

for each j from 2 to $S.Length - 1$

    if Abs($S[j]$) > Abs($S[z]$) then

        $z \leftarrow j$

$y.b[y.p] \leftarrow (S[z] > 0)$

$y.m[y.p] \leftarrow z$ //a binary encoded EXOR sum

$y.p \leftarrow y.p + 1$ //a POE of $y.p$ elements of $\vec{y}$

$plocal \leftarrow y.p$

if $(S[z] > 0)$ then

    $coveredminterms \leftarrow (S[0] + S[z])/2$

else

    $coveredminterms \leftarrow (S[0] - S[z])/2$

if $coveredminterms > w \times S.Length/2$ then

    return

//compute restricted subfunction

$S' \leftarrow$ new integer array of length $S.Length/2$

$S'[0] \leftarrow coveredminterms$

$variablebit \leftarrow$ ShiftLeft(1, $Integer(Log2(z))$)

$maskbitsL \leftarrow variablebit - 1$

$maskbitsH \leftarrow OnesComplement(maskbitsL)$

$k \leftarrow 1$

if $(S[z] > 0)$ then

    //compute spectrum of $f\big|_{\bar{y}_{p-1}}$

    for each j from 1 to $S.Length - 1$

        if $(j \wedge variablebit = 0)$ then

            $S'[k] \leftarrow (S[j] + S[j \oplus z])/2$

            $k \leftarrow k + 1$

else //compute spectrum of $f\big|_{y_{p-1}}$

    for each j from 1 to $S.Length - 1$

        if $(j \wedge variablebit = 0)$ then

            $S'[k] \leftarrow (S[j] - S[j \oplus z])/2$

            $k \leftarrow k + 1$

CalcGate1(S', w, y)

//upon return $y.p$ will be increased

//remap new $y.m$ expressions

for each j from $plocal$ to $y.p - 1$

    $y.m[j] \leftarrow (maskbitsL \wedge y.m[j]) \vee$ ShiftLeft($maskbitsH \wedge y.m[j]$, 1)

return

4.5 XAX EPOE Synthesis of Incompletely Specified Functions

The algorithm for EPOE synthesis of incompletely specified functions is similar to the above algorithm with these differences: initially f_{ON} is analyzed rather than f, a second spectrum representing the DC-set is required, and the remainder function computation ignores DC-minterms.

$S \leftarrow$ FHT(f_{rem}) //Fast Hadamard Transform

$S_{DC} \leftarrow$ FHT(f_{DC})

while $S[0] > 1$

    $y.p \leftarrow 0$ //prepare to compute y_0

    CalcGate2(S, S_{DC}, w, y)

    EPOEList.Add(y)

    $f_{rem} \leftarrow f_{rem} \oplus y(f_{ON} \vee f_{OFF})$

    $S \leftarrow$ FHT(f_{rem})

if $S[0] = 1$ then

    EPOEList.Add(CalcMintermGate(f_{rem}))

The CalcGate2 algorithm extends the CalcGate1 algorithm with a preference for restricted functions which cover the DC-set. A helper function, below, computes the number of DC-minterms covered by either $f\big|_{y_i}$ or $f\big|_{\overline{y}_i}$ depending on which of these restricted functions covers the most 1-minterms.

Function CoveredMinterms (S_a, S_b, z):

if $(S_a[z] \ > 0)$ then

    return $(S_b[0] + S_b[z])/2$

else

    return $(S_b[0] - S_b[z])/2$

Function CalcGate2$(S, S_{DC}, w,$ y):

$z \leftarrow 1$ //encoding for $y_i = x_0$

$DCminterms =$ CoveredMinterms(S, S_{DC}, z)

for each j from 2 to $S.Length - 1$

    if (Abs($S[j]$) > Abs($S[z]$)) OR (Abs($S[j]$) = Abs($S[z]$) AND

        $DCminterms <$ CoveredMinterms(S, S_{DC}, z))

    then

        $z \leftarrow j$

        $DCminterms \leftarrow$ CoveredMinterms(S, S_{DC}, z)

$y.b[y.p] \leftarrow (S[z] > 0)$

$y.m[y.p] \leftarrow z$ //a binary encoded EXOR sum

$y.p \leftarrow y.p + 1$ //a POE of $y.p$ elements of $\vec{y}$

$plocal \leftarrow y.p$

$covered1minterms \leftarrow$ CoveredMinterms(S, S, z)

$covered0minterms \leftarrow S.Length/2 - covered1minterms - DCminterms$

if $covered1minterms > w \times (covered0minterms + covered1minterms)$ then

return

//compute restricted subfunction

$S' \leftarrow$ new integer array of length $S.Length/2$

$S'[0] \leftarrow covered1minterms$

$S_{DC}' \leftarrow$ new integer array of length $S.Length/2$

$S_{DC}'[0] \leftarrow DCminterms$

$variablebit \leftarrow$ ShiftLeft(1, $Integer(Log2(z))$)

$maskbitsL \leftarrow variablebit - 1$

$maskbitsH \leftarrow OnesComplement(maskbitsL)$

$k \leftarrow 1$

if $(S[z] > 0)$ then

//compute spectrum of $f\big|_{\overline{y}_{p-1}}$

for each j from 1 to $S.Length - 1$

if $(j \wedge variablebit = 0)$ then

$S'[k] \leftarrow (S[j] + S[j \oplus z])/2$

$S_{DC}'[k] \leftarrow (S_{DC}[j] + S_{DC}[j \oplus z])/2$

$k \leftarrow k + 1$

else //compute spectrum of $f\big|_{y_{p-1}}$

for each j from 1 to $S.Length - 1$

if $(j \wedge variablebit = 0)$ then

$S'[k] \leftarrow (S[j] - S[j \oplus z])/2$

$S_{DC}'[k] \leftarrow (S_{DC}[j] - S_{DC}[j \oplus z])/2$

$k \leftarrow k + 1$

CalcGate2($S', {S_{DC}}', w, y$)

//upon return $y.p$ will be increased

//remap new $y.m$ expressions

for each j from $plocal$ to $y.p - 1$

$y.m[j] \leftarrow (maskbitsL \wedge y.m[j]) \vee$ ShiftLeft($maskbitsH \wedge y.m[j]$, 1)

return

4.6 XAX SPOE Synthesis of Incompletely Specified Functions

Extending [12], there is one XAX SPOE synthesis algorithm for both completely and incompletely specified functions. The algorithm for SPOE synthesis is similar to the EPOE algorithms with these differences: initially f_{ON} is analyzed rather than f, a second spectrum representing the union of the ON-set and DC-set is used, all POE expressions are implicants of the union of the ON-set and DC-set, and the remainder function computation creates a logical sum of POE expressions.

$S \leftarrow$ FHT(f_{rem}) //Fast Hadamard Transform

$S_{ON \vee DC} \leftarrow$ FHT($f_{ON \vee DC}$)

while $S[0] > 1$

$y.p \leftarrow 0$ //prepare to compute y_0

CalcGate3($S, S_{ON \vee DC}, y$)

SPOEList.Add(y)

$f_{rem} \leftarrow f_{rem}\bar{y}$

$S \leftarrow$ FHT(f_{rem})

if $S[0] = 1$ then

SPOEList.Add(CalcMintermGate(f_{rem}))

Function CalcGate3($S, S_{ON \vee DC}, y$):

$z \leftarrow 1$ //encoding for $y_i = x_0$

$ONDCminterms =$ CoveredMinterms($S, S_{ON \vee DC}, z$)

for each j from 2 to $S.Length - 1$

if (Abs($S[j]$) $>$ Abs($S[z]$)) OR (Abs($S[j]$) = Abs($S[z]$) AND

$ONDCminterms <$ CoveredMinterms($S, S_{ON \vee DC}, z$)) then

$z \leftarrow j$

$ONDCminterms \leftarrow$ CoveredMinterms($S, S_{ON \vee DC}, z$)

$y.b[y.p] \leftarrow (S[z] > 0)$

$y.m[y.p] \leftarrow z$ //a binary encoded EXOR sum

$y.p \leftarrow y.p + 1$ //a POE of $y.p$ elements of $\vec{y}$

$plocal \leftarrow y.p$

//return if y represents an implicant

if $ONDCminterms = S.Length/2$ then

return

$S' \leftarrow$ new integer array of length $S.Length/2$

$S'[0] \leftarrow$ CoveredMinterms(S, S, z)

$S_{ON \vee DC}{}'$ ← new integer array of length $S.Length/2$

$S_{ON \vee DC}{}'[0]$ ← $ONDCminterms$

$variablebit$ ← ShiftLeft(1, $Integer(Log2(z))$)

$maskbitsL \leftarrow variablebit - 1$

$maskbitsH \leftarrow OnesComplement(maskbitsL)$

$k \leftarrow 1$

if $(S[z] > 0)$ then

//compute spectrum of $f\big|_{\bar{y}_{p-1}}$

for each j from 1 to $S.Length - 1$

if ($j \wedge variablebit = 0$) then

$S'[k] \leftarrow (S[j] + S[j \oplus z])/2$

$S_{ON \vee DC}{}'[k] \leftarrow (S_{ON \vee DC}[j] + S_{ON \vee DC}[j \oplus z])/2$

$k \leftarrow k + 1$

else //compute spectrum of $f\big|_{y_{p-1}}$

for each j from 1 to $S.Length - 1$

if ($j \wedge variablebit = 0$) then

$S'[k] \leftarrow (S[j] - S[j \oplus z])/2$

$S_{ON \vee DC}{}'[k] \leftarrow (S_{ON \vee DC}[j] - S_{ON \vee DC}[j \oplus z])/2$

$k \leftarrow k + 1$

CalcGate3($S', S_{ON \vee DC}{}', y$)

//upon return $y.p$ will be increased

//remap new $y.m$ expressions

for each j from $plocal$ to $y.p - 1$

$y.m[j] \leftarrow (maskbitsL \wedge y.m[j]) \vee$ ShiftLeft($maskbitsH \wedge y.m[j]$, 1)

return

4.7 Postprocessing

Extending [12], after an EPOE or SPOE expression is computed it can be modified in two independent postprocessing stages. Both postprocessing stages employ LCC operations to transform POE cubes into a more optimal form. The first postprocessing stage minimizes the number of input variables in each POE cube. This requires examining 2^{p-1} forms of each y_i sum.

The second postprocessing stage minimizes the total number of unique y_i sums, ignoring polarity. The goal of this stage is to reduce the number of corresponding EXOR gates, although doing so will increase gate fan-out. First those unique y_i sums consisting of two or more input variables are stored in an initial sum list along with their respective number of POE references. Then, starting with those sums which contain the most input variables and ending with those sums which contain exactly two input variables, an attempt is made to transform all instances of a sum to any other sum in the initial sum list. Those sums which cannot be transformed are stored in a committed sum list which augments the initial sum list. One consequence of this refactoring is the creation of POE cube expressions which contain y_i sums of more than $n - p + 1$ input variables.

There are other options for postprocessing. For instance, it may be desirable to decompose some POE expressions into simpler forms when technology-specific costs can

be reduced. Another option is using LCC operations with GF(2) matrix inversion methods to generate a reduced row echelon form of a POE expression. Further, if two different POE expressions are functionally equivalent their reduced row echelon forms will be identical.

4.8 Complexity

Extending [12], Selezneva describes an ESPP decomposition method in which groups of $n+1$ Zhegalkin polynomials [26], i.e., $n+1$ cubes of a positive polarity Reed-Muller expansion, are factored into at most two low-complexity pseudoproducts. If these groups are created from Hamming codes for block sizes $n = \{3,7,15,31,\ldots\}$ [27], Selezneva's method results in an upper bound of $2^{n+1}/(n+1)$ POE cubes per switching function. Therefore, EPOE expressions can be viewed as having $O((2^{n^2}/n!)^{2^{n-\lceil \log_2 n+1 \rceil+1}})$ equivalent forms. The function $2^{n-\log_2(n+1)+1}$ is used for XAX POE-cube-list memory allocation and has been sufficient for all (default mode) EPOE and SPOE syntheses performed thus far.

In XAX EPOE synthesis of completely specified functions, each gate uses one $O(n2^n)$ FHT function call and recursive spectrum computations of order $O(2^n + 2^{n-1} + 2^{n-2} + \ldots)$; therefore gate computation is order $O(n2^n + 2^{n+1})$. In total the complexity of XAX synthesis is $O((n+2)2^{2n+1}/(n+1)) \approx O(2^{2n})$. In XAX EPOE and SPOE synthesis of incompletely specified functions, complexity increases by a power of two, yielding $O((n+2)2^{2n+2}/(n+1)) \approx O(2^{2n})$.

To calculate the postprocessing complexity requires a conservative estimate for p. In a XAX EPOE expression where $w = 2/3$, it holds that $p \leq n-2$ for all or all but one

of the POE cubes. The complexity of the first postprocessing stage is approximately $O((n-2)2^{2n-2}/(n+1)) \approx O(2^{2n})$; in practice this stage requires significantly less time than synthesis due to highly sequential memory accesses. The complexity of the second postprocessing stage is approximately $O((n-2)2^{3n}/(n+1)) \approx O(2^{3n})$; in practice this stage requires time which is closer to a linear increase over synthesis time.

4.9 Experimental Results

All three algorithms are integrated into the XAX program. The default mode of XAX performs EPOE synthesis of completely specified multi-output functions; the "don't care" EPOE mode, activated by using the command-line option "-d," performs EPOE synthesis of incompletely specified multi-output functions; and, the SPOE mode, activated by using the command-line option "-o," performs SPOE synthesis of incompletely specified multi-output functions. In the following experiments XAX synthesis of MCNC benchmarks [28] and Espresso benchmarks [29] is compared in a variety of modes. The XAX EPOE syntheses are run 16 times using the following approximate cover criterion values with the best result taken: 0.510417, 0.541667, 0.572917, 0.604167, 0.635417, 0.666667,0.697917, 0.729167, 0.760417, 0.791667, 0.822917, 0.854167, 0.885417, 0.916667, 0.947917, 0.979167.

The default output format of XAX is a single multi-output Verilog file. For example, the EPOE and SPOE syntheses of the well-known benchmark function "rd53" is shown below. In both syntheses the Verilog expression "`assign y3 = x[0]^x[1];`" defines the combination logic expression "`y3`" which is used to generate outputs `f[0]`, `f[1]`, and `f[2]`. In order to have a fair comparison with previous work

which did not share expressions across outputs, XAX includes the command-line option "-s" to generate a collection of single-output syntheses; i.e., all experiments except for Experiment 3 use XAX with the command-line option "-s".

```
//XAX EPOE synthesis of rd53, w = 0.510417
module rd53 (x, f);
input [4:0] x;
output [2:0] f;
wire c0;
wire c1;
wire c2;
wire c3;
wire c4;
wire c5;
wire y3;
wire y5;
wire y11;
wire y18;
wire y16;
wire y17;
wire y1f;
assign y3 = x[0]^x[1];
assign y5 = x[0]^x[2];
```

```
assign y11 = x[0]^x[4];

assign y18 = x[3]^x[4];

assign y16 = x[1]^x[2]^x[4];

assign y17 = y16^x[0];

assign y1f = y17^x[3];

assign c0 = x[0] & x[1] & x[2];

assign c1 = ~y18 & ~y16 & x[0];

assign c2 = y3 & y5 & ~y18 & y11;

assign c3 = y1f;

assign c4 = ~y3 & ~y5;

assign c5 = ~y18 & y17;

assign f[0] = c0^c1^c2;

assign f[1] = c3;

assign f[2] =  1 ^ c4^c5;

endmodule


//XAX SPOE synthesis of rd53

module rd53 (x, f);

input [4:0] x;

output [2:0] f;

wire c0;

wire c1;

wire c2;
```

```
wire c3;
wire c4;
wire c5;
wire c6;
wire y3;
wire y5;
wire ya;
wire yc;
wire y12;
wire y14;
wire y1c;
wire y1f;
assign y3 = x[0]^x[1];
assign y5 = x[0]^x[2];
assign ya = x[1]^x[3];
assign yc = x[2]^x[3];
assign y12 = x[1]^x[4];
assign y14 = x[2]^x[4];
assign y1c = x[2]^x[3]^x[4];
assign y1f = y3^y1c;
assign c0 = x[0] & x[1] & x[2] & x[3];
assign c1 = x[0] & x[1] & x[4] & yc;
assign c2 = x[2] & x[3] & x[4] & y3;
```

```
assign c3 = y1f;
assign c4 = y3 & yc;
assign c5 = y5 & y12;
assign c6 = ya & y14;
assign f[0] = c0|c1|c2;
assign f[1] = c3;
assign f[2] = c4|c5|c6;
endmodule
```

A common relative cost approximation for an SOP expression or ESOP expression is the literal count. Considering the above multi-level logic definitions, using a literal count fails to quantify expression sharing and fan-out costs. For instance, in the expressions $(x_0 \oplus x_1)x_2 \vee (x_0 \oplus x_1)x_3$ and $(x_0 \oplus x_4)x_2 \vee (x_0 \oplus x_1)x_3$ there are six literals; in terms of literal fcount there is no cost benefit for the prior expression sharing the Exclusive-OR sum $(x_0 \oplus x_1)$ in two products. Therefore, as in prior work [7, 18, 31] herein the main metric in the following synthesis experiments will be the total number of technology-independent AND/OR/NAND/NOR gate inputs (TI, or μ). In this metric, inverters are free and fan-out is unlimited.

Two refinements of this cost metric are the total number of gate inputs in a high-level CMOS implementation (μ_C) and the total number of gate inputs in a high-level FPGA implementation (μ_F). These names are used for historical reasons; it is important to note that the high-level FPGA cost may not correlate to cost in modern FPGA implementations. In the high-level CMOS metric there are no Exclusive-OR gate

primitives, so a two-input Exclusive-OR expression costs four using a four-input gate primitive of the form $x_0\overline{x}_1 \vee \overline{x}_0 x_1$. Consequently all k-input Exclusive-OR expressions cost $4(k-1)$; i.e., each k-input Exclusive-OR expression is factored into two-input Exclusive-OR expressions and implemented as a multi-level circuit consisting of four-input gates of the form $x_0\overline{x}_1 \vee \overline{x}_0 x_1$. In the high-level FPGA metric a two-input Exclusive-OR expression costs two. Reevaluating the above expressions, the expressions $(x_0 \oplus x_1)x_2 \vee (x_0 \oplus x_1)x_3$ costs $\mu_C = 4 + (2+2) + 2 = 10$ and $\mu_F = 2 + (2+2) + 2 = 8$; the expression $(x_0 \oplus x_4)x_2 \vee (x_0 \oplus x_1)x_3$ costs $\mu_C = (4+4) + (2+2) + 2 = 14$ and $\mu_F = (2+2) + (2+2) + 2 = 10$.

4.9.1 Experiment 1.

In the first experiment, shown in Table 4.1, XAX EPOE synthesis for completely specified functions is compared with the ESPP logic synthesizer of Ishikawa et al. [7]. The best XAX EPOE results used approximate cover criterion values ranging from 0.5104167 through 0.760417. Because the ESPP logic synthesizer was permitted to iterate continuously until a minimum set of pseudoproducts was computed, it was expected that their method would produce near-minimum EXOR-AND-EXOR costs. However, in several benchmark functions XAX performed better than ESPP, the difference being extreme in the case of “rd84”. This is a cause for concern, raising the question whether or not the ESPP logic synthesizer became trapped in a local minimum or had some sort of bug.

Function	#TI ESPP (CMOS)	#TI XAX -s (CMOS)	XAX CMOS %diff. from Post-processing	#TI ESPP (FPGA)	#TI XAX -s (FPGA)	XAX FPGA %diff. from Post-processing	XAX Time (s)
bw	380	613	14.40%	239	409	6.60%	1.249
clip	408	422	16.80%	324	237	12.20%	0.525
con1	57	56	0.00%	38	36	0.00%	0.192
inc	207	385	14.40%	121	260	6.50%	1.257
misex1	172	211	18.50%	125	142	9.00%	1.464
rd53	72	87	21.60%	46	45	13.50%	0.418
rd73	214	180	30.80%	122	78	23.50%	0.308
rd84	622	282	30.40%	306	131	22.50%	0.195
sao2	297	504	28.70%	219	347	15.40%	0.421
Z5xp1	82	272	8.40%	56	167	5.10%	0.747
Z9sym	203	143	36.70%	135	73	27.70%	0.236
z4	87	112	0.00%	57	58	0.00%	0.2
Geomean	181.7237	217.8252		117.8986	124.9878		0.458512

Table 4.1. Total gate inputs (#TI) for XAX EPOE and ESPP [7] synthesis of selected MCNC benchmark functions.

4.9.2 Experiment 2.

In the first experiment the incompletely specified function “bw” was treated as a completely specified function in which the DC-minterms were included in the OFF-set. In order to quantify the performance of the two XAX EPOE algorithms, the second experiment synthesized a number of incompletely specified functions (including “bw”) with the two XAX EPOE algorithms. As can be seen in Table 4.2, the CMOS cost percentage difference between the two algorithms varied widely, from near 0 to approximately 50%. This variation indicates that improvements were highly function dependent. Furthermore, because POE expression computation is heuristic, it is possible

for the completely specified EPOE synthesis algorithm to perform better than the incompletely specified EPOE synthesis algorithm.

Function	#TI XAX -s (CMOS)	#TI XAX -s -d (CMOS)	% diff.	#TI XAX -s (FPGA)	#TI XAX -s -d (FPGA)	% diff.	XAX -s Time (s)	XAX -s -d Time (s)
bw	613	581	5.20%	409	388	5.10%	0.83	0.62
ex1010	5964	5032	15.60%	3620	2416	33.30%	2.79	1.10
exp	954	870	8.80%	638	564	11.60%	0.91	1.73
inc	385	314	18.40%	260	207	20.40%	0.85	0.44
misex3c	3668	3221	12.20%	2544	2140	15.90%	10.89	11.56
pdc	4506	2339	48.10%	3522	1636	53.50%	63.80	52.59
spla	4837	4817	0.40%	3833	3793	1.00%	67.58	61.57
Geomean	1950.36	1617.63		1356.11	1054.86		5.06	4.10

Table 4.2. Total gate inputs (#TI) of both XAX EPOE synthesis methods of selected incompletely specified MCNC and Espresso benchmark functions.

4.9.3 Experiment 3.

In order to have a more modern test of the XAX EPOE algorithms, in the third experiment a number of benchmark functions of up to 26 primary inputs which had previously been synthesized with EXORCISM4 [25] were selected. All functions were synthesized again with EXORCISM4, and both the EXORCISM4 results and the XAX EPOE results were compared as one multi-output circuit; specifically, the function "spla" was synthesized with the XAX EPOE algorithm for incompletely specified functions and all others with the XAX EPOE algorithm for completely specified functions. This would allow the cube sharing across multiple outputs that occurs in EXORCISM4 to be compared with the Exclusive-OR-sum-of-input sharing across multiple outputs that occurs in the XAX EPOE algorithms. While it is possible for the XAX EPOE algorithms to share POE expressions across multiple outputs, the greedy-search algorithm rarely

finds such POE expressions; increasing POE expression sharing is a topic for future work.

The experiment 3 results in Table 4.3 show that the XAX EPOE algorithms performed much better on average, but in a number of specific functions EXORCISM4 performed better. While it is difficult to predict which method would produce the best results, some general observations would be that XAX EPOE synthesis tends to work better than ESOP with sparse functions, symmetric functions, and autosymmetric functions (i.e., those functions which can be GF(2) transformed into a simpler form). In the majority of functions EXORCISM4 took what appears to be exponentially less time than the XAX EPOE algorithms took, and this would seem to follow from a comparison of two-level logic with three-level logic; however, in a few benchmarks where EXORCISM4 was dealing with a relatively large cube list the time difference between two-level ESOP synthesis and three-level EPOE synthesis was reversed. One direction for future research would be to create an ESOP-postprocessing algorithm to bridge the gap between these two synthesis methods. The best XAX EPOE results used approximate cover criterion values ranging from 0.5104167 through 0.916667.

Function	#TI EX4	Time EX4	#TI EX4 -q16	Time EX4 -q16	#TI XAX -o -s	XAX Time (s)
add6	1354	0.05	1282	0.38	217	0.173
alu4	6097	0.61	5913	4.22	3766	0.747
cps	7974	0.19	7658	2.19	5500	1660.488
duke2	1745	0.02	1648	0.22	1918	149.53
ex5	2412	0.03	2370	0.31	1072	0.8
in7	498	0.001	478	0.04	545	838.912
misex3	9705	1.8	9586	16.94	10384	1.985
spla (dc)	6300	0.37	6132	4.29	4054	56.968
table3	4964	0.06	4925	0.85	5180	2.243
table5	4703	0.05	4653	0.68	5672	10.538
vg2	2616	0.04	2615	0.49	1605	815.177
Geomean	3306	0.0722	3217	0.863	2310	17.492

Table 4.3. Total CMOS gate inputs (#TI) for EXORCISM4, EXORCISM4 with quality=16, and XAX EPOE/EPOE with don't cares using multi-output synthesis of selected Espresso benchmark functions.

4.9.4 Experiment 4.

Because the XAX program has numerous command line options to control POE expression selection, the fourth experiment was devised as a tournament to see which options performed best. The XAX EPOE algorithm for completely specified functions was used for all tests, and the results were generated as a collection of single output syntheses. Three modes were compared, the default mode, the greedy mode, and a random mode. Within both the default mode and greedy mode tests, the maximum number of input variables in each y_i sum was either one, two, three, or unrestricted. In the random mode tests whenever two y_i expressions were found to cover an equivalent number of minterms, selection would change from the y_i expression found first to the one

found later based on command-line options: “-r 0” denotes changing selection $1/n$ of the time, “-r 1” denotes changing selection $1/2n$ of the time, “-r 2” denotes changing selection $1/4n$ of the time, and “-r 3” denotes changing selection $1/8n$ of the time,.

The CMOS cost results for experiment 4 are shown in Table 4.4. The best performing mode was the greedy mode using an unrestricted number of input variables in each y_i sum, although several other modes performed similarly. At the most extreme the default mode results when using an unrestricted number of input variables in each y_i sum had 8.39% more gate inputs than the best-performing mode. The relative performance of the different modes was similar considering FPGA costs.

	Default Mode				Greedy Mode				Random Mode					
Function	-i 1	-i 2	-i 3	without -i	-i 1	-i 2	-i 3	without -i	-r 0	-r 1	-r 2	-r 3	Best	% diff.
bw	782	654	613	613	782	654	613	613	628	630	608	613	608	0.82%
clip	1869	566	411	423	1869	602	411	411	437	431	423	424	411	2.84%
con1	66	56	56	56	66	56	56	56	56	56	56	56	56	0.00%
inc	443	387	387	387	443	391	387	387	393	394	378	387	378	2.33%
misex1	250	215	211	211	250	215	211	211	241	227	211	211	211	0.00%
rd53	261	135	103	87	261	135	103	87	82	87	87	82	82	5.75%
rd73	1367	548	323	180	1367	558	323	168	208	196	198	192	168	6.67%
rd84	2895	1096	653	282	2895	1148	582	281	328	309	317	268	268	4.96%
sao2	748	504	508	504	748	496	492	492	520	510	504	500	492	2.38%
Z5xp1	713	318	272	272	713	318	272	272	279	273	276	272	272	0.00%
Z9sym	828	165	153	143	828	165	153	138	150	137	131	140	131	8.39%
z4	375	128	112	112	375	126	112	112	112	111	112	112	111	0.89%

Table 4.4. Total CMOS gate inputs (#TI) for XAX EPOE default mode, greedy mode, and random mode syntheses of selected MCNC benchmark functions.

4.9.5 Experiment 5.

The fifth experiment repeated the fourth experiment with the XAX SPOE algorithm. Again the best performing mode was the greedy mode using an unrestricted

number of input variables in each y_i sum, although several of the particular functions for which the greedy mode performed best were different than in the fourth experiment. At the most extreme the default mode results when using an unrestricted number of input variables had 10.45% more gate inputs than the best-performing mode. As in the fourth experiment the relative performance of the different modes was similar considering FPGA costs. It was then decided to focus on CMOS total number of gate inputs, which parallels the SPP comparisons performed by others previously [18, 31].

	Default Mode				Greedy Mode				Random Mode					
Function	-i 1	-i 2	-i 3	without -i	-i 1	-i 2	-i 3	without -i	-r 0	-r 1	-r 2	-r 3	Best	% diff.
bw	478	478	506	506	478	476	489	489	631	594	535	526	476	5.93%
clip	1306	695	514	453	1306	660	514	458	488	507	482	429	429	5.30%
con1	36	32	35	35	36	32	35	35	71	35	35	67	32	8.57%
inc	237	263	249	249	237	272	269	269	337	291	287	253	237	4.82%
misex1	163	176	180	180	163	176	180	180	226	184	192	192	163	9.44%
rd53	191	100	80	72	191	100	80	72	72	72	72	72	72	0.00%
rd73	1011	431	274	215	1011	431	282	207	295	239	286	215	207	3.72%
rd84	2292	919	613	421	2292	915	608	398	505	577	437	445	398	5.46%
sao2	601	501	520	555	601	499	497	544	672	622	605	584	497	10.45%
Z5xp1	669	299	305	305	669	299	292	292	352	312	317	309	292	4.26%
Z9sym	861	216	216	232	861	216	216	232	268	236	244	220	216	6.90%
z4	513	172	106	106	513	166	106	106	106	106	106	106	106	0.00%

Table 4.5. Total CMOS gate inputs (#TI) for XAX SPOE default mode, greedy mode, and random mode syntheses of selected MCNC benchmark functions.

4.9.6 Experiment 6.

The sixth experiment compared the XAX SPOE algorithm with the SPPk algorithm of Ciriani [30] with respect to literals and synthesis time. While comparing literal counts in three-level logic could be considered academic, it does allow for the

XAX SPOE algorithm to be run with a single postprocessing stage. (The XAX command-line option "-p" selects use of a single postprocessing stage.)

The SPPk algorithm is a heuristic EXOR-AND-OR synthesis program for completely specified functions which uses the parameter k to control the number of prime pseudoproduct implicants considered in a cover for a function. In its most speculative mode, SPPk with $k = 0$ (or SPP0), computation time is exponentially faster than in an exact SPP synthesis; SPP0 syntheses produced a number of literals comparable to one half the number of literals in an exact SOP synthesis plus one half the number of literals in an exact SPP synthesis. Table 4.6 illustrates that XAX SPOE consistently outperformed SPP0 in literal counts; also, the synthesis times were typically significantly better for the XAX SPOE algorithm, assuming a speedup adjustment of 50 for integer performance of i7-2600K/Pentium III 450.

Function	Literals XAX -o -s -p	XAX Time (s)	Literals SPP0	SPP0 Time (s/50)	Literals SPP	SPP Time (s/50)
add6	189	0.83	1212	0.32	*	*
addm4	684	0.92	939	149.08	520	546.8
amd	610	0.313	905	1936.52	*	*
dist	541	0.26	639	0.46	422	1238.5
f51m	166	0.31	216	0.26	146	6.78
m4	783	0.75	785	1.28	646	362.46
max1024	882	0.53	1098	3.84	*	*
max512	676	0.4	693	0.8	517	252.18
mlp4	522	0.42	643	0.14	318	15.56
newcond	135	0.15	166	0.24	122	311.74
Geomean:	433.24	0.43	624.78	2.07	*	*

Table 4.6. Literals for XAX SPOE using a single postprocessing stage, SPPk where $k = 0$ (SPP0), and Exact SPP [30] synthesis of selected Espresso benchmark functions.

4.9.7 Experiment 7.

The seventh experiment compared the SPOE algorithm with exact minimum SOP and exact minimum SPP in terms of CMOS cost [31]. Compared to the exact minimum SPP method in the sixth experiment, this later algorithm had specialized methods for remapping autosymmetric functions to make synthesis more tractable; this approach could be applied to XAX algorithms (or in general any synthesis algorithm), albeit at a time penalty for functions which are not autosymmetric. The results are shown in Table 4.7. The XAX SPOE algorithm CMOS costs were typically closer to the exact minimum SPP costs than the exact minimum SOP costs.

Function	#TI SOP	#TI XAX -o -s	XAX Time (s)	#TI Exact SPP
5xp1	341	297	0.78	64
adr4	415	140	0.20	117
life	746	209	0.50	180
mlp4	853	817	0.30	524
rd53	171	72	0.70	64
rd73	883	215	0.70	207
rd84	2029	421	0.10	420
z4	311	106	0.60	100
Geomean:	551.68	216.93	0.40	158.23

Table 4.7. Total CMOS gate inputs (#TI) for exact-minimum SOP, XAX SPOE, and Exact SPP [31] synthesis of selected Espresso benchmark functions.

4.9.8 Experiment 8.

The eighth experiment compared the XAX SPOE algorithm in the greedy mode with the ORAX algorithm [18]. The ORAX algorithm remaps autosymmetric functions [17] and applies exact-minimum SOP synthesis on the result. For some autosymmetric functions this approach produced results which are not only lower in CMOS cost but also

require less time. Table 4.8 shows that the greedy mode XAX SPOE syntheses typically had lower CMOS costs and exponentially longer synthesis times than ORAX did.

Function	#TI XAX - o -g -s	XAX Time (s)	#TI ORAX	ORAX Time (s/50)
add6	327	0.57	1377	0.019
adr4	140	0.28	233	0.0028
dekoder	55	0.18	433	*
f51m	263	0.67	395	0.0044
m1	197	0.159	183	0.0034
max1024	1308	0.648	1146	0.015
newcwp	39	0.3	41	0.0016
radd	140	0.39	233	0.002
rd53	72	0.3	83	0.0012
rd73	207	0.42	495	0.0014
sqn	119	0.27	219	0.0014
wim	94	0.423	87	*
z4	106	0.77	157	0.0024
z9sym	232	0.55	368	0.0022
Geomean:	152.9	0.403	255.2	0.003

Table 4.8. Total CMOS gate inputs (#TI) for XAX SPOE and ORAX synthesis of selected Espresso benchmark functions.

4.10 Conclusion

In this section the theory of EXOR-AND-based synthesis was expanded and related to previous work [12]. Three spectral algorithms for EPOE and SPOE were presented. All three algorithms use a heuristic approach rather than relying on an input cube list or computing prime implicant expressions. Compared to heuristic SPP methods, the spectral methods are fast, synthesizing functions of up to 26 inputs at a rate of (at worst) approximately 6 1/2 minutes per output. In terms of classical digital technology, the results for EPOE synthesis were variable, sometimes better than ESPP synthesis and

other times worse than ESOP synthesis; the results for SPOE synthesis were more consistently superior to heuristic SPP synthesis, but provide no guarantee that performance will be superior to SOP synthesis. Consequently the presented algorithms are better suited to tournament or hybrid synthesis implementation than to stand-alone implementation.

5

Summary

New methods for linear reversible circuit synthesis and EXOR-AND-based circuit synthesis were presented herein. The new linear reversible circuit synthesis methods are the AECM method, the MCG method, and the GJCO method variants. Three new EXOR-AND-based circuit synthesis methods were implemented in the program XAX: an EPOE synthesis method for completely specified functions, an EPOE synthesis method for incompletely specified functions, and an SPOE synthesis method for incompletely specified functions [12].

The linear reversible circuit experiments demonstrated that the new methods consistently outperformed previous methods in terms of both CNOT gate counts and quantum gate counts. One unexpected result was that for small circuits the MCG method often produced the lowest quantum gate counts of all the methods, even though it was not designed to emphasize pseudo-Hadamard-inverse/pseudo-Hadamard gate pairs. Further, MCG can readily be adapted to produce permutation syntheses. Considering that building reliable reversible quantum computers of up to 10 qubits still poses technical challenges, it seems worthwhile to develop variants of MCG which further reduce gate counts. For large linear reversible circuits the GJCO variants could be extended numerous ways, such as using deeper searches to achieve better subcolumn eliminations or through evolutionary approaches.

To facilitate the development of EXOR-AND-based circuit synthesis methods, terms relating to POE expressions have been formally defined [12] and extended. Among

these new terms is the LCC operation and product transformation. Three heuristic EXOR-AND-based circuit synthesis algorithms were presented, as well as LCC-based postprocessing algorithms. These algorithms were implemented in the program XAX. In the XAX EPOE and SPOE experiments, product transformation typically achieved additional gate count reduction at a reasonable time penalty. In the XAX EPOE experiments, using an approximate cover criterion of $w = {}^{2}/_{3}$ was typically not the best; consequently a multi-pass approach is suggested for best results using a small number of different approximate cover criterion values. In the XAX SPOE experiments, the best results were achieved using both a greedy approach and performing a small number of passes using a different maximum number of literals per EXOR factor (as shown in Table 4.5, and comparing Table 4.8 with previous work [12]). Future work in this area would be to investigate the speedup gained by performing XAX synthesis in hardware and developing iterative extensions of the EXOR-AND-based circuit synthesis algorithms.

References

1. R. V. Meter, M. Oskin. "Architectural implications of quantum computing," ACM Journal on Emerging Technologies in Computing Systems. Volume 2 Issue 1, pp. 31-63, Jan. 2006.

2. R. Hughes et al. "A Quantum Information Science and Technology Roadmap," Available: http://qist.lanl.gov/, Apr. 2, 2004. Accessed Jan. 26, 2013.

3. D. Maslov. Reversible Logic Synthesis Benchmarks Page, Available: http://webhome.cs.uvic.ca/~dmaslov/definitions.html, 2011. Accessed Feb. 2, 2016.

4. B. Schaeffer, L. Tran, A. Gronquist, M. Perkowski, P. Kerntopf, "Synthesis of Reversible Circuits Based on Products of Exclusive Or Sums," 43th IEEE International Symposium on Multiple-Valued Logic, pp. 35–40, 2013.

5. G. Bard, Algebraic Cryptonalisis. Dordrecht, Springer, 2009.

6. K. N. Patel, I. L. Markov, J. P. Hayes, "Optimal Synthesis of Linear Reversible Circuits," Quantum Information & Computation, Vol. 8, no. 3, pp. 282-94, 2008.

7. R. Ishikawa, T. Hirayama, G. Koda, K. Shimizu, "New Three-level Boolean Expression Based on EXOR Gates," IEICE Trans. & Inf. Syst., E87-D, no. 5, pp. 1214-1222, 2004.

8. F. Luccio, L. Pagli, "On a New Boolean Function with Applications," IEEE Trans. on Computers, Vol. 48, March, pp. 296-310, 1999.

9. C. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design, OBDD – Foundations and Applications. Berlin, Heidelberg, New York, Springer-Verlag, pp. 235-249, 1998.

10. W. Günther, R. Drechsler, "Linear Transformations and Exact Minimization of BDDs," Proc. of the Great Lakes Symp. on VLSI, pp. 325-330, 1998.

11. M. Karpovsky, R. Stanković, and J. Astola, "Construction of Linearly Transformed Planar BDD by Walsh Coefficients," Proceedings of the 2004 International Symposium on Circuits and Systems, Vol.4, pp.IV-517-20, 2004.

12. B. Schaeffer, "Product Transformation and Heuristic EXOR-AND-OR Logic Synthesis of Incompletely Specified Functions," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, doi: 10.1109/TCAD.2017.2664064, available at http://ieeexplore.ieee.org/document/7842606/, accessed 3/6/2017.

13. L. Tran, B. Schaeffer, A. Gronquist, M. Perkowski, P. Kerntopf, "Synthesis of Reversible Circuits Based on EXORs of Products of EXORs," Trans. on Comp. Sci., XXIV, Special Issue on Reversible Computing, LNCS 8911, pp. 111-128, 2014.

14. R. Brayton, G. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, Logic Minimization Algorithms for VLSI Synthesis. Kluwer Academic Publishers, pp. 15-33, 1984.

15. T. Sasao, Switching Theory for Logic Synthesis. New York, Springer Science Business Media, pp. 1-55, 99-100, 289-295, 303; 1999.

16. M.Karpovsky, R. S. Stankovic, J. Astola, Spectral Logic and Its Applications for the Design of Digital Devices. Hoboken, Wiley-Interscience, pp. 2-27, 36-47, 169-171; 2008.

17. A. Bernasconi, V. Ciriani, F. Luccio, and L. Pagli, "Exploiting Regularities for Boolean Function Synthesis." Mathematical Systems Theory, vol.39, no.4, pp. 485-501, 2006.

18. A. Bernasconi, V. Ciriani, F. Luccio, and L. Pagli, "Synthesis of Autosymmetric Functions in a New Three-Level Form." Mathematical Systems Theory, vol.42, no.4, pp. 450-464, 2008.

19. A. De Vos., Reversible Computing, Fundamentals, Quantum Computing, and Applications. Weinheim, Wiley-VCH Verlag GmbH & Co., 2010.

20. B. Schaeffer, M. Perkowski, "Linear Reversible Circuit Synthesis in the Linear Nearest-Neighbor Model," 42nd IEEE International Symposium on Multiple-Valued Logic, pp. 157-160, May 14-16, 2012.

21. B. Schaeffer, "Computer Aided Design of Permutation, Linear, and Affine-Linear Reversible Circuits in the General and Linear Nearest-Neighbor Model", Master's Thesis, 2013.

22. B. Schaeffer, M. Perkowski, "A Cost Minimization Approach to Synthesis of Linear Reversible Circuits", unpublished manuscript, Available: http://arxiv.org/abs/1407.0070, 2014.

23. D. Maslov, "Linear Depth Stabilizer and Quantum Fourier Transformation Circuits with No Auxiliary Qubits in Finite-neighbor Quantum Architectures," Phys. Rev. A, vol. 76, pp. 052310-1-052316-7, 2007.

24. T. Sasao, "EXMIN2, a simplification algorithm for exclusive-OR-sum-of-products expressions for multiple-valued-input two-valued-output functions," IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., Vol. 12, no. 5, pp. 621-632, 1993.

25. A. Mishchenko, M. Perkowski, “Fast Heuristic Minimization of Exclusive Sum-of-Products,” International Workshop on Applications of the Reed-Muller Expansion in Circuit Design, pp. 242-250, 2001.

26. S. Selezneva, “On the Length of Boolean Functions in the Class of Exclusive-OR Sums of Pseudoproducts,” Moscow University Computational Mathematics and Cybernetics, Vol. 38, no. 2, pp. 64-68, 2014.

27. F. J. MacWilliams, N. J. A. Sloane, The Theory of Error-correcting Codes. Amsterdam, Elsevier/North-Holland, pp. 23-25, 1977.

28. K. McElvaine, IWLS’93 Benchmarks Set, Version 4.0, MCNC International Workshop on Logic Synthesis, 1993.

29. S. Yang, Synthesis and Optimization Benchmarks, User Guide, Microelectronic Center of North Carolina, 1991.

30. V. Ciriani, “Logic Minimization using Exclusive OR Gates,” Proceedings of the ACM/IEEE 38th DAC, pp. 115-120, 2001.

31. A. Bernasconi, V. Ciriani, F. Luccio, L. Pagli, “Three-Level Logic Minimization Based on Function Regularities,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., Vol. 22, no. 8, August, pp. 1005-1016, 2003.