Portland State University

PDXScholar

Dissertations and Theses | Dissertations and Theses

Spring 6-21-2013

Computer Aided Design of Permutation, Linear, and Affine-Linear Reversible Circuits in the General and Linear Nearest-Neighbor Models

Ben Schaeffer
Portland State University

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

Part of the Other Electrical and Computer Engineering Commons

Let us know how access to this document benefits you.

Recommended Citation

Schaeffer, Ben, "Computer Aided Design of Permutation, Linear, and Affine-Linear Reversible Circuits in the General and Linear Nearest-Neighbor Models" (2013). *Dissertations and Theses.* Paper 986.
https://doi.org/10.15760/etd.986

This Thesis 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. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Computer Aided Design of Permutation, Linear, and Affine-Linear Reversible Circuits in the General and Linear Nearest-Neighbor Models

by

Ben Schaeffer

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

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Marek Perkowski, Chair
Douglas Hall
Xiaoyu Song

Portland State University
2013

© 2013 Ben Schaeffer

Abstract

With the probable end of Moore's Law in the near future, and with advances in nanotechnology, new forms of computing are likely to become available. Reversible computing is one of these possible future technologies, and it employs reversible circuits. Reversible circuits in a classical form have the potential for lower power consumption than existing technology, and in a quantum form permit new types of encryption and computation.

One fundamental challenge in synthesizing the most general type of reversible circuit is that the storage space for fully specifying input-output descriptions becomes exponentially large as the number of inputs increases linearly. Certain restricted classes of reversible circuits, namely affine-linear, linear, and permutation circuits, have much more compact representations. The synthesis methods which operate on these restricted classes of reversible circuits are capable of synthesizing circuits with hundreds of inputs. In this thesis new types of synthesis methods are introduced for affine-linear, linear, and permutation circuits, as well as a synthesizable HDL design for a scalable, systolic processor for linear reversible circuit synthesis.

Dedication

For my mother.

Acknowledgments

I would like to express my thanks to several people for their support: my advisor Marek Perkowski for his encouragement to explore new ideas, committee members Douglas Hall and Xiaoyu Song for their time and attention, Addy Gronquist for his willingness to tackle a SystemVerilog linear reversible circuit synthesis project with me, Robin Marshall for his ideas on digital design, and everyone from the Portland Quantum Logic Group for their feedback.

Table of Contents

Abstract i
Dedication ii
Acknowledgments iii
Table of Contents iv
List of Figures v
List of Tables vii
1 Introduction 1
2 Reversible Circuits and Their Mathematical Representation 6
2.1 Overview 6
2.2 Classical Reversible Circuits 7
2.4 Classical Reversible Gates 13
2.5 Classes of Classical Reversible Circuits 15
3 Synthesis Methods for Permutation, Linear, and Affine-Linear Reversible Circuits 17
3.1 Discussion 17
3.2 Permutation Reversible Circuit Synthesis 18
3.3 Linear Reversible Circuit Synthesis 20
3.3.1 Discussion 20
3.3.2 Gaussian Elimination-based Linear Reversible Circuit Synthesis 22
3.3.3 Convergence of Gaussian Elimination-based Linear Reversible Circuit Synthesis 25
3.3.5 Linear Reversible Circuit Synthesis in the General Model 28
3.3.6 Linear Nearest-Neighbor Gaussian Elimination (LNNGE) 31
3.3.7 Linear Nearest-Neighbor Alternating Elimination (LNNAE) 35
3.3.8 Search Methods for LNNGE and LNNAE 41
3.3.9 Linear Reversible Circuit Synthesis Tests 43
3.3.10 Related Linear Reversible Circuit Synthesis Methods 46
3.4 Affine-Linear Reversible Circuit Synthesis 52
3.4.1 Optimal LNN Affine-Linear Reversible Circuit Synthesis Study of the 4×4 Input Reversal Circuit 53
4 LNNAE Hardware Design 57
4.1 Overview 57
4.2 Initial design 59
4.3 Hardware Implementation Discussion 60
4.4 Systolic Implementation 66
4.5 Results 71
Conclusion 73
References 75
Appendix 81
Appendix A: Source. 81
Appendix B: 16 Fundamental Types of Linear Reversible Circuit Synthesis 132
Appendix C: Systolic 2D Shift Register LNNAE Data Flow 133
Appendix D: Pseudo-method Test Results 153

List of Figures

2.1 Relationships between the classical reversible circuit classes 9

2.2 An example of Boolean invertible linear system of equations matrix (a) and a symbolic matrix representation (b) corresponding to a CNOT gate 10

2.3 An example matrix product representing a cascade of two CNOT gates which produces the identity matrix 10

2.4 Matrix representation of a SWAP gate composition of three CNOT gates. 11

2.5 Permutation matrix representation of a CNOT gate. 12

2.6 Schematic and truth table representations for the NOT gate, CONTROLLED-NOT (CNOT) gate, TOFFOLI gate, and SWAP gate.. 13

2.7 Schematic and truth table representation of the FREDKIN gate.. 14

3.1 An example permutation reversible circuit. 18

3.2 A comparison of different synthesis methods which perform an input reversal permutation 20

3.3 Employing Gaussian Elimination to synthesize linear reversible circuits 22

3.4 An invertible matrix after the first iteration of Gaussian Elimination 26

3.5 Example “Algorithm 1” search to find identical multibit numbers 29

3.6 A comparison of “Algorithm 1”[21] and “Modified Algorithm 1” 30

3.7 LNNGE synthesis of a "distance 2" CNOT gate 35

3.8 Matrix representation of LNNAE algorithm flow 36

3.9 LNNAE synthesis of the 3×3 input reversal function 38

3.10 LNNAE algorithm flow when synthesizing of the 3×3 input reversal function....... 39

3.11 Penalty matrix based on quadratic distance from matrix diagonal 47

4.1 Block diagram of proposed LNNAED recursive search system 58

4.2 Block diagram of the initial LNNAE coprocessor design which used dual row and column matrix access lines 59

4.3 Robin Marshall's systolic 2D shift register LNNAE design 62

4.4 Redesigned systolic 2D shift register LNNAE matrix.. 65

4.5 Redesigned systolic 2D shift register LNNAE system. 65

4.6 Organization of the LNNAE system testbench (designed by Addy Gronquist)......... 70

List of Tables

3.1 Comparisons of LNN linear reversible circuit synthesis methods (average adjacent CNOT gate counts) 44

3.2 Iterative deepening synthesis tests on 100 16×16 functions using "Best of Eight" search (average adjacent CNOT gate counts) 44

3.3 Frequency distribution of all optimally synthesized LNN linear reversible functions up to size 5x5 46

3.4 An example of how optimal linear reversible circuit synthesis of the input reversal function does not necessarily lead to an optimal affine-linear reversible circuit 56

4.1 Truth table for initial LNNAE coprocessor kernel block 60

4.2 Truth table for redesigned systolic LNNAE kernel 63

1

Introduction

Digital technology research explores new approaches to increase performance, reduce materials needed for manufacture, and reduce power consumption. At this time some people in the field are predicting the end of Moore's law within the next ten years [1]. With ongoing improvements in nanotechnology, new possibilities may become available. Reversible computing, either in a classical or quantum form, may become a future technology for niche digital applications. Other terms for reversible computing in the literature are reversible logic, adiabatic computing, and information-lossless computing.

Classical reversible computing is of interest because of its potential for low power consumption and implementation on a nanoscale. Unlike digital technology based on irreversible circuits, a reversible computer composed of reversible circuits can dissipate energy below the Landauer limit, which has recently been experimentally verified [2]. Introduced formally by Landauer [3] and later defended by Bennett [4], the Landauer limit is based on the principle that irreversible computing implies a physical irreversible process, and this process necessitates the dissipation of at least $kT\ln 2$ energy to erase a bit of information to avoid an increase in entropy. Permitting an increase in entropy in a system leads to increasing disorganization from a thermodynamics perspective. In engineering terms irreversible computing devices have a minimal requirement for heat dissipation to avoid electrical, material, and chemical changes which would result in system failure. Therefore ordinary irreversible digital devices will always produce some waste heat, even if all circuit elements are wired together with superconductive material.

After years of research classical reversible computing technology still has not become commercially viable. One recent experiment [5] demonstrated how classical reversible computing can work on a scale significantly larger than nanoscale using existing electronics technology, and, though successful, the experimental results clarify why industry has not adopted a classical reversible approach. The experiment demonstrated a charge-based RC circuit that reversibly stored a low and high value at a cost of a fraction of the Landauer limit while delivering 100 times the Landauer limit. Unfortunately this required a clock frequency under 900 Hz and did not demonstrate any kind of gates with multiple inputs and outputs. Complex Boolean functions typically require significantly more gates to be implemented in reversible circuits compared to traditional digital synthesis, so it is difficult to judge how well the experimenter's approach would fare when high gate counts would significantly reduce both performance and energy savings. From the time classical reversible computing was postulated digital technology has improved steadily, and currently there are many inexpensive low-power competing devices available. While these devices still are far above the Landauer limit in waste energy, they use a fraction of the energy per bit that high-performance computers use. Perhaps future advances in optical, molecular, or DNA [6] technology will lead to practical classical reversible computer implementations.

Quantum computing is of interest because it permits new forms of computation. While it could be said that the quantum phenomena employed occur on a small scale and operations are performed at extremely low power levels, the general consensus is that quantum computers require some kind of a hybrid system for measurement and performing algorithms. Reversible quantum computing circuits employ both classical

reversible and quantum gates, making it possible to implement algorithms that take advantage of quantum superposition and entanglement. Using quantum superposition, n quantum bits, or qubits, can be treated as if they represent 2^n binary words of width n. Once a superposition of qubits is established both quantum and quantum realizations of classical reversible gates can be applied to increase the probability that a particular word of interest will be measured. Quantum entanglement makes new forms of encryption possible which cannot be decrypted without the original encryption key.

Quantum computing is in its infancy and only a few devices, such as the Quantum Random Number Generator [7] and the D-Wave One [8] adiabatic quantum computer, are commercially available. At this time it still is unclear what architecture would be the best candidate for realizing the full potential of quantum computing. Multiple architectures have been proposed for this purpose including quantum dot, scalable ion-trap, nuclear magnetic resonance (NMR), Josephson junction, and linear optical quantum computing [9, 10].

In the general model of quantum computing, each qubit can directly interact with every other qubit. While this model would be ideal, some proposed quantum computing architectures have arrangements which restrict the gates they permit. Restricted qubit arrangements require quantum circuits to be synthesized in such a way that all gates employed are permissible for the target architecture. One restricted qubit arrangement model which has received attention [11-19] is the linear nearest-neighbor (LNN) model in which qubits reside in a one-dimensional array and interact only with their nearest neighbors. The LNN model is applicable to forms of quantum dot, NMR, and measurement-only optical quantum computers. A complication of using restricted qubit

arrangements is that in order to use the output from most reversible circuit synthesis programs a secondary synthesis step is required. For any significant number of wires the original synthesis will not be optimal and therefore performing secondary synthesis runs the risk of getting further away from an optimal synthesis. Currently the most advanced published work [20] on attempted optimal synthesis is limited to all four-wire LNN arbitrary reversible circuits.

Even in the event some quantum computing architecture becomes viable, there is no consensus on how best to approach synthesis of reversible and quantum circuits for systems with 100 to 1000 qubits. For reasonable synthesis times, functions need to be fully specified in RAM. To fully specify a 32×32 arbitrary reversible circuit requires 2^{32} 32-bit words which amounts to 64MB of RAM, a size which is found in servers today. Unfortunately increasing the number of qubits by one requires double the amount of RAM, so fully specifying a 64×64 arbitrary reversible circuit does not appear to be viable in the foreseeable future. Linear reversible circuits are an important subset of arbitrary reversible circuits that can be compactly represented and quickly synthesized [21] in the general model. Unfortunately, to convert these linear reversible circuits to the LNN model typically requires a significant increase in gate count.

The main focus of this thesis is the introduction of new scalable synthesis methods to directly synthesize linear reversible circuits in the LNN model. The motivation is to answer some basic questions about these reversible circuits in general and, more specifically, to create synthesis methods for future quantum computer technologies such as forms of quantum dot, NMR, and measurement-only optical quantum computers. My new methods are compared with exact optimal synthesis of all

five-wire LNN linear reversible circuits and with older linear reversible synthesis methods for circuits representing 8 to 64 qubits. The LNN development of these new methods has led to some insights on how general linear reversible circuits can be better synthesized, as well as considerations for affine-linear reversible circuit synthesis in both the general and LNN models.

The secondary focus of the thesis is a synthesizable HDL design for a scalable, systolic processor that performs LNN linear reversible circuit synthesis. Unlike the development of new synthesis methods referred to above which are my own design, the HDL work was done in conjunction with other PSU ECE students. The HDL design was a redesign of an architecture suggested by Robin Marshall in a discussion I had with him and Dr. Marek Perkowski. Later the HDL design was implemented with the help of Addy Gronquist, who created assertions, testbench routines, and demonstrated compilation using a Mentor Graphics Veloce emulator.

2

Reversible Circuits and Their Mathematical Representation

2.1 Overview

This section will introduce several key underlying concepts: reversible circuits, matrix representations of interest, classical reversible gates, and classes of reversible circuits. Later sections will introduce new methods for linear reversible circuit synthesis in the linear nearest-neighbor (LNN) model and compare them with optimal synthesis for circuits with five wires.

In the LNN model data elements are arranged in a one-dimensional array with interactions limited to adjacent data elements. The LNN model applies to forms of linear ion trap, quantum dot, NMR, and measurement-only optical quantum computers. Although there have been several articles [15-18] written about synthesis in the LNN model, the majority of synthesis methods use the general model. One advantage of approaching synthesis in the LNN model is that the results can be extended to more complex models, and the general model may be unrealistic for a scalable quantum computer [23].

There is essentially one work [21] which laid down the foundation for efficient linear reversible circuit synthesis in the general model, and this method will later be shown to map poorly to the LNN model. An outgrowth of my LNN specific synthesis methods was the development of approaches to improve on general model synthesis of linear reversible circuits.

2.2 Classical Reversible Circuits

In the broadest sense a reversible circuit is a physical device which is restricted to performing invertible operations. All classical reversible circuits can be represented by permutation matrices of dimension $2^N \times 2^N$ as illustrated in Figure 2.5. Because reversible circuits do not erase information, they can achieve energy loss values below the Landauer limit $kTln2$ [2, 3].

A classical reversible circuit is a device which implements a mapping of inputs to outputs that uses the same set and has a one-to-one and onto relationship. For convenience and to maintain common usage, the phrase "Classical Reversible Circuits" will imply a two-state device which is mapped to Boolean values unless otherwise stated; i.e. $\boldsymbol{B}=(0,1)$ and $f\colon \boldsymbol{B}^n \rightarrow \boldsymbol{B}^n$. A Boolean classical reversible circuit has the following property: applying output values as stimulus to the output permits recovery of input values at the input provided the underlying hardware is fully reversible.

A fundamental difference between reversible circuits and other types of circuits is that there is no fan-out. Although lacking fan-out may seem to be a significant limitation, with the additional wires irreversible functions can be mapped to reversible functions of higher dimensions. Two classes of additional or nonoutput wires can be used for this mapping: ancilla wires, meaning wires that are used temporarily and later restored to their original values, and garbage wires, meaning wires that become corrupted through use and once modified can no longer serve a purpose in later functions. The algorithms for linear reversible circuit synthesis do not use ancilla or garbage wires, and this is an advantage in quantum computing where getting something as small as a ten-qubit system to work is a major challenge.

Both classical reversible circuits and quantum reversible circuits can use the same schematic representation as illustrated in Figure 2.5. In this representation *n*-input variables, or qubits in a quantum computer context, on the far left side of the diagram can be considered as driving *n*-signals on a series of *n*-horizontal lines, called wires. Signal propagation is strictly horizontal, and after passing through a series of perpendicular gates which will be defined later, output values are measured on the far right side of the diagram. Groups of gates can be thought of as a subcircuit which, in the schematic, is denoted by a rectangular block which spans all locally involved wires. While this schematic representation may appear to imply a physical layout, in quantum computing the horizontal lines, gates, and subcircuits denote occurrences in time, not space.

2.3 Matrix Representations

Two types of Boolean square matrices are of interest in the representation of reversible circuits, permutation matrices and invertible linear system of equations matrices. Given an input vector X, the matrix cells of M represent coefficients which compose equations in the form $Y=MX$, and this fundamental equation serves as the mathematical representation for different classes of reversible circuits, illustrated in Figure 2.1. Permutation matrices have exactly one *1* matrix cell on every row and column and therefore $n-1$ matrix cells with *0*'s on every row and column. Invertible linear system of equations matrices have a much wider range of configurations, although no rows or columns will contain all *0*'s, nor can any row be identical to another row or any column be identical to another column.

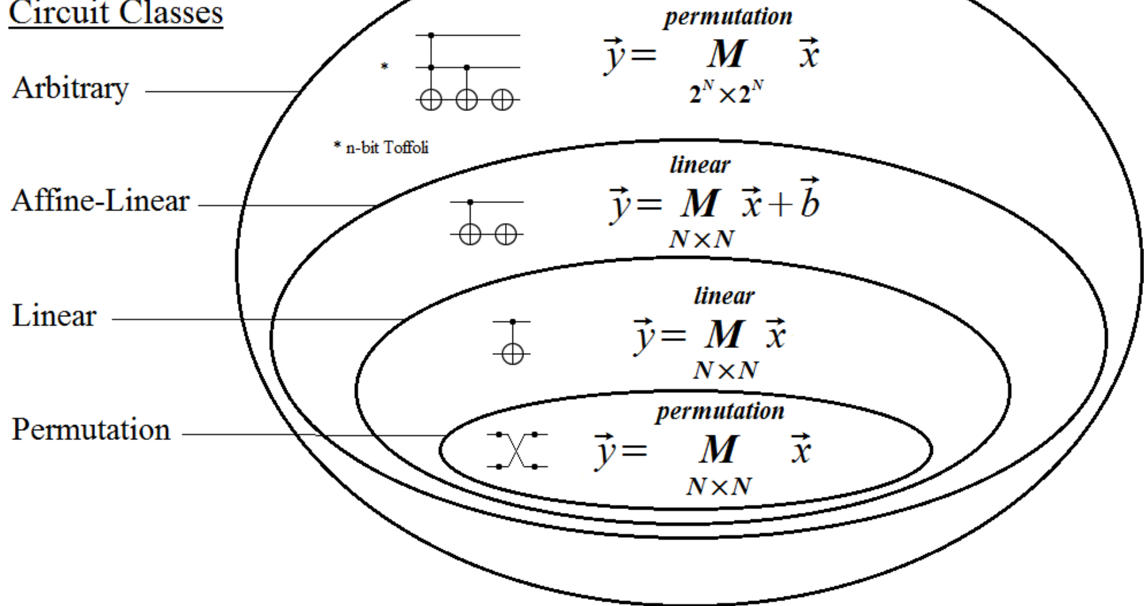


Figure 2.1: Relationships between the classical reversible circuit classes.

Invertible linear system of equations matrices of size $n \times n$ can be used to represent linear reversible circuits with n wires. Operations on this matrix representation form the basis of linear reversible circuit synthesis methods. In this context $Y=MX$ has a different treatment mathematically than real and complex matrices, as an XOR operation performs Boolean addition and an AND operation performs Boolean multiplication. As described in previous works [21, 22], the aforementioned mathematical treatment is equivalent to matrix operations in Galois field two, expressed as either *GF(2)* or F_2. A Galois field representation will not be emphasized here, as the invertible linear system of equations matrix representation can be extended to any composite number base larger than two and polynomial representation of finite field variables is not employed in any of the reversible circuit synthesis methods under consideration in this thesis.

Boolean Matrix Representation		
Wire	Input	Output
D_0	$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$	$\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$
D_1		

(a)

Symbolic Matrix Representation		
Wire	Input	Output
D_0	$\begin{bmatrix}a & \\ & b\end{bmatrix}$	$\begin{bmatrix}a & \\a & b\end{bmatrix}$
D_1		

(b)

Figure 2.2: An example of Boolean invertible linear system of equations matrix (a) and a symbolic matrix representation (b) corresponding to a CNOT gate.

Figure 2.2(a) illustrates an example Boolean invertible linear system of equations matrix where input vector $X=[a, b]^T$ and output vector $Y=MX=[a, a\oplus b]^T$. Figure 2.2(b) shows the same matrix in a symbolic representation. Introduced in [24], symbolic representation uses blank cells to represent matrix cells with *0*'s and sequential lowercase characters, with a unique character used one or more times in each column, to represent matrix cells with *1*'s. This results in a matrix which appears similar to the output vector Y in the previous figure and makes synthesis output more readable in larger matrices. Figure 2.3 shows that the same matrix multiplied by its inverse creates the identity matrix.

$$\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}=\begin{bmatrix}1\times 1\oplus 0\times 1 & 1\times 0\oplus 0\times 1\\1\times 1\oplus 1\times 1 & 1\times 0\oplus 1\times 1\end{bmatrix}=\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}\begin{bmatrix}1 & 0\\1 & 1\end{bmatrix}$$

Figure 2.3: An example matrix product representing a cascade of two CNOT gates which produces the identity matrix.

The three elementary row operations form a cornerstone of linear algebra theory: scaling a row, swapping two rows, and adding or subtracting two rows. For Boolean matrices, and subsequently for Boolean invertible linear system of equations matrices, there is no parallel to scaling a row, nor is there any difference between row addition and subtraction. Therefore for Boolean matrices the elementary row operations are swapping two rows and the modulo-two addition of one row to another. These elementary row operations can be performed via matrix multiplication using representative matrices as shown in Figure 2.3 above, the swap matrix appearing similar to the identity matrix but with two rows swapped, and the modulo-two addition matrix appearing similar to the identity diagonal with a single off-diagonal matrix cell with a *1*. One property of Boolean invertible linear system of equations matrices is that a SWAP gate matrix can be decomposed into a product of three modulo-two addition matrices as illustrated in Figure 2.4. This permits certain implementations of quantum computers to perform swap operations even though the underlying qubits are effectively fixed in place.

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Figure 2.4: Matrix representation of a SWAP gate composition of three CNOT gates.

Lemma 1. If a Boolean matrix is upper or lower triangular it represents an invertible Boolean linear system of equations. The proof is similar to the proof for invertible matrices of real values by backward elimination [25]. Proceeding one column at a time, all off-diagonal *1* matrix cells are converted to *0* by modulo-two addition with the

diagonal matrix cells. Once a column has no off-diagonal *1* matrix cells that column remains unchanged in subsequent modulo-two addition operations used to eliminate off-diagonal *1* matrix cells in other columns. After all columns are reduced to containing a single *1* matrix cell on the diagonal, the entire matrix will be equal to the identity matrix.

Quantum circuits of n wires can be represented mathematically by unitary matrices of dimension $2^n \times 2^n$. (Unitary matrices satisfy the equation $I = U^* U = U U^*$.) If synthesis is limited to classical reversible gates and functions, then a vector of 2^n values n-bit wide is a sufficient representation for arbitrary reversible circuits.

CNOT-down

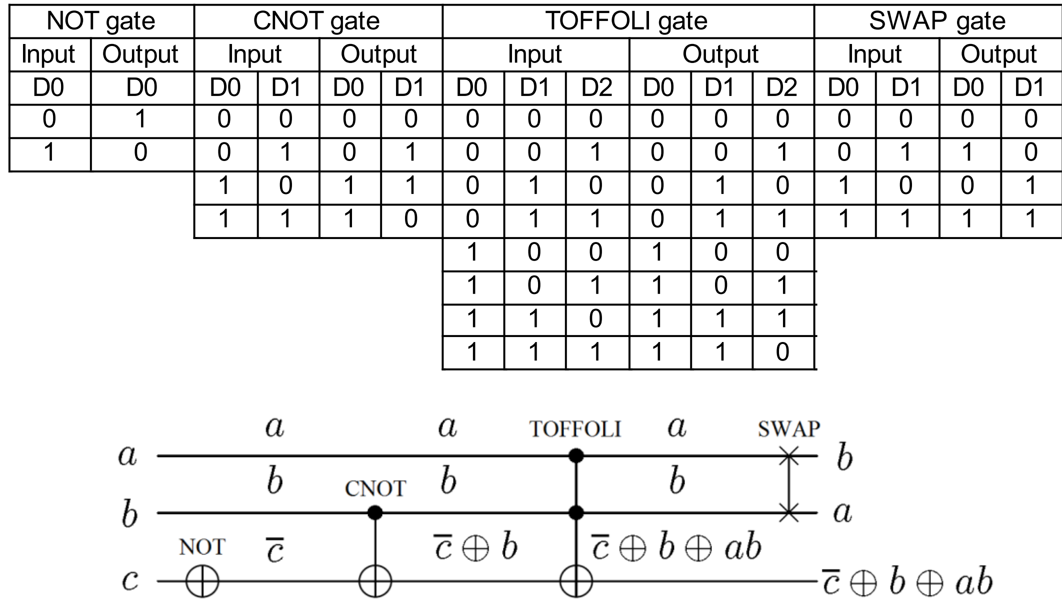
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{bmatrix}$$

CNOT-up

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \delta \\ \gamma \\ \beta \end{bmatrix}$$

Figure 2.5: Permutation matrix representation of a CNOT gate.

2.4 Classical Reversible Gates



NOT gate		CNOT gate				TOFFOLI gate						SWAP gate			
Input	Output	Input		Output		Input			Output			Input		Output	
D0	D0	D0	D1	D0	D1	D0	D1	D2	D0	D1	D2	D0	D1	D0	D1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	1	0	0	1	0	1	1	0
		1	0	1	1	0	1	0	0	1	0	1	0	0	1
		1	1	1	0	0	1	1	0	1	1	1	1	1	1
						1	0	0	1	0	0				
						1	0	1	1	0	1				
						1	1	0	1	1	1				
						1	1	1	1	1	0				

Figure 2.6: Schematic and truth table representations for the NOT gate, CONTROLLED-NOT (CNOT) gate, TOFFOLI gate, and SWAP gate.

Truth tables and schematic representations for all classical reversible gates are shown in Figure 2.6. An important property of these gates is that they are all self-inverse, and because of this property classical reversible circuit synthesis output can be applied in reverse order to produce an inverse circuit. In some texts another gate is used which performs a controlled-swap operation. Known as the FREDKIN gate in quantum computing, it can be composed of two CNOT gates and a TOFFOLI gate which is shown in Figure 2.7. Larger gates can be represented as a generalized TOFFOLI $k \times k$ gate with k-1 control lines and one target line.

FREDKIN gate					
Input			Output		
D0	D1	D2	D0	D1	D2
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	0	1
1	1	1	1	1	1

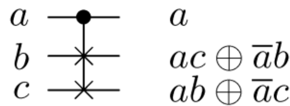


Figure 2.7: Schematic and truth table representation of the FREDKIN gate.

Although the synthesis methods in this work focus on matrix representation for an invertible linear system of equations, all of the classical reversible gates can be mapped to two-dimensional Hilbert space. Putting aside the SWAP gate and FREDKIN gate, the classical reversible gates' effect on input vectors in two-dimensional Hilbert space can be summarized as follows:

A NOT gate permutes all rows of an input vector.

A CNOT gate permutes *1/2* of the rows of an input vector.

A TOFFOLI gate permutes *1/4* of the rows of an input vector.

A *4×4* TOFFOLI gate permutes *1/8* of the rows of an input vector.

...

An $n \times n$ TOFFOLI gate permutes $1/2^{n-1}$ of the rows of an input vector, i.e. two rows.

In classical reversible circuits the TOFFOLI gate, also known as the CONTROLLED-CONTROLLED-NOT gate, implements the function $C' = C \oplus AB$. This gate is universal, so given unlimited TOFFOLI gates and ancilla wires to work with, any Boolean function can be composed. In quantum reversible circuits the situation is more complicated, as a set of gates is required to create a universal set. The CNOT gate combined with all 1×1 quantum gates is one possible universal set [27].

2.5 Classes of Classical Reversible Circuits

1. The Identity Circuit

Formula: $Y=X$ where Y has n elements.

Number of circuits: 1.

This is the identity function represented by n parallel wires.

2. Inverter Networks

Formula: $Y=X\oplus B$ where Y has n elements.

Number of circuits: 2^n.

This reversible circuit is composed solely of NOT gates. The vector entries with 1's in vector B correspond to NOT gates, making these circuits trivial to synthesize.

3. Permutation Circuits

$Y=MX$ where Y has n elements and the matrix M is an $n\times n$ permutation matrix.

Number of circuits: $n!$.

This type of reversible circuit is composed solely of SWAP gates. When SWAP gates are not available, as is the case in quantum architectures like NMR, swapping is achieved through linear reversible circuits.

4. Linear Reversible Circuits

Formula: $Y=MX$ where Y has n elements and the matrix M is an $n \times n$ invertible matrix.

Number of circuits: approximately $0.29 \cdot 2^{n^2}$ [35].

This type of reversible circuit is composed of CNOT gates and, hardware permitting, SWAP gates. It is a superset of permutation reversible circuits.

5. Affine-Linear Reversible Circuits

Formula: $Y=MX \oplus B$ where Y has n elements and the matrix M is an $n \times n$ invertible matrix.

Number of circuits: approximately $0.29 \cdot 2^{n(n+1)}$.

This type of reversible circuit is composed of NOT gates, CNOT gates, and, hardware permitting, SWAP gates. It is a superset of linear reversible circuits.

6. Arbitrary Reversible Circuits

Formula: $Y=MX$ where Y has 2^n elements and the matrix M is a $2^n \times 2^n$ permutation matrix.

Number of circuits: $2^n!$.

This is the most complex set of reversible circuits and is a superset of all previously introduced reversible circuits; it is an active area of research [15-18, 20, 28, 31, 32].

3

Synthesis Methods for Permutation, Linear, and Affine-Linear Reversible Circuits

3.1 Discussion

Prior works have introduced methods for permutation reversible circuits using sorting algorithms [28] and linear reversible circuit synthesis in the general model using Kronrod's method for inverting matrices [21, 28]. This section will introduce new methods of synthesis of permutation and linear reversible circuits in the LNN model. In both the general model and the LNN model, synthesis of affine-linear reversible circuits can be treated as a linear reversible circuit synthesis followed by an inverter network synthesis. Affine-linear reversible circuit synthesis may be regarded as a less compelling problem than linear reversible circuit synthesis for two reasons. The first is that the linear reversible circuit component at worst results in approximately $\frac{1}{2}n^2$ CNOT gates [21] in the general model and $2n^2-3n+1$ CNOT gates (which will be derived later) in the LNN model, yet the inverter network requires at worst n NOT gates in both models. The second is that in technologies like NMR and ion trap, the latency of a CNOT gate operation is approximately three times that of a NOT gate. Consequently methods to improve linear reversible circuit synthesis will receive the greatest focus.

All of these methods can be used to specify and synthesize reversible circuits with hundreds of variables, and their worst-case gate count for large numbers of wires is significantly lower than arbitrary reversible circuit synthesis methods. One inherent disadvantage these methods have is the limited number of possible circuits they can describe as compared with arbitrary reversible circuits. Nonetheless, because

decoherence is such a fundamental roadblock for quantum computing, it seems appropriate to explore all possible ways to improve the class of linear reversible circuits.

3.2 Permutation Reversible Circuit Synthesis

If the underlying hardware directly supports a SWAP operation, permutation reversible circuit synthesis can be optimally solved by adapting the selection sort algorithm for synthesis in the general model and the insertion sort or bubble sort for synthesis in the LNN model [28]. An informal argument that these sort algorithms produce optimal SWAP gate counts would be that the algorithms essentially iterate through a series of cycles, and each cycle of *k* wires is optimally synthesized with *k-1* SWAP gates. For instance, the following cycle of three wires can optimally be synthesized with two SWAP gates: *(abc)*, which is a mapping of *a* to *b*, *b* to *c*, and *c* to *a*.

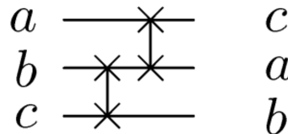


Figure 3.1: An example permutation reversible circuit.

If a SWAP gate is not available in hardware, permutation reversible circuit synthesis relies on CNOT gates. If the target hardware is the general model, then a SWAP gate list generated from selection sort synthesis can be converted to a linear reversible circuit where each SWAP gate corresponds to a triplet of CNOT gates in the form *CNOT(1, 2), CNOT(2, 1), CNOT(1, 2)*. Consequently a cycle of *k* wires can be synthesized with *3k-3* CNOT gates. It warrants mentioning that with the addition of one

ancilla line preset to *0* it would be possible to synthesize cycles of *k* wires optimally with $2k+2$ CNOT gates, though synthesis with ancilla lines is beyond the scope of this work.

Continuing under the condition that a SWAP gate is not available in hardware, if the target hardware is in the LNN model, then converting output from the insertion sort-based or bubble sort-based algorithms may or may not generate optimal synthesis. The situation may also hold if the target hardware supports SWAP gates at a latency cost near to but under three CNOT gates. Figure 3.2 shows a comparison of three different realizations of the 4×4 input reversal function. The circuit in Figure 3.2(1) is based on the selection sort algorithm and has 2 SWAP or 6 CNOT gates which is optimal, but after LNN conversion has 27 adjacent CNOT gates. The circuit in Figure 3.2(2) is based on the insertion sort and has $\frac{1}{2}(n^2-n)=6$ adjacent SWAP gates or 18 adjacent CNOT gates. The circuit in Figure 3.2(3) is an optimal linear reversible circuit synthesis which has $n^2-1=15$ adjacent CNOT gates. While it is difficult to compute optimal linear reversible circuits for slightly larger numbers of wires, the pattern in Figure 3.2(3) can be generated algorithmically for large circuits using an LNN version of Gaussian Elimination.

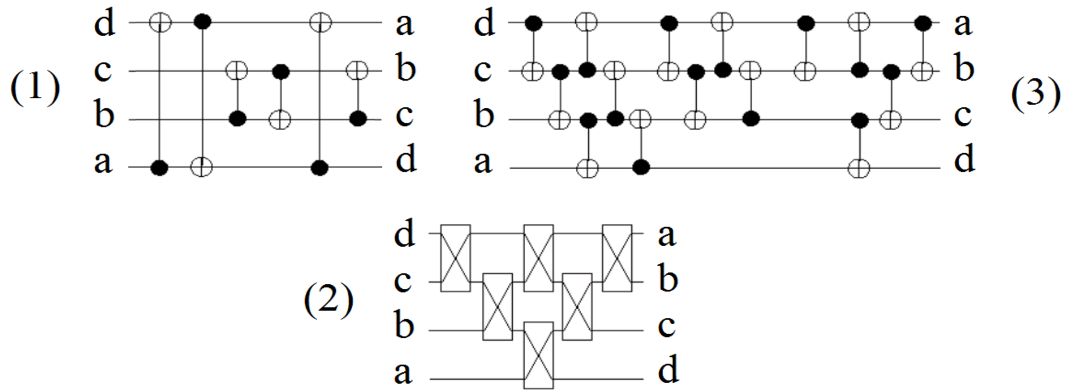


Figure 3.2: A comparison of different synthesis methods which perform an input reversal permutation.

3.3 Linear Reversible Circuit Synthesis

3.3.1 Discussion

Considering real and complex linear systems of equations in the form *Y=MX*, there are two fundamental types of computations that are of utility for a broad number of applications: the computation of vector *Y* when matrix *M* and vector *X* are known, and the computation of vector *X* when matrix *M* and vector *Y* are known. The former task is simpler, performed using matrix multiplication, while the latter is more complicated because it requires a matrix inverse computation. Matrix inverse computation algorithms and hardware constitute their own area of study.

A third fundamental type of computation which has similarities to the matrix inversion computation arises in linear reversible circuit synthesis: the computation of a minimal (or near minimal) elementary row operation decomposition for an invertible matrix. Because each elementary row operation corresponds to a CNOT gate in a linear reversible circuit, this type of computation is useful in minimizing linear reversible

circuit cost, which in quantum computing is equivalent to minimizing CNOT gate count. (For simplicity elementary swap operations will be treated as unavailable primitive operations in hardware). Since the goal of linear reversible circuit synthesis is a CNOT gate list, this third fundamental type of problem is not made easier by knowing a matrix's inverse. As a consequence of this as well as other difficulties, methods for inverting large matrices like Strassen's algorithm do not seem to be adaptable to linear reversible circuit synthesis. Gaussian Elimination-based algorithms like Gauss-Jordan, Kronrod's method for matrix inversion, and the LNN algorithms introduced here do generate elementary row operations, making these algorithms adaptable to linear reversible circuit synthesis.

One benefit of adapting Gaussian Elimination-based algorithms for synthesis is that their maximum number of elementary row operations can be calculated, thus setting an upper bound on CNOT gate counts, while one undesirable consequence of adapting these algorithms for synthesis is that their results will typically not be optimal. One peculiar issue with Gaussian Elimination [29] is that although this algorithm is widely accepted as convergent, it has not yet been formally proven to be so. For linear reversible circuit synthesis, Gaussian Elimination-based methods can be viewed in the narrower context of operations on Boolean linear system of equations, making it simpler to examine convergence and the reasons why the algorithm works. As part of this examination, the adaptation of Gaussian Elimination for linear reversible circuit synthesis will be reviewed, and then a postulate will be proposed which summarizes how all types Gaussian Elimination-based algorithms treat convergence and perform linear reversible circuit synthesis. Finally an analysis will follow to illustrate why Gaussian Elimination appears to be convergent.

3.3.2 Gaussian Elimination-based Linear Reversible Circuit Synthesis

$$\begin{bmatrix} & b & \\ a & & \\ & & c \end{bmatrix} R_1 \rightarrow R_1 \oplus R_2 \begin{bmatrix} a & b & \\ a & & \\ & & c \end{bmatrix} R_2 \rightarrow R_1 \oplus R_2 \begin{bmatrix} a & b & \\ & b & \\ & & c \end{bmatrix} R_1 \rightarrow R_1 \oplus R_2 \begin{bmatrix} a & & \\ & b & \\ & & c \end{bmatrix} \quad \text{(a)}$$

$$M_{circuit}^{-1} M_{circuit} = I \quad \text{(b)}$$

$$M_{circuit}^{-1} = M_{CNOT3} M_{CNOT2} M_{CNOT1} \quad \text{(c)}$$

$$(\ldots M_{CNOT3} M_{CNOT2} M_{CNOT1}) M_{circuit} = I \quad \text{(d)}$$

$$M_{circuit} = M_{CNOT1} M_{CNOT2} M_{CNOT3} \cdots \quad \text{(e)}$$

$$\left(\begin{bmatrix} a & b & \\ & b & \\ & & c \end{bmatrix} \begin{bmatrix} a & & \\ a & b & \\ & & c \end{bmatrix} \begin{bmatrix} a & b & \\ & b & \\ & & c \end{bmatrix} \right) \begin{bmatrix} & b & \\ a & & \\ & & c \end{bmatrix} = \begin{bmatrix} a & & \\ & b & \\ & & c \end{bmatrix} \quad \text{(f)}$$

Input Output (g)

Figure 3.3: Employing Gaussian Elimination to synthesize linear reversible circuits.

Figure 3.3, which will be described in detail later in this section, illustrates how Gaussian Elimination-based synthesis decomposes a function from output to input. Specifically Figure 3.3 illustrates algorithmic synthesis of the SWAP gate realized with CNOT gates. Paraphrasing [24, 30],

> ...each row of this $n \times n$ matrix corresponds to a wire, and the value on this wire is calculated as the XOR sum of input variables. In such a representation the identity matrix corresponds to the original inputs on the wires.
>
> A modulo-2 addition form of Gaussian Elimination serves as the foundation of both "Algorithm 1" and the methods introduced in this work. Given some invertible function in matrix A, Gaussian Elimination uses elementary row operations, each corresponding to a CNOT gate, to compute A^{-1}. This inverse

matrix is used to solve for the identity matrix, i.e. $I=A^{-1}A$. As long as elementary row operations are restricted to modulo-2 addition, Gaussian Elimination can be applied to representations of linear reversible circuits. Gaussian Elimination can be viewed as an elementary row operation sequence generator. Each elementary row operation sequence directly maps to a CNOT gate sequence. Applying a Gaussian Elimination-generated CNOT gate sequence to a reversible circuit in reverse order realizes the linear reversible function corresponding to the function matrix. Alternately applying the same CNOT gate sequence to a reversible circuit in forward order realizes the linear reversible function corresponding to the inverse of the original problem matrix.

Performing Gaussian Elimination on a transposed version of a problem matrix usually results in a different number of row operations, which corresponds to a different number of CNOT gates. Transposed matrix output requires swapping control and target values for each CNOT gate before it can be used [21] which is a consequence of the linear algebra property $A^TB^T=(BA)^T$. After control and target swapping, the resulting CNOT gate sequence is in the correct order to perform the corresponding linear reversible function.

Once an elementary row operation sequence for a linear reversible function has been calculated, it can be used on the identity matrix to compose the inverse of a problem matrix. Problem matrices usually are not equivalent to their inverses, even in a sizable portion of low complexity linear reversible functions. In the typical case where a matrix does not equal its inverse, Gaussian Elimination can be performed on both the inverse matrix and its transposed version. This

generates additional synthesis alternatives which typically result in a different number of CNOT gates.

Figure 3.3 uses symbolic Boolean matrix representation which assigns a unique character to each column and replaces *0*'s with blanks and *1*'s with characters. This Boolean matrix representation was originally developed to increase readability of synthesis debug output and was not used mathematically. In Figure 3.3(a) the first phase of Gaussian Elimination begins with operations on the far left column. A forward substitution is required to establish a value of *1* on the diagonal, so *R2* is used to modify row *R1*. Next a backward substitution is required to make the column upper triangular, so *R1* is used to change row *R2*. The middle column is already upper triangular, so no additional changes are required, and therefore the first phase of Gaussian Elimination is complete. The second phase of Gaussian Elimination begins with operations on the far right column, which already is equal to its corresponding column in the identity matrix. Finally a backward substitution on the middle column is required, changing row *R1* and resulting in the matrix becoming equal to the identity matrix. The entire process illustrated in Figure 3.3(a) can be summarized as a matrix inverse computation expressed in Figure 3.3(b), or, in more detail, as a matrix inverse decomposition into the product of three elementary row matrices in Figure 3.3(c), $M_{CNOT3}M_{CNOT2}M_{CNOT1}$. In general the matrix inverse solution Gaussian Elimination produces is a product of a variable number of elementary row operation matrices in Figure 3.3(d). Using the property that the Boolean elementary row operations being used here are self-inverse, Figure 3.3(e) demonstrates that the product of these matrices in reverse order is a decomposition of the matrix being synthesized. Therefore the linear reversible circuit in Figure 3.3(g) is

composed of CNOT gates corresponding to matrices M_{CNOT3}, M_{CNOT2}, and finally M_{CNOT1} in Figure 3.3(f) .

3.3.3 Convergence of Gaussian Elimination-based Linear Reversible Circuit Synthesis

Postulate 1: If, using a finite number of elementary row operations, an $n \times n$ linear system of equations matrix can be made upper or lower triangular, then the original matrix is invertible, Gaussian Elimination will converge, and the corresponding linear reversible circuit will be synthesizable. Conversely if, using a finite number of elementary row operations, an $n \times n$ linear system of equations matrix cannot be made upper or lower triangular, then the matrix is singular, Gaussian Elimination will not converge, and the matrix does not correspond to a linear reversible circuit.

An informal analysis of convergence will be discussed next. For simplicity the common approach for Gaussian Elimination will be examined in which, during the first phase of the algorithm, an upper triangular matrix is established by operating on columns from left to right. In the first iteration of the first phase, the left-most, or first, column is operated on to put it into an upper triangular matrix pattern. If the left-most column consists entirely of zeros then the matrix cannot have an inverse and therefore is singular. If the left-most column has one or more ones then through elementary row operations it can become upper triangular, in which case it is unknown if the matrix is singular or invertible. This argument for separate treatment of zero-filled columns from other columns is as follows:

From the point of view of a single column in an $n \times n$ Boolean matrix, its matrix cells can be only one of two unique values, *0* and *1*. Assuming a column is not zero-filled and treating all other matrix cells with *1*'s as copies of one particular matrix cell whose value is *1*, it can be argued that through forward substitution and backward elimination any finite-dimensioned column can be reduced to a single *1* in the top matrix cell with the remainder matrix cells *0*. This pattern is the only possible pattern the first column can have if it is upper triangular as is illustrated in Figure 3.4. Additionally, after the first column has been modified to contain a single *1* matrix cell, there are no further eliminations possible; i.e. by exhaustively searching all 2×2 elementary row operations on the identity matrix it was verified that no sequence of elementary row operations on the 2×2 identity matrix result in a zero-filled column.

$$\begin{bmatrix} 1 & a_{12} & a_{13} & \cdots & a_{1n} \\ 0 & a_{22} & a_{23} & \cdots & a_{2n} \\ 0 & a_{32} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix}$$

Figure 3.4: An invertible matrix after the first iteration of Gaussian Elimination.

Considering the case where the first column is not zero-filled, after the first iteration of Gaussian Elimination any matrix will be in the pattern in Figure 3.4. Starting from this pattern, the top matrix row no longer affects the determination of whether or not the original matrix is singular or invertible. This follows because during the second phase

of Gaussian Elimination off-diagonal matrix cells in the top row require only backward elimination to achieve the identity matrix. Backward elimination on the top row depends on the submatrix $M_{(2,2)-(n,n)}$ being amenable to becoming upper triangular. If the subcolumn $M_{(2,2)-(2,n)}$ is zero-filled, then the submatrix $M_{(2,2)-(n,n)}$ is singular and cannot become upper triangular. Also if the subcolumn $M_{(2,2)-(2,n)}$ is zero-filled, then through exhaustive search it can be shown that the submatrix $M_{(1,1)-(2,2)}$ is singular and no sequence of elementary row operations can simultaneously make the first and second column equal to their identity matrix values simultaneously. This would seem to be a reasonable argument that M is singular.

If the subcolumn $M_{(2,2)-(2,n)}$ is not zero-filled, then, using the same treatment as was used on the first column, the second column can be made upper triangular. Starting from this pattern, the top two matrix rows no longer affect the determination of whether or not the original matrix is singular or invertible. The determination now rests on whether or not the submatrix $M_{(3,3)-(n,n)}$ is amenable to becoming upper triangular. This process continues until either a subcolumn is discovered to be zero-filled, meaning that the original matrix was singular, or, after $n-1$ iterations, the matrix cell $M_{(n,n)}$ is verified to be a 1, meaning that the matrix has become upper triangular and is therefore invertible. Thus Gaussian Elimination can be viewed as both a convergent algorithm for invertible matrices and a validity checking mechanism for singular matrices.

3.3.4 Gaussian Elimination-based Matrix Validity and Synthesis Verification

Employing Gaussian Elimination for input validity checking is fast and applies to to all hardware models. A related checking mechanism is output verification, i.e.

verifying that the CNOT gate sequence resulting from synthesis composes the desired circuit. This is performed by representing each CNOT gate as an elementary row operation, performing the entire sequence of elementary row operations on the identity matrix, and testing equality with the matrix that was synthesized originally.

A consequence of the limited arrangements of invertible matrices which represent Boolean linear system of equations is that Gaussian Elimination-based methods require only $n-1$ iterations per phase. In effect the last column is solved indirectly because there is no invertible matrix that permits the last column's diagonal cell to be zero, and this is verifiable through exhaustive searching of all 2×2 matrices or using the 5×5 optimal LNN linear reversible circuit synthesis database which will be introduced later.

3.3.5 Linear Reversible Circuit Synthesis in the General Model

In the reversible circuit synthesis literature only two linear specific methods, both for the general model, have been discussed, Gaussian Elimination and Kronrod's method. "Algorithm 1" [21] is based on Kronrod's method [33] for inverting matrices which is more commonly known as the "Method of the Four-Russians" inversion (M4RI) [34], though this name is somewhat inaccurate as one of the four authors was not Russian. A key idea introduced in [33] was that matrix operations on Boolean matrices are fundamentally more limited than real matrices. For instance, by treating sub-rows, i.e. two or more adjacent bits in a row, as a single number of interest, a significant portion of the elementary row operations that Gaussian Elimination would normally perform separately on two or more columns can be efficiently combined. Considering the simplest version of "Algorithm 1" where two bits taken from adjacent columns in a single row are

treated as a single multibit number, it follows that the two bits can at most express four unique two-bit numbers; therefore whenever Boolean matrices are of dimension 5×5 or greater as is the case in Figure 3.5, any pair of columns will contain at least one repeated two-bit value. In each iteration of "Algorithm 1" the non-zero repeated numbers belonging to a set of two or more columns are found and are next eliminated by searching from the highest row to the second lowest row. Once all multibit repeated values are eliminated, the first phase of Gaussian Elimination is performed on columns in the set. After all columns are processed the remainder matrix will be upper triangular and there will be a corresponding CNOT gate sequence; the matrix is then transposed and the whole process repeated, which results in the identity matrix and a transposed CNOT gate sequence.

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 3 & 0 & 0 \\ 2 & 1 & 2 \\ 1 & 0 & 2 \\ 3 & 3 & 3 \\ 3 & 1 & 3 \\ 0 & 3 & 2 \end{bmatrix}$$

Figure 3.5: Example “Algorithm 1” search to find identical multibit numbers.

The larger the Boolean matrix, the greater the number of repeated numbers in each set of columns and the greater the opportunity for backward elimination to do approximately two or more Gaussian Elimination column processing iterations in one iteration. Also, larger Boolean matrix dimensions permit using wider multibit words which are chosen as a fraction of log_2n, thus improving efficiency of repeated number elimination. "Algorithm 1" in [21] employed this approach not for speed, which was the

original motivation in [33], but to reduce the total number of elementary row operations and hence the number of CNOT gates.

Improving upon "Algorithm 1" was difficult for this writer, but by using a two-pass approach approximately two percent lower gate count was achieved. The first pass employs the original "Algorithm 1" which searches to eliminate repeated multibit values starting from the top row and proceeding down, and the second pass uses my "Modified Algorithm 1" which searches to eliminate repeated multibit values starting from the second lowest row and proceeding upwards. Figure 3.6 compares "Algorithm 1" output on the left and "Modified Algorithm 1" on the right, demonstrating that these two methods can produce different synthesis results even in a *4×4* linear reversible circuit.

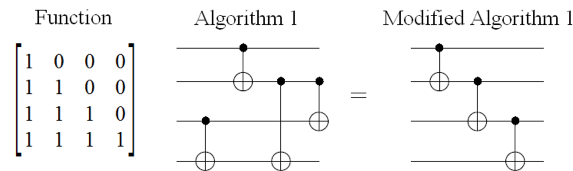


Figure 3.6: A comparison of "Algorithm 1"[21] and "Modified Algorithm 1".

Another small improvement came from recognizing a minor issue in "Algorithm 1" [21] in which an upper triangular matrix was transposed before synthesizing. I discovered that while this matrix transmission did simplify code it sometimes increased CNOT gate counts. Even in *8×8* matrices synthesizing both upper triangular matrix functions and their transposed forms produced realizations which frequently differed by a small number of CNOT gates. My discovery of the above property combined with the prior two-pass approach yields a four-pass approach.

3.3.6 Linear Nearest-Neighbor Gaussian Elimination (LNNGE)

In this writer's opinion the work in [21] was insightful, recognizing the potential for adapting Kronrod's method to linear reversible circuit synthesis and reducing the worst-case gate count to be $O(n^2/log_2n)$. Unfortunately applying Kronrod's method to linear reversible circuit synthesis in the LNN model produces poor results. As was stated in [24],

> the fundamental drawback converting synthesis output to the LNN model is that the output of both "Algorithm 1" and Gaussian Elimination includes distant CNOT gates where the control and target are not adjacent. Distant gates result in a cost increase, as prior work has shown; a CNOT gate with distance d of two or greater between its target and control requires $4d-4$ adjacent CNOT gates [31]. Test results will show how poorly this approach compares to direct LNN synthesis.
>
> A new method called LNNGE provides a fast and efficient synthesis alternative for LNN hardware. LNNGE follows the same form as Gaussian Elimination but restricts row operations to adjacent rows. Because of this restriction forward substitution may need to be performed multiple times on the same column to establish a 1 on its associated diagonal.
>
> Gaussian Elimination operations can start either with the first column to achieve an upper triangular matrix or less commonly with the last column to achieve a lower triangle matrix. In the second phase of Gaussian Elimination the triangle matrix is reduced to the identity matrix.

To perform an LNNGE using the upper triangular matrix approach, *n-1* columns are initially processed from left to right. In this first phase of the algorithm, each column is searched for the lowest row containing a *1*. All *1*'s located in matrix cells not on the identity diagonal represent terms in an XOR sum which require elimination. At most one CNOT-up gate, which corresponds to forward substitution, and one CNOT-down gate, which corresponds to backward elimination, are required to move the lowest *1* in a column up one row. Gates are applied to repeatedly raise the lowest *1* in the column being processed until it is on the identity diagonal. After repeating this procedure for *n-1* columns, the last column will have a *1* on its diagonal row and an upper right triangle matrix will be established, thus ending the first phase of LNNGE. In the second phase, columns are processed from right to left. Each column is searched to find the highest row containing a *1*. Then all *0*'s between the diagonal row and the highest row containing a *1* are operated on with CNOT-up gates. The highest *1* in the column is repeatedly lowered using CNOT-up gates until it is on the identity diagonal. By using only CNOT-up gates in the second phase of the algorithm, the upper right triangle matrix is preserved. The pseudocode for LNNGE follows:

```
FOR col FROM 0 TO N-2
  FOR row FROM N-1 TO col+1 by -1
    IF M[row][col] THEN
      IF NOT M[row-1][col] THEN
        CNOT(M[row]->M[row-1])
      CNOT(M[row-1]->M[row])
```

```
FOR col FROM N-1 TO 1 by -1
  row = 0
  WHILE row<col AND NOT M[row][col]
    row = row+1
  IF row NOT EQUAL col THEN
    FOR rowh FROM col TO row-1 by -1
      IF NOT M[rowh-1][col] THEN
        CNOT(M[rowh]->M[rowh-1])
      row = row+1
      WHILE row <= col
        CNOT(M[row]->M[row-1])
        row = row+1
```

LNNGE's maximum gate count can be calculated through simple analysis. In the first phase the maximum number of gates is $2(n-1)+2(n-2)+\ldots = n^2-n$. In the second phase the identity diagonal is already established, making the worst-case gate count for the longest column $2(n-1)-1$. Summing up the subsequent rows leads to a maximum number of gates of $(2(n-1)-1)+(2(n-2)-1)+\ldots=n^2-n-(n-1)=n^2-2n+1$. Thus the maximum number of gates for LNNGE is $(n^2-n)+(n^2-2n+1)=2n^2-3n+1$. In comparison "Algorithm 1" has at worst an upper limit of $\frac{1}{2}n^2+14n$ distant gates [21].

In Figure 3.7 LNNGE synthesis is illustrated step by step. In each step an input matrix on the left has an elementary row operation performed, denoted by a corresponding CNOT gate, which results in the output matrix on the right. Following the LNNGE algorithm, the first column from the left is examined and found to not be upper triangular. In step (1) a forward substitution from row 3 to row 2 is performed on the left matrix which results in the right matrix. In the right matrix modified bits in row 2 are displayed with a dark background. In step (2) a backward elimination from row 2 to row 3 is performed, modifying all bits in row 3. In step (3) a backward elimination from row 1 to row 2 is performed, modifying one bit in row 2 and resulting in the first column becoming upper triangular. Next the second column is examined and found to not be upper triangular. In step (4) a backward substitution from row 2 to row 3 is performed, modifying two bits in row 3 and resulting in the second column becoming upper triangular and ending the first phase of LNNGE. In the second phase of LNNGE the third column is examined and found not to be identical to its corresponding identity matrix column. In step (5) a backward substitution from row 3 to row 2 is performed, modifying one bit in row 2 and resulting in the last column becoming identical to its corresponding identity matrix column. The second column is examined and found to be identical to its corresponding identity matrix column ending the second phase of LNNGE.

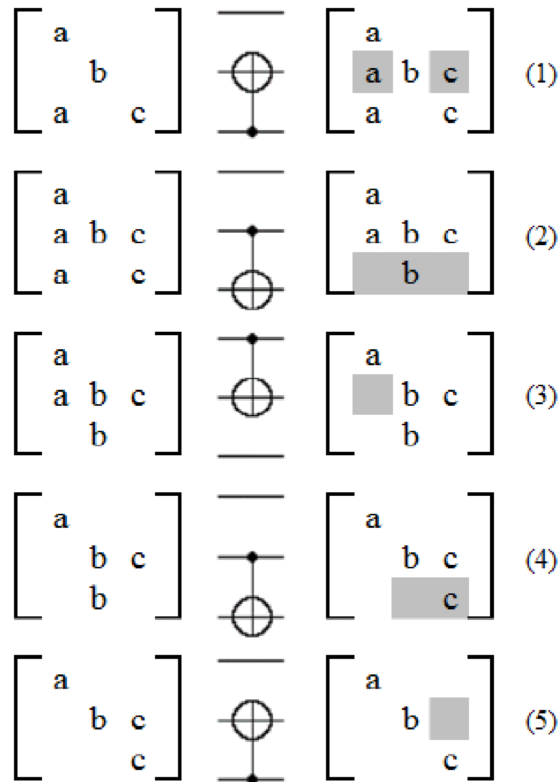


Figure 3.7: LNNGE synthesis of a "distance 2" CNOT gate.

3.3.7 Linear Nearest-Neighbor Alternating Elimination (LNNAE)

Another method I developed for linear reversible circuit synthesis is called Alternating Elimination. Alternating Elimination uses an approach of calculating an inverse by processing one diagonal element of a matrix at a time. This diagonal processing is achieved by adapting an approach used in "Algorithm 1" in which the final CNOT gate list is created from two separate lists. The first list that "Algorithm 1" generates corresponds to the first phase of Gaussian Elimination performed on the input function, and this produces a triangular matrix remainder function and a partial decomposition synthesis from output towards input. The second list that "Algorithm 1" generates corresponds to the first phase of Gaussian Elimination performed on the

transposed triangular matrix remainder function. This produces an identity matrix remainder and, after swapping target and control lines, a remainder decomposition synthesis from input towards output.

Alternating Elimination employs a similar approach at a more granular scale, producing up to *n-1* lists generated from output to input and *n-1* lists generated from input to output. These smaller lists represent adjacent subcircuits, so as they are generated they are appended to one of two associated master lists. Upon algorithm completion these two master lists are combined in the same way that "Algorithm 1" lists are combined.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 3.8: Matrix representation of LNNAE algorithm flow.

Figure 3.8 illustrates LNNAE's algorithm flow from the point of view of the outermost iteration when operating on a matrix representation of a *3×3* linear reversible circuit. The algorithm for LNNAE, which is a form of Alternating Elimination derived from LNNGE specifically for synthesis in the LNN model, is shown below.

```
FOR col FROM 0 TO N-2
  FOR row FROM N-1 TO col+1 by -1
    IF M[row][col] THEN
      IF NOT M[row-1][col] THEN
        CNOT(M[row]->M[row-1])
      CNOT(M[row-1]->M[row])
```

```
M = Transpose(M)
FOR row FROM N-1 TO col+1 by -1
  IF M[row][col] THEN
    IF NOT M[row-1][col] THEN
      TRANSPOSEDCNOT(M[row]->M[row-1])
    TRANSPOSEDCNOT(M[row-1]->M[row])
M = Transpose(M)
```

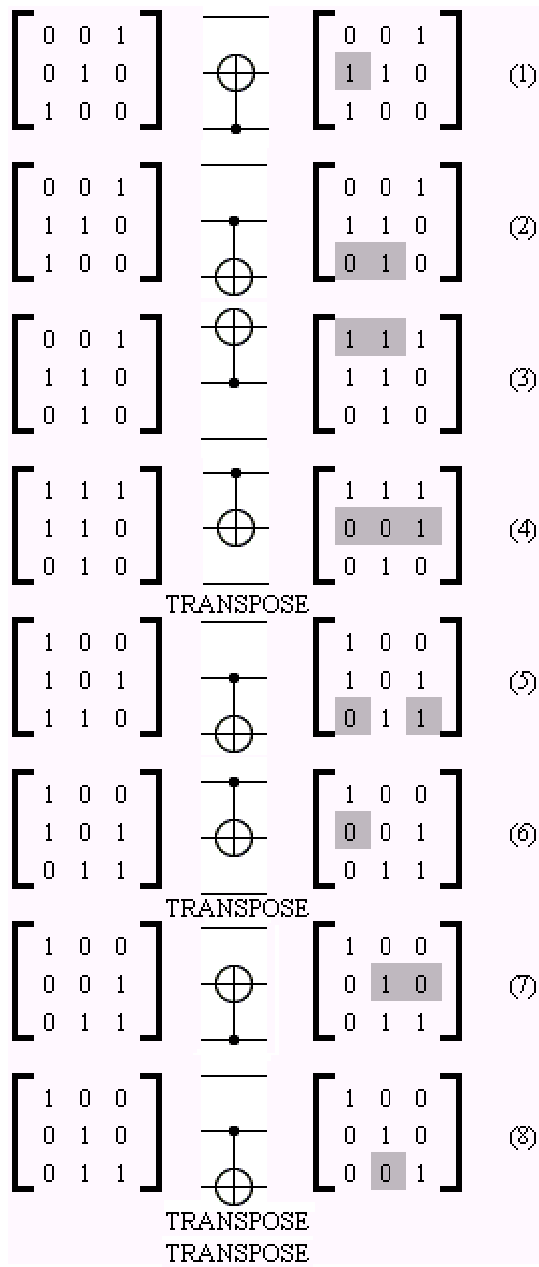


Figure 3.9: LNNAE synthesis of the 3×3 input reversal function.

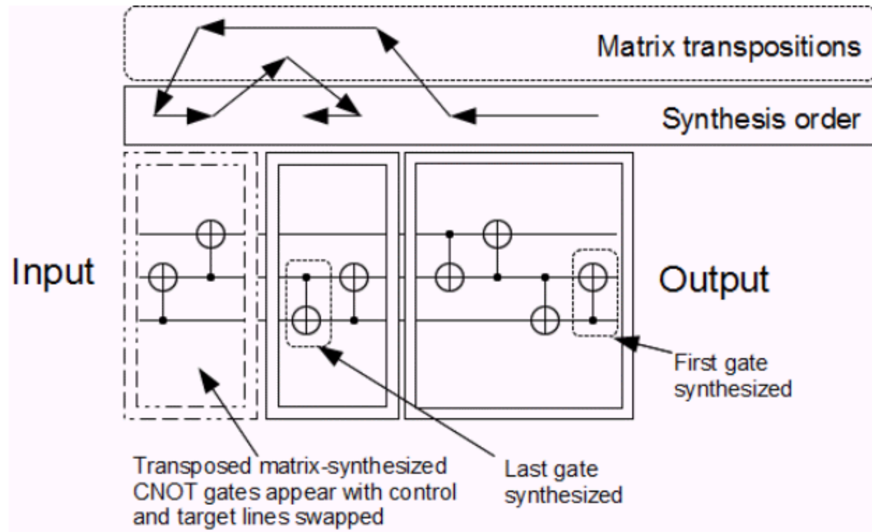


Figure 3.10: LNNAE algorithm flow when synthesizing of the 3×3 input reversal function.

Unless the computing hardware supports column operations on matrices, LNNAE is slower than LNNGE because of the repeated matrix transpositions. Unlike LNNGE, LNNAE has only one phase, and as a result both CNOT-up and CNOT-down gates appear throughout LNNAE synthesized CNOT gate lists. There are similarities between LNNGE and LNNAE, however. Both methods have a maximum of $2n^2-3n+1$ CNOT gates, as well as the ability to begin the algorithm operating on elements either in the first or the last column of the matrix. LNNAE retains this flexibility at each iteration of its outermost loop. The general form of Alternating Elimination has significantly more flexibility, as there are $n!$ different ways to process n diagonal elements.

In Figure 3.9 LNNAE synthesis of the 3×3 input reversal circuit is illustrated step by step which results in the linear reversible circuit in Figure 3.10. In each step an input matrix on the left has an elementary row operation performed, denoted by a

corresponding CNOT gate, which results in the output matrix on the right. Following the LNNAE algorithm, the first column from the left is examined and found to not be upper triangular. In step (1) a forward substitution from row 3 to row 2 is performed on the left matrix which results in the right matrix. In the right matrix modified bits in row 2 are displayed with a dark background. In step (2) a backward elimination from row 2 to row 3 is performed, modifying two bits in row 3. In step (3) a forward substitution from row 2 to row 1 is performed, modifying two bits in row 1. In step (4) a backward elimination from row 1 to row 2 is performed, modifying three bits in row 2 and resulting in the first column becoming upper triangular. Steps (1) through (4) correspond to the group of four CNOT gates from right to left in Figure 3.10 on the "Output" side of the circuit. The matrix is transposed and the first column is examined and found to not be upper triangular. In step (5) a backward elimination from row 2 to row 3 is performed, modifying two bits in row three. In step (6) a backward elimination from row 1 to row 2 is performed, modifying one bit in row 2 and resulting in the first row and column becoming identical to their corresponding identity matrix row and column. Steps (5) and (6) correspond to the two CNOT gates from left to right in Figure 3.10 on the "Input" side of the circuit with their control and target wires swapped. Next the matrix is transposed back to its original orientation and the second column is examined and found not to be upper triangular. In step (7) a forward substitution is performed from row 3 to row 2, modifying two bits in row 2. In step (8) a backward substitution from row 2 to row 3 is performed, modifying one bit in row 3 and resulting in the second column becoming upper triangular. Steps (7) and (8) correspond to the two CNOT gates from right to left in Figure 3.10 in the middle of the circuit. After transposing, examining the second column,

and transposing back the matrix is now equivalent to the identity matrix and no further processing is necessary. The last transpose operation is unnecessary and can be conditionally skipped to reduce computation time, but better results can be obtained by performing the last 5×5 portion of LNNAE synthesis by using an optimal LNN linear reversible circuit synthesis database.

The input reversal function is notable because it requires n^2-1 LNN CNOT gates to synthesize optimally and is the most costly LNN linear reversible circuit for circuits with relatively few wires, a result obtained from optimally synthesizing all possible LNN linear reversible circuits of up to five wires. The SWAP gate is the most well-known instance of the input reversal function, requiring $2^2-1=3$ CNOT gates.

3.3.8 Search Methods for LNNGE and LNNAE

Typically LNNGE and LNNAE do not give optimal results, and in some simple circuits can even be outperformed by "Algorithm 1" and Gaussian Elimination. Synthesizing a "distance 2" CNOT, which optimally requires only four adjacent CNOT gates, will result in five adjacent CNOT gates if the target is below the control line and the LNNGE "upper triangular matrix" method is used as is illustrated in Figure 3.7.

Two complementary search strategies were developed to improve performance for LNNAE and LNNGE. The first search method, called "Best of Eight", performs eight types of synthesis for the same circuit and uses the result with the lowest gate count. These eight types are:

1. Begin operations on the first column.
2. Begin operations on the last column.
3. Using the transposed matrix, begin operations on the first column.
4. Using the transposed matrix, begin operations on the last column.
5. Using the inverse matrix, begin operations on the first column.
6. Using the inverse matrix, begin operations on the last column.
7. Using the transposed inverse matrix, begin operations on the first column.
8. Using the transposed inverse matrix, begin operations on the last column.

The second search strategy employs a depth parameter to search for beneficial gate sequences. It is employed in two synthesis methods, "Linear Nearest-Neighbor Alternating Elimination with Depth" (LNNAED) and "Linear Nearest-Neighbor Gaussian Elimination with Depth" (LNNGED). In these methods a value of $depth=0$ invokes LNNAE or LNNGE respectively, and $depth>0$ invokes a recursive call to LNNAED or LNNGED with a value of $depth-1$. In LNNGED recursion is limited to the first phase of Gaussian Elimination when both CNOT-up and CNOT-down gates are available. The recursive search occurs when synthesizing subcolumns with patterns such as *101*, *1001*, *10001*, ... These patterns, having k zeros, can be optimally synthesized $k+1$ different ways to achieve patterns such as *111*, *1111*, *11111*, ... Each of the $k+1$ matrices is fully synthesized and the synthesis yielding the minimum gate count determines which of the $k+1$ subcolumn syntheses to use. Each recursive search calculates an upper bound of the total CNOT gates required for synthesis. This upper bound will always be equal to or lower than the total from the last recursive search. Thus LNNAED and LNNGED always

produce equal or lower CNOT gate counts than LNNAE and LNNGE respectively produce.

3.3.9 Linear Reversible Circuit Synthesis Tests

A set of tests was performed for circuits with 8 to 64 wires *(Table 3.1)*. Each set consisted of synthesizing 100 randomized linear reversible circuits with multiple methods. The synthesis results for Gaussian Elimination and "Algorithm 1" were converted to adjacent CNOT gate circuits for comparison. The n-wire 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.

When iterative deepening was used with LNNGED the gains decreased quickly and the computation time generally expanded geometrically. For example, performing a 64-wire "Best of Eight" LNNGED synthesis with $depth=1$ on an Intel® Celeron® Processor 450 @2.20 GHz typically took approximately 0.75 seconds, and performing the same synthesis using $depth=2$ took approximately 432 seconds and yielded a 2.0% lower gate count. Iterative deepening used on 16 wires yielded better gains with depth and faster computation times *(Table 3.2)*.

On average LNNAE and LNNGE performed similarly, and it appears that the method producing a lower gate count for any particular function is data-dependent. For 16×16 functions LNNAED performed better than LNNGED as depth increased, and at $depth=4$ LNNAED produced lower gate counts for a majority of functions. There was a similar performance difference between LNNGED and LNNAED when compared with

optimal syntheses of all 9999360 linear reversible functions of size 5×5. Using *depth=4* LNNGED achieved the optimal gate count 3921893 times whereas LNNAED achieved the optimal gate count 5300413 times. This performance difference demonstrates the benefit of the LNNAED approach of searching throughout the entire synthesis, as opposed to searching only in the first part of the synthesis as is the case with LNNGE.

Table 3.1: Comparisons of linear reversible circuit synthesis methods in the LNN model (average adjacent CNOT gate counts).

Wires	Gaussian Elimination	Algorithm 1	LNNGE	LNNAE	LNNGE Best of 8	LNNAE Best of 8
8	239.28	175.99	64.81	64.88	57.38	57.54
16	2268.16	1533.33	309.37	310.88	296.59	295.03
24	8169.97	5431.95	754.86	750.78	728.69	729.18
32	20045.8	10673.45	1381.54	1385.72	1356.1	1353.79
40	39608.31	20718.29	2205.94	2208.77	2172.72	2170.05
48	69233.85	36781.09	3226.51	3224.5	3186.12	3184.16
56	111316.22	61496.31	4430.49	4430.15	4387.38	4385.42
64	166681.3	96637.6	5836.04	5836.63	5778.1	5777.75

Table 3.2: Iterative deepening synthesis tests on 100 16×16 functions using "Best of Eight" search (average adjacent CNOT gate counts).

Depth	LNNGED (Average Adjacent CNOT Gates)	LNNAED (Average Adjacent CNOT Gates)
0	296.59	295.03
1	277.16	268.31
2	268.22	256.37
3	262.62	250.2
4	259.79	246.71

The optimal LNN linear reversible circuit synthesis databases for two to five wires were created by using a breadth-first search method explained later in 3.3.10. Using the frequency distribution of the optimal LNN linear reversible circuit synthesis database

for two to five wires in Table 3.3, an approximate optimal curve fit for average CNOT gate counts was computed to be $0.75\text{-}1.34n+0.86n^2$. A curve fit for LNNAE-based syntheses from 8 to 64 wires which were at depths which permitted synthesis within two minutes was computed to be $10\text{-}5.9n+1.5n^2$. Because the optimal synthesis curve fit is limited to five wires it is not expected to strongly correlate for larger n, and consequently curve fit comparisons could be viewed as weak. Considering that weakness, using $n=16$ the approximate optimal curve fit predicted an average CNOT gate count to be approximately 199 which can be compared with the Best of Eight LNNAED at *depth=4* result which was approximately 247. The $1.5n^2$ term in the LNNAE curve fit would seem to follow from having dense matrices require approximately n^2 operations between rows, each operation synthesizing a single CNOT gate resulting from a backward elimination 50% of the time and a pair of CNOT gates resulting from a forward substitution and backward elimination 50% of the time.

Table 3.3: Frequency distribution of all optimally synthesized LNN linear reversible functions up to size 5x5.

Adjacent CNOT Gates	2x2 Functions	3x3 Functions	4x4 Functions	5x5 Functions
0	1	1	1	1
1	2	4	6	8
2	2	10	22	38
3	1	22	69	148
4	0	44	202	526
5	0	44	492	1668
6	0	36	1039	4801
7	0	6	1944	12782
8	0	1	3089	31395
9	0	0	4113	70886
10	0	0	4276	148288
11	0	0	3174	286654
12	0	0	1485	510098
13	0	0	234	823464
14	0	0	13	1197022
15	0	0	1	1540264
16	0	0	0	1722606
17	0	0	0	1617314
18	0	0	0	1194802
19	0	0	0	622562
20	0	0	0	194966
21	0	0	0	18246
22	0	0	0	796
23	0	0	0	24
24	0	0	0	1

3.3.10 Related Linear Reversible Circuit Synthesis Methods

The initial LNN linear reversible circuit synthesis method created by the author was abandoned because it produced large gate counts and was slow. It was a heuristic method, using a penalty matrix like the one illustrated in Figure 3.11 to quantify how distant from the identity matrix an invertible linear system of equations matrix was. The

penalty was calculated as a sum of n column penalties. Each column penalty was largely a function of the most distant *1* above the diagonal matrix cell and the most distant *1* below the diagonal matrix cell, and to a lesser degree a function of column sparseness. Penalty matrices based on linear, exponential, quadratic, and cubic series were used. The matrices all had diagonal matrix cells with *1*s so that a penalty of n indicated the identity matrix. The quadratic penalty matrix appeared to perform slightly better on average, though in general which type of penalty matrix would synthesize the best was unpredictable.

Gate selection would follow from applying all legal elementary row operations to a problem matrix and choosing the resulting matrix which produced the lowest penalty. Whenever a penalty was discovered to reside at a local minimum, LNNGE would be used until the penalty dropped below the local minimum. This approach was later extended to select two or three CNOT gate operations at a significant time increase and without commensurate improvement on gate count.

$$\begin{bmatrix} 1 & 4 & 9 & 25 & \cdots \\ 4 & 1 & 4 & 9 & \cdots \\ 9 & 4 & 1 & 4 & \cdots \\ 25 & 9 & 4 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Figure 3.11: Penalty matrix based on distance from matrix diagonal.

In 4×4 LNN linear reversible circuit synthesis tests the heuristic penalty matrix method averaged approximately one less CNOT gate than LNNGE did, but for all cases 8×8 and above the heuristic method performed worse than LNNGE did, ultimately producing CNOT gate counts in excess of $2n^2$. The penalty matrix approach also became

increasingly slower, taking hours per each *64×64* function synthesis. Therefore the penalty matrix approach was abandoned, as was the attempt to improve synthesis by aiming for intermediate matrix patterns which indiscriminately clump around the diagonal.

I created two related pseudo-methods, Initial Gate Search and Truncated Initial Gate Search which inherited elements from the heuristic method exploration. Their approach is to find a lower-bound on CNOT gate count and maintain it while searching for better alternates. The term *pseudo-method* denotes reliance on other functions to actually do synthesis, and therefore they can be applied to the general model or the LNN model. In its simplest form Initial Gate Search iterates by employing two or more different syntheses on all matrices within one elementary row operation of a problem matrix, applying whichever elementary row operation corresponds to the synthesis with the lowest gate count, and, if the matrix is not the identity matrix, repeating the same treatment for the resulting matrix. If any of the methods use transposed matrices, as in "*Best of Eight" LNNGED depth=1* for example, a complication can arise whenever the best result comes from a synthesis method which uses a transposed matrix. In this case after the respective elementary row operation is applied, the matrix is transposed, the Initial Gate Search function recursively calls itself, and the resulting CNOT gate list is reversed and transposed (i.e. targets and controls are swapped). A deeper searching form of Initial Gate Search involves generating all matrices within two or even three elementary row operations of the problem matrix and performing two or more different syntheses on all of those, then choosing the best elementary row operation and repeating. While this approach is the most time-consuming and may not be practical at dimensions

above 16×16, it was the most efficient synthesis method for random 8×8 LNN linear reversible circuits producing an average adjacent CNOT gate count of 50.375 at a fraction of a second per synthesis.

The author observed that the largest improvements on gate count usually happened early in synthesis, especially during the first three elementary row operations. This led to a faster but less efficient *pseudo-method* called Truncated Initial Gate Search. In a Truncated Initial Gate Search a small number of iterations, typically one to three, of Initial Gate Search are performed with one set of synthesis methods, and the resulting remainder problem matrix is synthesized by another set of synthesis methods. A version of Truncated Initial Gate Search was able to scale up to larger circuits effectively, and on random 32×32 LNN linear reversible circuits produced an average gate count of 1255.35 taking approximately 24 seconds per synthesis. These and other pseudo-method results are shown in Appendix D.

Because both Initial Gate Search and Truncated Initial Gate Search have a large number of configuration possibilities they do not lend themselves to straightforward testing. Configurations that work for one size circuit in reasonable durations can become significantly slower for higher dimension circuits and comparatively less effective for smaller sized circuits compared to synthesis methods with depth parameters. Nonetheless, based on testing in Appendix D it would seem that for any particular linear reversible function pseudo-method synthesis may yield the most efficient synthesis.

Another related synthesis method which can be used for comparison and post-processing is retrieving gate sequences from a database. The largest database for LNN linear reversible circuits is the writer's 5×5 optimal LNN linear reversible circuit

synthesis database. This database contains all 33,554,432 5×5 matrices, with the 9,999,360 unique invertible matrices flagged as valid and the remaining 23,555,072 singular matrices flagged as invalid. Matrices are converted to unsigned 25-bit integers by concatenating matrix rows from top to bottom, and these values serve as pointers into the database's 33,554,432 bytes. In each byte representing a valid matrix there will be either a CNOT gate value associated with it, CNOT-up gates denoted as an integer in the range $[1, 4]$ and CNOT-down gates denoted as an integer in the range $[5, 8]$, or the value 15 which is reserved for the identity matrix. When retrieved, the CNOT gate is applied to the matrix which generates another matrix and associated pointer; once this pointer points to the identity matrix the reserved value 15 will be returned and synthesis completes.

The formula used to verify that there are 9,999,360 unique invertible 5×5 matrices was taken from De Vos [35]. This number is much lower than the total number of matrices (2^{25}= 33,554,432) due to excluding singular matrices, such as matrices with zero-filled rows and matrices which contain linear combinations of zero-filled rows. The formula for counting the number of unique invertible 5×5 matrices is as follows:

$$9{,}999{,}360=(2^5-1)*(2^5-2)*(2^5-4)*(2^5-8)*(2^5-16)$$

My final successful approach to creating the optimal 5×5 LNN linear reversible circuit synthesis database is similar to that used to create chess endgame tablebases. Tablebases are commonly used in chess and other similar deterministic games to store optimal moves in board arrangements with small numbers of remaining pieces; the usual requirement on number of pieces is that the resulting number of possible arrangements must be small enough to fit on typical PC computer hard drives. In contrast my previous

work, which used a breadth-first search of all possible CNOT gate sequences to determine optimal LNN synthesis of 4×4 linear reversible circuits, had a worst case time of approximately 45 minutes. Because the depth of the breadth-first approach was near n^2 for the complex circuits and there are $2n-2$ possible LNN CNOT gates at each level, the search time of this method is approximately proportional to $4n^{n^2}$ in the LNN model and n^{2n^2} in the general model. The tablebase approach proved significantly faster, calculating all 5×5 linear reversible circuits in approximately three seconds. Because the tablebase approach requires searching through 2^n matrices n^2 times, its computation time is proportional to $(2n-2)n^2 2^n$ in the LNN model gates and $n^4 2^n$ in the general model. The tablebase algorithm for the optimal 5×5 LNN linear reversible circuit synthesis database can be summarized as follows:

1. Set all database entries to zero which corresponds to a singular matrix.
2. Set the entry corresponding to the identity matrix to its corresponding reserved value OR'ed with a flag indicating that the entry requires expansion during phase “A”.
3. Loop until all expansions have completed.

    Phase “A”. Iteratively apply all possible CNOT gates on all matrices flagged for expansion during phase “A”. On all resulting entries which are zero assign the CNOT gate which connects it back to the entry being expanded OR the flag indicating expansion is required during phase “B”. Upon completion clear all phase “A” expansion flags from all entries.

    Phase “B”. Iteratively apply all possible CNOT gates on all matrices flagged for expansion during phase “B”. On all resulting entries which are

zero assign the CNOT gate which connects it back to the entry being expanded OR the flag indicating expansion is required during phase “A”. Upon completion clear all phase “B” expansion flags from all entries.

4. Verify that the number of invertible matrices is correct.

3.4 Affine-Linear Reversible Circuit Synthesis

As was mentioned previously, affine-linear reversible circuits can be represented by the equation $Y=MX \oplus B$ using Boolean multiplication and addition. A practical, scalable approach to affine-linear reversible circuit synthesis is to separately perform a linear reversible circuit synthesis on M, the invertible linear system of equations portion of the function to be synthesized, followed by an inverter network synthesis on B, the affine portion. This inverter network synthesis could be treated trivially as a subcircuit following the linear reversible circuit which contains at most n gates. A more efficient synthesis for both the general model and LNN model can be made by starting with an efficient linear reversible circuit and then performing an inverter network synthesis by searching all relevant NOT gate placements; relevant placements are on wires preceding CNOT gate controls or following the linear reversible circuit. Considering there may be $O(n^2)$ placements which must be propagated through $O(n^2)$ CNOT gates, searching all single NOT placements takes $O(n^4)$ time. To search all placements of two NOT gates would take $O(n^6)$ time, three NOT gates would take $O(n^8)$ time, etc. If this search is performed using two or more NOT gates and the search fails to fully synthesize the desired affine-linear function, an iterative approach would be to examine the remainder circuit which resulted in the closest approximation to the desired function and choose the

corresponding NOT gate placement that individually was closest to the desired function. In this context the NOT gate placements can be represented by a Boolean vector B', the desired function can be represented by B, and the closeness to the desired function is a count of 0s in the vector $B' \oplus B$.

One property discovered in LNN model synthesis that may also apply to the general model synthesis is that two different, functionally equivalent linear reversible circuit syntheses with equal gate counts may result in slightly different affine-linear reversible circuit NOT gate counts. This applies to optimal LNN linear reversible circuits and was discovered in the study that follows.

3.4.1 Optimal LNN Affine-Linear Reversible Circuit Synthesis Study of the 4×4 Input Reversal Circuit

The goal of this study was to determine how many different optimal syntheses of the 4×4 input reversal function there were and how they differed from one another when serving as the foundation for an LNN affine-linear reversible circuit synthesis. The results have bearing on memory requirements for storing optimal LNN affine-linear reversible circuit synthesis databases. If it could have been proven that any optimal LNN linear reversible circuit can serve as a foundation for an optimal LNN affine-linear reversible circuit, then computing and storing all 6×6 optimal LNN affine-linear reversible circuits would require $2^{6\times6}=64$GB of memory. Since this study disproves the relation, it appears that storing all 6×6 optimal LNN affine-linear reversible circuits would require $2^{6\times7}=4$TB of memory to account for the six rows of affine vector B. Using the computation times from the hard drive-based and RAM-based versions of the optimal

5×5 LNN linear reversible circuit synthesis database project as a guide, if the aforementioned 4TB memory is a hard drive, then computation can be expected to run 700 times slower than if it were RAM, putting it in the realm of several years.

The approach was to first determine every possible optimal LNN gate sequence for the input reversal circuit. This required first creating an optimal database and then using it to perform a depth-first search working backwards from the input reversal function to all neighboring functions which corresponded to circuits that, when optimally synthesized, required one less CNOT gate. Each successful search which resulted in a 15 CNOT gate count was stored in a synthesis list. For each of these syntheses, a collection of derivative circuits was created by testing all possible placements of a single NOT gate. Given that there are four wires, and knowing that an affine-linear reversible circuit will propagate at least one NOT gate to its output, the resulting output pattern of inverters will be one of $2^4-1=15$ possible derivative affine-linear reversible circuits. The informal proof for the property that adding a NOT gate to a linear reversible circuit results in an output with at least one NOT gate propagating through to the output can be made through exhaustive search of all *2×2* affine-linear reversible circuits. Each synthesis will then be associated with a particular 15-bit pattern, and statistics on the frequency of all 15-bit patterns will be tallied.

To test the functionality of the code a preliminary *2x2* input reversal function test was performed, meaning two wires were arranged to swap with each other and two remaining wires were arranged as straight-through. The SWAP gate is well known to have two optimal syntheses of three CNOT gates. It is fairly simple to calculate the different affine possibilities resulting from the addition of a single NOT gate to the

circuit, four obvious ones from placing an inverter on each output and another which results in inverted output on both wires which swap. The computational results were as expected, finding two optimal syntheses of the function with each synthesis being a single NOT gate away from the expected five affine-linear reversible circuits. Four of these patterns were the trivial case of placement of a NOT gate on each of the four wires following the linear reversible circuit, and one of these patterns corresponded to inserting a single NOT gate prior to the last CNOT gate's control line.

Next I tested the 4×4 input reversal circuit. With four lines and four NOT gates it follows that any linear reversible circuit can form the basis of $2^4 = 16$ different affine-linear reversible circuits, and because one of these is the trivial case which uses zero NOT gates only 15 affine-linear reversible circuits are of interest. The 4×4 input reversal circuit test results indicated that there were *122,256* unique optimal LNN syntheses. By adding a single NOT gate in all locations of each of the *122,256* unique circuits I discovered that each linear circuit was capable of becoming 10 out of the possible 15 affine-linear reversible circuits. Furthermore these 10 out of 15 affine-linear reversible circuit were in four different categories. These categories are shown in Table 3.4 where 10 columns appear with dark backgrounds which represent each category's reachable equivalent affine-linear circuits. These categories covered all possible configurations. Therefore using a set of four unique circuit syntheses, with one representative synthesis from each of the four categories, all 15 affine-linear reversible circuits can be synthesized with one NOT gate.

Table 3.4 An example of how optimal linear reversible circuit synthesis of the input reversal function does not necessarily lead to an optimal affine-linear reversible circuit.

3440 input reversal circuits can become these 10 affine reversible circuits using one NOT gate:

Wire	Affine Vector														
D0	⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1
D1	⊕1	⊕1			⊕1	⊕1			⊕1	⊕1			⊕1	⊕1	
D2	⊕1	⊕1	⊕1	⊕1					⊕1	⊕1	⊕1	⊕1			
D3	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1							

26516 input reversal circuits can become these 10 affine reversible circuits using one NOT gate:

Wire	Affine Vector														
D0	⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1
D1	⊕1	⊕1			⊕1	⊕1			⊕1	⊕1			⊕1	⊕1	
D2	⊕1	⊕1	⊕1	⊕1					⊕1	⊕1	⊕1	⊕1			
D3	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1							

26516 input reversal circuits can become these 10 affine reversible circuits using one NOT gate:

Wire	Affine Vector														
D0	⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1
D1	⊕1	⊕1			⊕1	⊕1			⊕1	⊕1			⊕1	⊕1	
D2	⊕1	⊕1	⊕1	⊕1					⊕1	⊕1	⊕1	⊕1			
D3	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1							

65784 input reversal circuits can become these 10 affine reversible circuits using one NOT gate:

Wire	Affine Vector														
D0	⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1		⊕1
D1	⊕1	⊕1			⊕1	⊕1			⊕1	⊕1			⊕1	⊕1	
D2	⊕1	⊕1	⊕1	⊕1					⊕1	⊕1	⊕1	⊕1			
D3	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1	⊕1							

4

LNNAE Hardware Design

4.1 Overview

This section discusses the development of a scalable RTL SystemVerilog description of an LNNAE computer. The testing methodology was to use both directed tests and randomized tests comparing LNNAE as a C function, a behavioral SystemVerilog implementation, and an RTL SystemVerilog implementation.

The LNNAE synthesis algorithm was chosen for hardware implementation because synthesis tests of randomized 32×32 matrices showed that at equal depths LNNAED gate counts were smaller than LNNGED gate counts 80-90% of the time. Examining the software implementation revealed that the majority of the processing time in LNNAED was spent performing recursive calls to LNNAE to compare CNOT gate counts. Also, LNNAE employs $2n-2$ matrix transpositions each of which take $O(n^2)$ time. LNNGE uses bit tests and 64-bit XOR operations, making it $O(n^2)$; LNNAE similarly has bit tests and 64-bit XOR operations but also one matrix transpose operation per each row and each column, making it $O(n^3)$.

A custom hardware implementation of LNNAE would yield improved performance if the transpose operation was either eliminated or performed in a single clock cycle, thus achieving $O(n^2)$. Also, a portion of LNNAE computation time was dedicated to creating CNOT gate sequences which LNNAED discarded. Therefore the hardware LNNAE units were simplified to return only a CNOT gate count. For simplicity of debugging and testing, the LNNAE units were limited to synthesizing LNN linear reversible circuits of dimension 4×4.

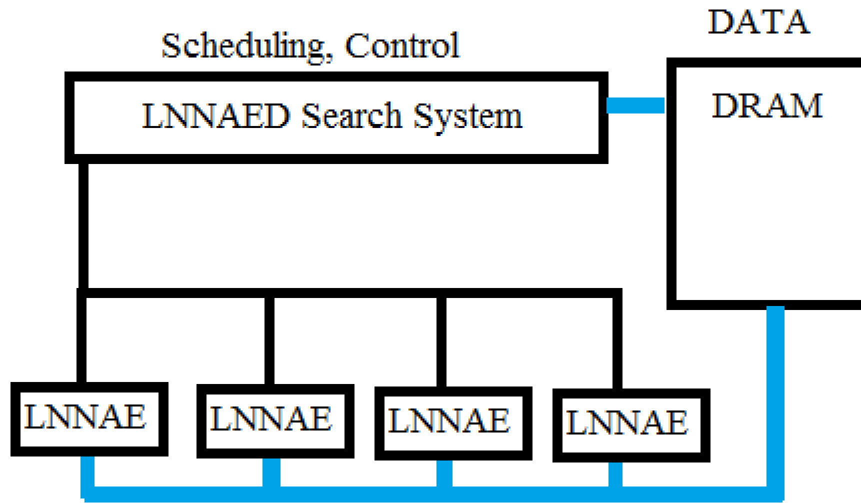


Figure 4.1: Block diagram of proposed LNNAED recursive search system.

A future goal is to make multiple LNNAE blocks run in parallel as coprocessors of a recursive LNNAED search system shown in Figure 4.1. The main LNNAED controller would run the recursive parts of the LNNAED for *depth>0*, produce matrices for *depth=0*, and manage LNNAE coprocessor scheduling. The LNNAE coprocessors would compute the *depth=0* LNN CNOT gate counts.

4.2 Initial design

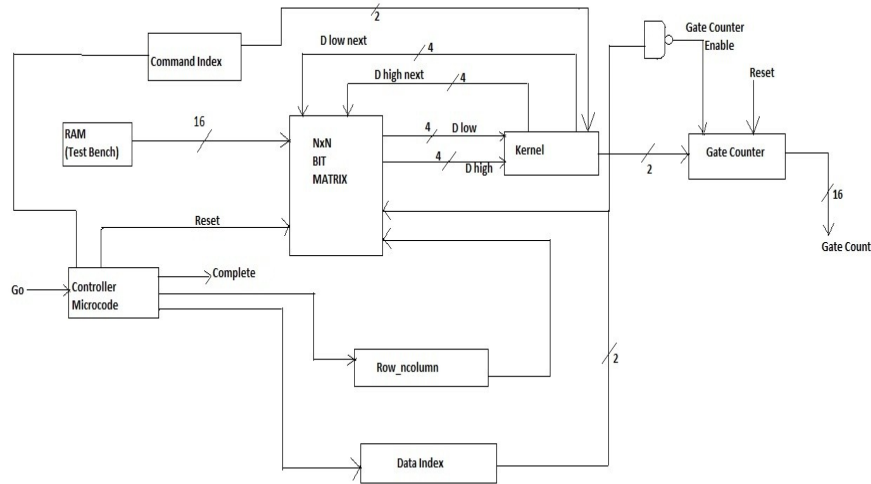


Figure 4.2: Block diagram of the author's initial LNNAE coprocessor design using dual row and column matrix access lines.

The block diagram of the initial $O(n^2)$ LNNAE coprocessor design is shown in Figure 4.2. The primary goal of this design was to minimize the number of clock cycles required for processing a matrix. To achieve this, matrix data was stored in a two-dimensional array which had dual row and column access. In each cycle of computation a pair of rows or columns would be fetched, modified, and stored. Modification occurred in the kernel which permitted a maximum of two CNOT gates to be synthesized in one cycle, as is shown in Table 4.1. In the table input registers REG1 and REG2 represent an *n*-bit value which could come from either a matrix row or column. The variable *Row_Column_Index* held the index of the control bits in each fetched row and column, and these control bits were used to determine if forward substitution or backward elimination was required.

Table 4.1: Truth table for initial LNNAE coprocessor kernel block.

REG1[Row_Column_Index]	REG2[Row_Column_Index]	REG1*	REG2*	TotalGates Increment
0	0	REG1	REG2	0
0	1	REG1 xor REG2	REG1	2
1	0	REG1	REG2	0
1	1	REG1	REG1 xor REG2	1

Using this approach an $n \times n$ LNNAE unit could establish the first column and row in *2n-2* cycles, the second column and row in *2n-4* cycles, the next in *2n-6* cycles, etc. Specifically the 4×4 LNNAE unit took *6+4+2=12* clock cycles.

4.3 Hardware Implementation Discussion

After a working proof-of-concept of the dual row and column LNNAE unit was made in VHDL, a scaling issue became evident. It appeared that the area required for a sizable LNNAE unit, one that could synthesize reversible circuits of 64 wires and above, would depend mainly on the size of the n×n matrix block. In the dual row and column LNNAE unit design each row and column used *n* parallel routing channels, and assuming a target FPGA with *k* routing channels both vertically and horizontally adjacent to each logic element (LE), the FPGA area that would be required for large LNNAE units would be over n^4/k^2 LEs. This includes two horizontal and two vertical routing channels used for control and data-in. In several class discussions Robin Marshall, Dr. Marek Perkowski, and the author discussed alternative designs. Dr. Perkowski suggested using a wired-OR to combine *n* parallel routing channels, but from researching FPGAs it appeared that only some FPGAs supported wired-OR but exclusively on external pins. A compact alternative to using a wired-OR would be to employ a chain of *n-1* two-input OR gates,

but for LNNAE units of $n=64$ or larger this would lead to a significant propagation delay. While it was never implemented, a reasonable compromise seemed to be use of a multilevel $k\times1$ OR network which would be fast and use only $k\cdot log_k n$ routing channels per column and per row. Using $n=64$ and $k=4$ (vertical and horizontal routing channels per LE) in a three level OR-OR-OR network of 4×1 OR gates, 12×12 routing channels per matrix cell would suffice at a cost of only three combinational logic delays. Provided five combinational logic delays would be acceptable, this could be reduced to 10×10 routing channels, leaving enough for the data-in and control and still fit into an overall area of 3×3 LEs per matrix cell.

Robin Marshall suggested a mixture of a systolic approach and a two-dimensional shift register approach shown in Figure 4.3. In this approach there are two ancillary columns stored in an adjacent two n-bit word block to the right of the matrix and two ancillary rows stored in an adjacent two word block below the matrix. Pairs of matrix columns or rows are shifted through their respective adjacent blocks. This process acts similarly to the kernel in the dual row and column design, performing the required logic which represents forward substitution and backward elimination. Combinational logic inside these adjacent blocks drives a shared two-bit *CNOTs* gate count line which must indicate zero *CNOTs* gates during matrix alignment (no-operation) cycles. For a complete iteration through a row or column, $n+2$ clock cycles are necessary to align the matrix, and subsequently an entire synthesis requires $2(n+2)(n-1)=2n^2+2n-4$ cycles.

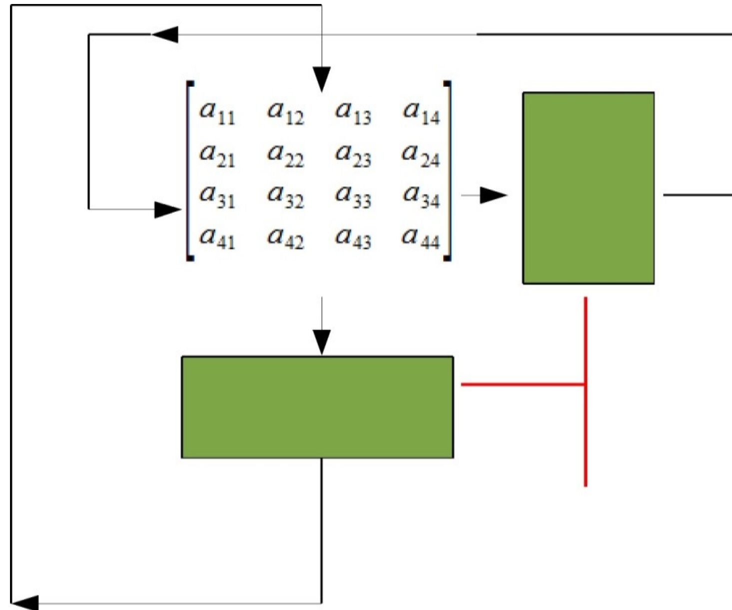


Figure 4.3: Systolic two-dimensional shift register LNNAE design (suggested by Robin Marshall).

Comparing the two designs led to some interesting conclusions. The dual row and column approach had a running time of approximately n^2-n cycles and an approximate implementation area proportional to $9n^2+12n+4$ LEs, ignoring nonmatrix blocks. The systolic two-dimensional shift register approach had a running time of approximately $2n^2+2n-4$ cycles and required an approximate implementation area of n^2+4n+4 LEs, making it roughly double the running time and roughly one-ninth the area. Considering that nine systolic two-dimensional shift register units running in parallel occupy approximately the same area as one dual row and column unit does, for $n=64$ wires the systolic two-dimensional shift register throughput is over four times the dual row and column throughput. Furthermore the systolic two-dimensional shift register approach

could be employed on FPGAs with fewer routing channels per LE than the the dual row and column approach could.

Because the systolic two-dimensional shift register approach to LNNAE synthesis had better throughput and scaling potential, it was chosen as a starting point for research. The author redesigned the systolic two-dimensional shift register to shorten computation time and simplify control. The first change came from an observation that the logic could be simplified if the next-state version of the upper row or column H^* was made to be a function of the next-state version of the lower row or column L^* shown in Table 4.2.

Table 4.2: Truth table for redesigned systolic LNNAE kernel.

L[0] input	H[0] input	L*	H*	TotalGates Increment
0	0	L	H	0
0	1	L XOR H	L* XOR H	2
1	0	L	H	0
1	1	L	L* XOR H	1

The second and more significant change was the redesign of the systolic two-dimensional shift register with fewer flip-flops to achieve a faster runtime. The key to this design was fixing the location of the control bits responsible for determining elementary row operations on the outside edge of the matrix; specifically for matrix $M_{(1,1)-(n,n)}$, cells $M_{(n-1,1)}$ and $M_{(n,1)}$ would now be fixed elementary row operation control bits and $M_{(n,n-1)}$ and $M_{(n,n)}$ would now be fixed elementary column operation control bits. By fixing the location of the control bits and shifting the matrix up once and left once on each LNNAE iteration, two dummy cycles previously required for matrix alignment could be eliminated. This created a complicated dataflow and hardware redesign. In the new design L^* would be computed by n combinational logic blocks (*combL*) operating in

parallel and located inside the matrix. The L^* outputs would now drive the inputs of the last row and column LEs. Similarly H^* would be computed by n combinational logic blocks (*combH*) operating in parallel which would now drive the inputs of the first row and column LEs.

Once the new dataflow and hardware design was created and passed rudimentary tests, it was integrated into a larger test which was done chiefly by Addy Gronquist with some assistance from the author. The larger design implemented a behavioral SystemVerilog module, an RTL SystemVerilog module, and, using DPI calls to the original C code version of LNNAE, a "Golden model" SystemVerilog module. A testbench was created to compare these three approaches which employed a few directed tests and many randomized tests.

Figure 4.4 shows the internal matrix structure, and how shifting can occur in either the vertical or the horizontal direction. Figure 4.5 shows a simplified block diagram describing the logic surrounding the matrix. An n-bit wide 2×1 multiplexer bank permits switching between loading new matrix values and computation of elementary row operations by connecting the output of the *combH* bank to the input of the top row of the matrix. The *CNOTs* block outputs a two-bit value which ranges between zero and two, per Table 4.2, and is accumulated in the *totalCNOT* block. Also shown are the control lines *enable*, *loading*, *shiftdirection*, and *loadValue* which are currently driven externally from the testbench.

Figure 4.6 illustrates the block diagram of the testbench. The testbench output was a scoreboard which aggregated the directed and randomized test results which is shown in Section 4.5. All tests passed without errors.

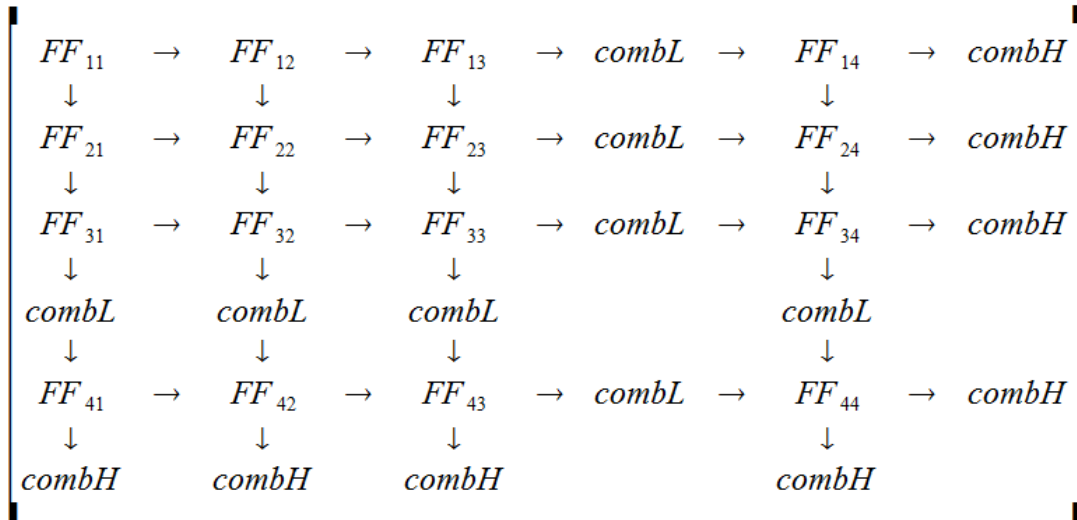


Figure 4.4: Redesigned systolic two-dimensional shift register LNNAE matrix.

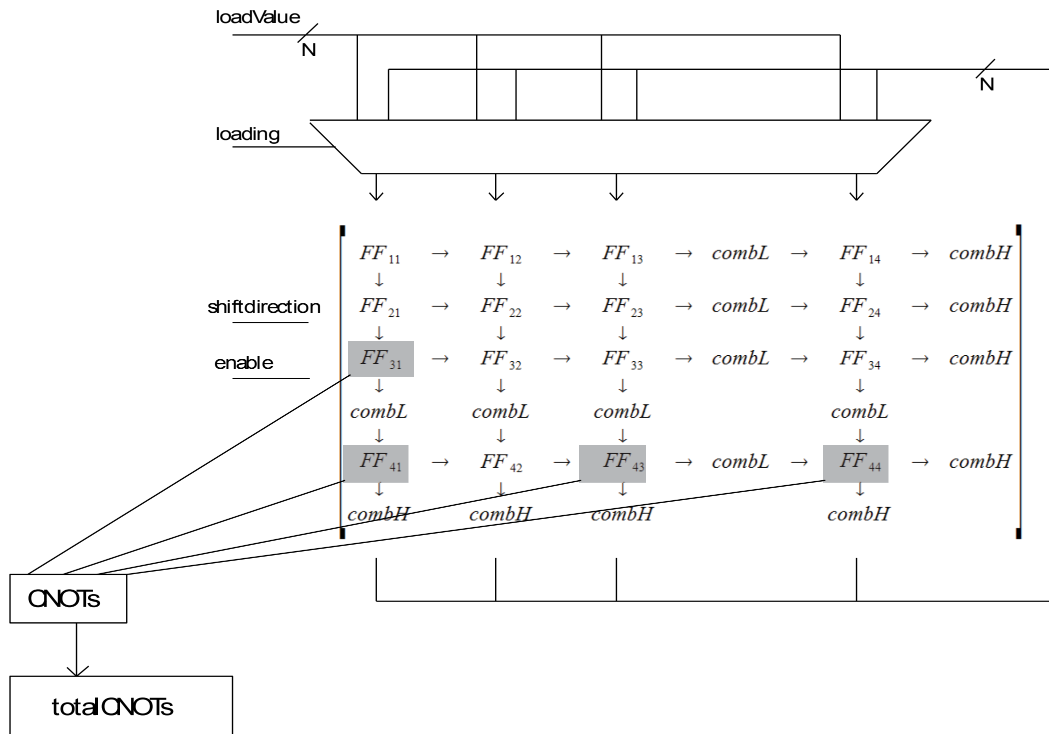


Figure 4.5: Redesigned systolic two-dimensional shift register LNNAE system.

4.4 Systolic Implementation

The full source code for the redesigned systolic two-dimensional shift register LNNAE coprocessor and its test suite is available in the Appendix. The test suite which was organized and implemented chiefly by Addy Gronquist tested my behavioral, RTL, and C language LNNAE implementations against one other. The following code is from the behavioral SystemVerilog implementation of the systolic two-dimensional shift register LNNAE computer and shows the redesigned dataflow:

```
// outer loop, iterate N-1 times
for (i = N; i > 1; i--) begin
  //row processing phase, iterate N-1 times
  for(row = N; row > 1; row--) begin
    if (m[N-1][0]) begin
      count++;
      if (!m[N-2][0])
        count++;
    end
    // first assign combinational logic for L and H
    if (!m[N-2][0] && m[N-1][0])
      L = m[N-2]^m[N-1];
    else
      L = m[N-2];
    if (m[N-1][0])
      H = L^m[N-1];
```

```
  else
    H = m[N-1];
  //perform new assignments
  m[N-1] = L;
  for(j = N - 2; j > 0; j--) begin
    m[j] = m[j-1];
  end
  m[0] = H;
end

for(column = N; column > 1; column--) begin
  //column processing phase, iterate N-1 times
  if (m[N-1][N-1]) begin
    count++;
    if (!m[N-1][N-2])
      count++;
  end
  if (!m[N-1][N-2] && m[N-1][N-1])
    L = {m[0][2]^m[0][3], m[1][2]^m[1][3],
        m[2][2]^m[2][3], m[3][2]^m[3][3]};
  else
    //4x4 matrix specific code
    L = {m[0][2], m[1][2], m[2][2], m[3][2]};
```

```
    if (m[N-1][N-1])
      H = L^{m[0][3], m[1][3], m[2][3], m[3][3]};
    else
      H = {m[0][3], m[1][3], m[2][3], m[3][3]};
    //perform new assignments
    for(k = 0; k < N; k++) m[k][N-1] = L[k];
    for(j = N - 2; j > 0; j--) begin
      for(k = 0; k < N; k++)
        m[k][j] =m[k][j-1];
    end
    for(k = 0; k < N; k++) m[k][0] = H[k];
  end
end
```

The following code is from the RTL SystemVerilog implementation of the systolic two-dimensional shift register LNNAE computer and defines the matrix in Figure 4.4:

```
generate //instantiate N-1 by N-1 section and ends
  for (i = 0; i < N-1; i++) begin: rowvar
    for (j = 0; j < N-1; j++) begin: colvar
      TwoDShiftCell1 a(h[i][j], v[i][j], shiftdirection,
          clock, h[i][j+1], v[i+1][j]);
    end
```

```
    //vertical wire glue
    combL cvlow(v[N-1][i], v[N][i], rowcombLcontrol,
        vCombLout[i]);
    TwoDShiftCell1 abottom(h[N-1][i], vCombLout[i],
        shiftdirection, clock, h[N-1][i+1], v[N][i]);
    combH cvhigh(vCombLout[i], v[N][i], rowcombHcontrol,
        vCombHout[i]);
    mux2to1 m21(vCombHout[i], datain[i], loading,
        v[N+1][i]);
    assign v[0][i] = v[N+1][i];
    //horizontal wire glue
    combL chlow (h[i][N-1], h[i][N], columncombLcontrol,
        hCombLout[i]);
    TwoDShiftCell0 aend(hCombLout[i], v[i][N-1],
        shiftdirection, clock, h[i][N], v[i+1][N-1]);
    combH chhigh(hCombLout[i], h[i][N], columncombHcontrol,
        hCombHout[i]);
    assign h[i][0] = hCombHout[i];
  end
endgenerate
```

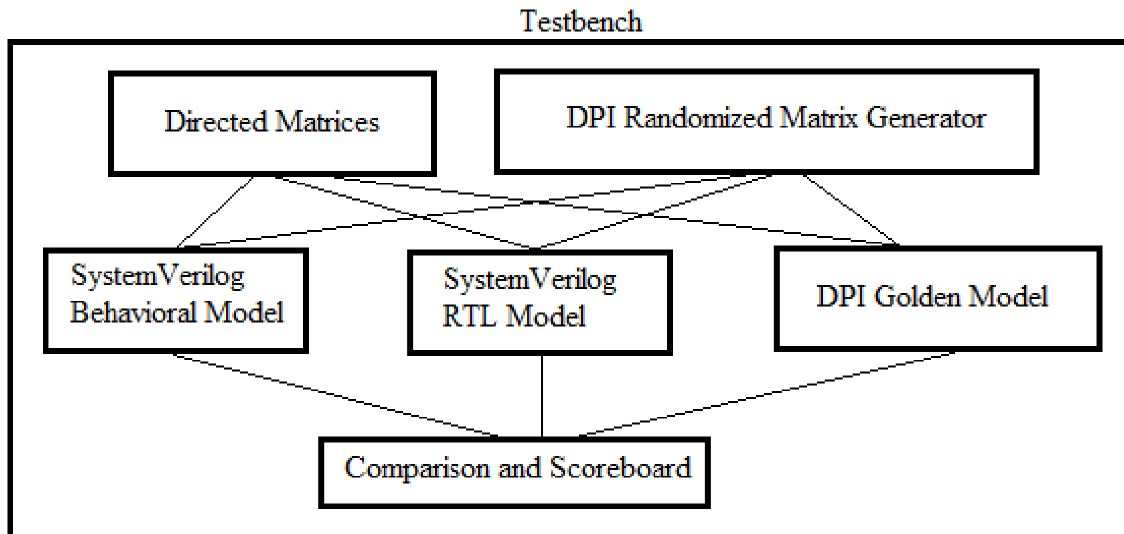


Figure 4.6: Organization of the LNNAE system testbench (designed by Addy Gronquist).

The following code is from the testbench for systolic two-dimensional shift register LNNAE system:

```
//SystemVerilog side:
import "DPI-C" function shortint unsigned random4by4();
import "DPI-C" function shortint unsigned run4by4(shortint
unsigned array);
import "DPI-C" function void seedRNG();

// C side:
unsigned short run4by4(unsigned short array);
unsigned short random4by4(void);
void seedRNG(void);
```

```c
// C Code LNNAE Golden Model
unsigned short run4by4(unsigned short array)
{
    LRCSInitialize();
    uint64_t inputreversal[4];
    int N = 4, list[64];
    int numGates = 0;
    unsigned short temp = array;
    for (int i = 0; i < 4 ; i++)
    {
        temp = array;
        temp = (temp & (0xf << 4*i)) >> 4*i;
        inputreversal[3-i] = temp;
    }
    numGates = LNNRGE_U(4, inputreversal,list);
    return numGates;
}
```

4.5 Results

```
# Error Count:
#      0
# gateCount Coverage
```

```
# gateCount =            0 covered         9 times
# gateCount =            1 covered        24 times
# gateCount =            2 covered        94 times
# gateCount =            3 covered       265 times
# gateCount =            4 covered       633 times
# gateCount =            5 covered      1276 times
# gateCount =            6 covered      2422 times
# gateCount =            7 covered      3858 times
# gateCount =            8 covered      5650 times
# gateCount =            9 covered      7636 times
# gateCount =           10 covered      9287 times
# gateCount =           11 covered     10334 times
# gateCount =           12 covered     10397 times
# gateCount =           13 covered      9444 times
# gateCount =           14 covered      7451 times
# gateCount =           15 covered      5428 times
# gateCount =           16 covered      3189 times
# gateCount =           17 covered      1665 times
# gateCount =           18 covered       703 times
# gateCount =           19 covered       199 times
# gateCount =           20 covered        35 times
# gateCount =           21 covered         5 times
```

4

Conclusion

In this thesis my linear reversible circuit synthesis algorithms LNNGE and LNNAE were presented, as well as "Best of Eight" and depth search methods. These algorithms can synthesize LNN linear reversible circuits with hundreds of wires generating no more than $2n^2-3n+1$ adjacent CNOT gates, as well as serve as a foundation for LNN affine-linear and permutation syntheses. The LNNAE algorithm stems from a more general algorithm called Alternating Elimination which I developed in order to expand the search space for minimizing the total number of elementary row operations needed to compute an inverse. Alternating Elimination employs the matrix transposition approach introduced in "Algorithm 1" [21] in order to solve for one diagonal matrix cell at a time, and the resulting elementary row operation sequence creates a bidirectional linear reversible circuit synthesis. Furthermore Alternating Elimination has no second phase of backward elimination operations as is the case with Gaussian Elimination, and as a result LNNAED has a larger search space and tends to outperform LNNGED as depth increases. When a deeper search is desired the methods Initial Gate Search and Truncated Initial Gate Search may further reduce CNOT gate counts. Future work in this area would be to investigate the adaptation my LNN-based synthesis methods to improve general linear reversible circuit synthesis.

Through LNN linear reversible circuit synthesis tests of randomized linear functions for up to 64 wires it was discovered that my methods had an average adjacent CNOT gate count that was asymptotic to $1.5n^2$. The tests indicated that for LNN linear reversible circuits up to 16 wires an adjacent CNOT gate count of approximately n^2 is

usually possible. I created a 5×5 optimal LNN linear reversible circuit synthesis database, and studying the results indicated that for 5 wires and below the exact upper bound is n^2-1 adjacent CNOT gates.

A redesign of Robin Marshall's systolic two-dimensional shift register LNNAE system was presented which used $2n-2$ fewer cycles per synthesis. Future work in this area would be to employ multiple LNNAE systems in parallel as part of a larger LNNAED system.

References

1. Michio Kaku. "Michio Kaku: Tweaking Moore's Law and the Computers of the Post-Silicon Era." Internet: http://www.youtube.com/watch?feature=player_embedded&v=bm6ScvNygUU, April 13, 2012 [Jan. 23, 2013].

2. Antoine Bérut, Artak Arakelyan, Artyom Petrosyan, Sergio Ciliberto, Raoul Dillenschneider, Eric Lutz. (Mar. 7 2012). "Experimental Verification of Landauer's Principle Linking Information and Thermodynamics." *Nature*. Vol 483, pp. 187-190, 2012. Available http://www.nature.com/nature/journal/v483/n7388/full/nature10872.html [Jan. 26, 2013].

3. Rolf Landauer. "Irreversibility and Heat Generation in the Computing Process." *IBM Journal of Research and Development.* Vol. 5, pp. 183-191, 1961.

4. Charles H. Bennett. "Notes on Landauer's principle, Reversible Computation and Maxwell's Demon." *Studies in History and Philosophy of Modern Physics.* Vol. 34, pp. 501-510, 2003.

5. Graham P. Boechler, Jean M. Whitney, Craig S. Lent, Alexei O. Orlov, and Gregory L. Sniderb. (Sept. 7, 2010)."Fundamental limits of energy dissipation in charge-based computing." *Applied Physics Letters.* Vol. 97, 103502, 2010. Available: http://apl.aip.org/resource/1/applab/v97/i10/p103502_s1?isAuthorized=no [Dec. 11, 2012].

6. Dmitri Maslov. "Reversible Logic Synthesis." Phd. thesis, The University of New Brunswick, Canada, 2003.

7. Anonymous. "Quantum Random Number Generator." Internet: http://qrbg.irb.hr/, 2007 [Jan. 26, 2013].

8. Anonymous. "D-Wave, The Quantum Computing Company." Internet: http://www.dwavesys.com/en/dw_homepage.html, 2012 [Jan. 26, 2013].

9. Rodney Van Meter, Mark Oskin. "Architectural implications of quantum computing." *ACM Journal on Emerging Technologies in Computing Systems.* Volume 2 Issue 1, pp. 31-63, Jan. 2006.

10. Richard Hughes et al. "*A Quantum Information Science and Technology Roadmap.*" Internet: http://qist.lanl.gov/, Apr. 2, 2004. [Jan. 26, 2013]

11. Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, David Petrie Moulton. "A new quantum ripple-carry addition circuit." Internet: http://arxiv.org/pdf/quant-ph/0410184.pdf, Oct. 22, 2004 [Jan. 26, 2013]

12. Yuichi Hirata, Masaki Nakanishi, Shigeru Yamashita, Yasuhiko Nakashima. "An Efficient Method to Convert Arbitrary Quantum Circuits to Ones on a Linear Nearest Neighbor Architecture." *Third International Conference on Quantum, Nano and Micro Technologies.* pp. 26-33, 2009. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4782917 [Jan. 26, 2013]

13. Austin G. Fowler, Simon J. Devitt and Lloyd C. L. Hollenberg. "Implementation of Shor's Algorithm on a Linear Nearest Neighbour Qubit Array." *Quantum Information & Computation.* Volume 4, Issue 4, pp. 237-251, July 2004.

14. Mozammel H. A. Khan. "Cost Reduction in Nearest Neighbour Based Synthesis of Quantum Boolean Circuits." *Engineering Letters.* Vol.16, issue 1. Available:

http://www.engineeringletters.com/issues_v16/issue_1/EL_16_1_01.pdf, 2008 [Jan. 26, 2013].

15. Amlan Chakrabarti, Susmita Sur-Kolay, Ayan Chaudhury. "Linear Nearest Neighbor Synthesis of Reversible Circuits by Graph Partitioning." Internet: http://arxiv.org/pdf/1112.0564.pdf, Dec. 27, 2012 [Jan. 27, 2013].

16. Amlan Chakrabarti, Susmita Sur-Kolay. "Nearest Neighbour based Synthesis of Quantum Boolean Circuits." *Engineering Letters.* Vol.15, issue 2. Available: http://www.engineeringletters.com/issues_v15/issue_2/EL_15_2_26.pdf, 2007 [Jan. 27, 2013].

17. Marek Perkowski, Martin Lukac, Dipal Shah, Michitaka Kameyama. "Synthesis of quantum circuits in Linear Nearest neighbor Model using Positive Davio Lattices." *Facta universitatis - series: Electronics and Energetics.* Vol. 24, br. 1, pp. 71-87, Apr. 2011.

18. Mehdi Saeedi, Robert Wille, Rolf Drechsler. "Synthesis of Quantum Circuits for Linear Nearest Neighbor Architectures." *Quantum Information Processing.* Vol. 10, No. 3, pp. 355-377, 2011. Available: http://arxiv.org/pdf/1110.6412v2 [Jan. 27, 2013].

19. Donny Cheung, Dmitri Maslov, Simone Severini. "Translation Techniques Between Quantum Circuit Architectures." Workshop on Quantum Information Processing, December 2007. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.133.7479&rep=rep1&type=pdf [Jan. 27, 2013].

20. Oleg Golubitsky, Dmitri Maslov. "A Study of Optimal 4-Bit Reversible Toffoli Circuits and Their Synthesis." *IEEE Transactions on Computers.* Vol. 61, no. 9, pp.

1341-1353, Sept. 2012. Available:
http://www.computer.org/csdl/trans/tc/2012/09/ttc2012091341-abs.html [Jan. 26, 2013].

21. 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, March 2008. Available: http://arxiv.org/abs/quant-ph/0302002 [Nov. 8, 2011]

22. A. Bogdanov, M.C. Mertens, C. Paar, J. Pelzl, A. Rupp. "A Parallel Hardware Architecture for Fast Gaussian Elimination Over GF(2)." *14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp.237-248, Apr. 24-26, 2006. Available: http://cs.ucsb.edu/~koc/docs/j14.pdf [Jan. 27, 2013].

23. Richard Graham. Lecture, "Ion-trap Quantum Computing." Electrical and Computer Engineering Department, Fourth Avenue Building, Portland State University, Portland, Oregon. February 24, 2012.

24. Ben Schaeffer, Marek Perkowski. "Linear Reversible Circuit Synthesis in the Linear Nearest-Neighbor Model." *2012 42nd IEEE International Symposium on Multiple-Valued Logic,* pp. 157-160, May 14-16, 2012. Available:
http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6214801&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6214801 [Jan. 27, 2013].

25. Paul Dawkins. "Paul's Online Math Notes." Internet:
http://tutorial.math.lamar.edu/Classes/LinAlg/SpecialMatrices.aspx, 2003 [Jan. 26, 2013].

26. M. Nielsen, I. Chuang. "*Quantum Computation and Quantum Information.*" Cambridge, United Kingdom. Cambridge University Press, 2000.

27. Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter.

“Elementary gates for quantum computation.” *Phys. Rev. A 52*, pp. 3457–3467, 1995. Available: http://pra.aps.org/pdf/PRA/v52/i5/p3457_1 [Jan. 27, 2013].

28. Mehdi Saeedi, Igor L. Markov. "Synthesis and Optimization of Reversible Circuits - A Survey." Internet: http://arxiv.org/pdf/1110.2574v1, Oct. 12, 2011 [Jan. 27, 2013].

29. Lloyd N. Trefethen. “Three Mysteries of Gaussian Elimination.” *ACM SIGNUM Newsletter*. 1985. Available: http://www.cse.illinois.edu/courses/cs591mh/trefethen/Three_Mysteries.pdf [Jan. 27, 2013].

30. Ben Schaeffer, Marek Perkowski. "Linear Reversible Circuit Synthesis Methods for the Linear Nearest-Neighbor Model." Unpublished manuscript, Jan. 11 2013.

31. I. L. Shende, S. S. Bullock, I. L. Markov. “Synthesis of Quantum Logic Circuits.” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.* Vol.25, no. 6, pp. 1000-1010, June 2006. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1629135 [Jan. 27, 2013].

32. D.M. Miller, D. Maslov, G.W. Dueck. "A Transformation Based Algorithm for Reversible Logic Synthesis." *Design Automation Conference Proceedings 2003*, pp. 318-323, June 2-6, 2003. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=1219016&contentType=Conference+Publications&queryText%3DA+Transformation+Based+Algorithm+for+Reversible+Logic+Synthesis [Jan. 27, 2013].

33. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradzev. "On Economical Construction of the Transitive Closure of an Oriented Graph." *Soviet Mathematics Doklady*, 1970, pp. 1209-1210.

34. Martin Albrecht, Gregory Bard, William Hart. "Efficient Multiplication of Dense Matrices over GF(2)." Internet: http://arxiv.org/pdf/0811.1714v1 Nov. 11, 2008. [Jan. 27, 2013].

35. Alexis De Vos. "*Reversible Computing: Fundamentals, Quantum Computing, and Applications.*" Wiley-VCH Verlag GmbH & Co. KGaA, Weinheim, Germany. doi: 10.1002/9783527633999.ch3 Available: http://onlinelibrary.wiley.com/doi/10.1002/9783527633999.ch3/pdf [Jan. 27, 2013].

36. H. T. Kung, W. M. Gentleman. "Matrix triangularization by systolic arrays." Computer Science Department, *Paper* 1603, 1982. Available: http://repository.cmu.edu/compsci/1603 [Jan. 27, 2013].

Appendix

Appendix A: Source.

LNNLinearReversibleSynthesisComparisons.c

```
// Comparisons of LNN Linear Reversible Circuit Synthesis Methods
// Copyright 2011, 2012 Ben Schaeffer
// Permission to copy this file is granted under the terms of the
// GNU Lesser General Public License. See COPYING.LESSER.txt for details.
// Date: October 24, 2012
// Version: 0.4
//
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
//
// Abstract: This program compares nearest neighbor linear reversible circuit
// synthesis methods with long range methods. The long range approaches
// were described in the article "Optimal Synthesis of Linear Reversal
// Circuits" by Patel, Markov, Hayes. The results represent average total
// nearest neighbor CNOT gates per synthesis of heavily randomized circuits.
//
// Can synthesize up to 64 wire circuits.
//
// Gate Encoding is based on the control wire and is described below:
// Format is CNOT(control, target)
// CNOT(0, 1) is encoded as 0, CNOT(1, 2) is encoded as 1, etc.
// CNOT(1, 0) is encoded as -1, CNOT(2, 1) is encoded as -2, etc.
//
// The synthesis output is an array of encoded gates and is
// INVALID-terminated. Applying the output gate sequence to the
// problem (i.e. input) circuit will change the circuit to the
// identity matrix.
//
// To solve the transposed circuit using LNN synthesis methods first
// convert the problem circuit with "TransposeCircuit", call the
// desired synthesis function, and then process the output by using
// "TransposeCNotList". After following these states the output gate
// order will be the same as the non-transposed approach, i.e.
// applying the output gate sequence to the problem (i.e. input)
// circuit will change the circuit to the identity matrix.

#include <stdio.h>
```

```
#include <stdlib.h>
#include <stdint.h>
#define FALSE (0)
#define TRUE (1)
#define TESTS_TO_RUN (100)
//This macro performs a nearest neighbor CNOT operation on circuit "a"
#define ApplyCNOT(a, c) if (c < 0) a[-c-1] ^= a[-c]; else a[c+1] ^= a[c]
//#define ApplyCNot(circuit, gate) if (gate < 0) circuit[-gate - 1] ^= circuit[-gate]; else circuit[gate + 1] ^=
circuit[gate]
#define INVALID (128)    //end of gate sequence marker
#define NSTART (8)
#define NEND (64)
#define NINCREMENT (8)
#define INVALID (128)
#define NMAX (64)
#define CNOTLISTSIZE (4*NMAX*NMAX)

typedef int bool;
static uint64_t identity[NMAX];

void Display(int N, uint64_t * circuit, uint64_t cost, int gates, int depth, int cnot);
void TransposeCircuit(int N, uint64_t * source, uint64_t * destination);
void ReverseandTransposeCNOTList(int * source, int * destination); //Transposes a INVALID-terminated
gate list
void Randomize(int N, uint64_t * circuit); //performs 2*N*N operations on solved matrix
void CopyCircuit(int N, uint64_t * source, uint64_t * destination);
void Initialize(void);
void DisplayAlgorithm1Progress(int N, uint64_t * circuit, int gatecount, int controlrow,
int targetrow);

// The following two functions modify the input circuit. Counts nearest neighbor gates.
int Algorithm_1_by_Patel_Markov_Hayes(int N, uint64_t * circuit, bool displayprogress); //returns gate
count
int Long_Range_Gaussian_CNOT_Synthesis(int N, uint64_t * circuit, bool displayprogress); //returns gate
count

// All subsequent functions make copies of the variable inputcircuit
// Linear Nearest Neighbor Gaussian Elimination using the "upper triangle matrix" approach
int LNNGE_UTM(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Gaussian Elimination using the "lower triangle matrix" approach
int LNNGE_LTM(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Gaussian Elimination with Depth using the "upper triangle matrix" approach
int LNNGED_UTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
// Linear Nearest Neighbor Gaussian Elimination with Depth using the "lower triangle matrix" approach
int LNNGED_LTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
// Linear Nearest Neighbor Alternating Elimination, solve for upper diagonal first
int LNNAE_U(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Alternating Elimination, solve for lower diagonal first
int LNNAE_L(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Alternating Elimination with Depth, solve for upper diagonal first
int LNNAED_U(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
// Linear Nearest Neighbor Alternating Elimination with Depth, solve for lower diagonal first
int LNNAED_L(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
```

```
int main(void)
{
	uint64_t inputcircuit[NMAX], circuit[NMAX], transposedcircuit[NMAX],
	inversecircuit[NMAX], inversetransposedcircuit[NMAX];
	int i, lowestcandidate, temp, cnotlist[CNOTLISTSIZE], gates;
	double algorithm1total;
	double GaussianEliminationtotal;
	double LNNGEtotal;
	double LNNGEBestOf8total;
	double LNNAEtotal;
	double LNNAEBestOf8total;
	double LNNGEDN16BestOf8total[5];
	double LNNAEDN16BestOf8total[5];
	bool displaysynthesis = FALSE;
	int N = NMAX;//Valid between 4 and 64

	Initialize();
	//Output in ".csv" format, compatible with spreadsheet programs
	printf("Comparisons of LNN Linear Reversible Circuit Synthesis"
	"Methods (Average Adjacent CNOT Gate Counts):\n, Gaussian Elimination,"
	" Algorithm 1, LNNGE, LNNAE, LNNGE Best of 8, LNNAED Best of 8\n");
	for (N = NSTART; N <= NEND; N += NINCREMENT){
		algorithm1total = 0;
		GaussianEliminationtotal = 0;
		LNNGEtotal = 0;
		LNNGEBestOf8total = 0;
		LNNAEtotal = 0;
		LNNAEBestOf8total = 0;
		//printf("Random linear reversible circuit synthesis of %d wires\n", N);
		for (i = 0;i < TESTS_TO_RUN; i++){
			Randomize(N, circuit);

			CopyCircuit(N, circuit, inputcircuit);
			gates = Long_Range_Gaussian_CNOT_Synthesis(N, inputcircuit,
displaysynthesis);
			GaussianEliminationtotal += gates;
			//printf("%d,", gates);

			CopyCircuit(N, circuit, inputcircuit);
			gates = Algorithm_1_by_Patel_Markov_Hayes(N, inputcircuit,
displaysynthesis);
			TransposeCircuit(N, inputcircuit, transposedcircuit);
			gates += Algorithm_1_by_Patel_Markov_Hayes(N, transposedcircuit,
displaysynthesis);

			//printf("%d,", gates);
			algorithm1total += gates;

			//Prepare transposed matrix for future function calls
			TransposeCircuit(N, circuit, transposedcircuit);

			//LNNGE best of 8 approaches
			lowestcandidate = LNNGE_UTM(N, circuit, cnotlist);
			LNNGEtotal += lowestcandidate;
			//printf("%d,", lowestcandidate);
```

```
//Use result to compute matrix inverse
CopyCircuit(N, identity, inversecircuit);
for(temp = 0; cnotlist[temp] != INVALID; temp++)
ApplyCNOT(inversecircuit, cnotlist[temp]);
TransposeCircuit(N, inversecircuit, inversetransposedcircuit);

temp = LNNGE_LTM(N, circuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNGE_UTM(N, transposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
temp = LNNGE_LTM(N, transposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNGE_UTM(N, inversecircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNGE_LTM(N, inversecircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNGE_UTM(N, inversetransposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
temp = LNNGE_LTM(N, inversetransposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
LNNGEBestOf8total += lowestcandidate;

//LNNAE best of 8 approaches
lowestcandidate = LNNAE_U(N, circuit, cnotlist);
LNNAEtotal += lowestcandidate;
//printf("%d,", lowestcandidate);
temp = LNNAE_L(N, circuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNAE_U(N, transposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
temp = LNNAE_L(N, transposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNAE_U(N, inversecircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
```

```
temp = LNNAE_L(N, inversecircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
temp = LNNAE_U(N, inversetransposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
temp = LNNAE_L(N, inversetransposedcircuit, cnotlist);
if (lowestcandidate > temp)
lowestcandidate = temp;
//printf("%d,", lowestcandidate);
LNNAEBestOf8total += lowestcandidate;
//printf("%f,", LNNAEBestOf8total);

//Iterative deepening test for LNNGED and LNNAED
if (N == 16)
{
	for (int d = 0; d < 5; d++)
	{
		//LNNGED best of 8 approaches
		lowestcandidate = LNNGED_UTM(N, circuit, cnotlist, d);
		//printf("%d,", lowestcandidate);
		temp = LNNGED_LTM(N, circuit, cnotlist, d);
		if (lowestcandidate > temp)
		lowestcandidate = temp;
		//printf("%d,", lowestcandidate);
		temp = LNNGED_UTM(N, transposedcircuit, cnotlist, d);
		if (lowestcandidate > temp)
		lowestcandidate = temp;
		temp = LNNGED_LTM(N, transposedcircuit, cnotlist, d);
		if (lowestcandidate > temp)
		lowestcandidate = temp;
		//printf("%d,", lowestcandidate);
		temp = LNNGED_UTM(N, inversecircuit, cnotlist, d);
		if (lowestcandidate > temp)
		lowestcandidate = temp;
		//printf("%d,", lowestcandidate);
		temp = LNNGED_LTM(N, inversecircuit, cnotlist, d);
		if (lowestcandidate > temp)
		lowestcandidate = temp;
		//printf("%d,", lowestcandidate);
		temp = LNNGED_UTM(N, inversetransposedcircuit, cnotlist,
d);

		if (lowestcandidate > temp)
		lowestcandidate = temp;
		temp = LNNGED_LTM(N, inversetransposedcircuit, cnotlist,
d);

		if (lowestcandidate > temp)
		lowestcandidate = temp;
		//printf("%d,", lowestcandidate);
		LNNGEDN16BestOf8total[d] += lowestcandidate;

		//LNNAED best of 8 approaches
		lowestcandidate = LNNAED_U(N, circuit, cnotlist, d);
```

```
                                        //printf("%d,", lowestcandidate);
                                        temp = LNNAED_L(N, circuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        //printf("%d,", lowestcandidate);
                                        temp = LNNAED_U(N, transposedcircuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        temp = LNNAED_L(N, transposedcircuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        //printf("%d,", lowestcandidate);
                                        temp = LNNAED_U(N, inversecircuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        //printf("%d,", lowestcandidate);
                                        temp = LNNAED_L(N, inversecircuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        //printf("%d,", lowestcandidate);
                                        temp = LNNAED_U(N, inversetransposedcircuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        temp = LNNAED_L(N, inversetransposedcircuit, cnotlist, d);
                                        if (lowestcandidate > temp)
                                        lowestcandidate = temp;
                                        //printf("%d,", lowestcandidate);
                                        LNNAEDN16BestOf8total[d] += lowestcandidate;
                                }
                        }
                }
                printf("%d, %f, %f, %f, %f, %f, %f\n", N,
                GaussianEliminationtotal/TESTS_TO_RUN,
                algorithm1total/TESTS_TO_RUN,
                LNNGEtotal/TESTS_TO_RUN,
                LNNAEtotal/TESTS_TO_RUN,
                LNNGEBestOf8total/TESTS_TO_RUN,
                LNNAEBestOf8total/TESTS_TO_RUN);
        }
        printf("Iterative deepening comparison of LNNGED and LNNAED for n=16\n");
        printf("Depth, Average CNOT gate count");
        for (int d = 0; d < 5; d++)
        printf("\n%f, %f", LNNGEDN16BestOf8total[d]/TESTS_TO_RUN,
        LNNAEDN16BestOf8total[d]/TESTS_TO_RUN);
        return 0;
}


void TransposeCircuit(int N, uint64_t * source, uint64_t * destination){
        uint64_t destinationflag = 1, sourceflag;
        for (int i=0; i < N; i++)
        destination[i] = 0;
        for (int destinationcolumn = 0; destinationcolumn < N; destinationcolumn++)
        {
```

```
			sourceflag = 1;
			for (int sourcecolumn = 0; sourcecolumn < N; sourcecolumn++)
			{
				if (source[destinationcolumn] & sourceflag)
				destination[sourcecolumn] |= destinationflag;
				sourceflag<<=1;
			}
			destinationflag<<=1;
		}

}

void ReverseandTransposeCNOTList(int * source, int * destination){
		int lower = 0, higher = 0;
		while(source[higher] != INVALID)
		higher++;
		destination[higher] = INVALID;
		while(source[lower] != INVALID)
		destination[--higher] = -(source[lower++]+1);
}

void Initialize(void){
		uint64_t one = 1;
		for (int i = 0; i < NMAX; i++){
			identity[i] = one<<i;
		}
}

void Randomize(int N, uint64_t * circuit){ //performs 2*N ^ 2 operations on solved matrix
		int count = 2*N*N, x1, x2;

		for (int i = 0; i < N; i++)
		circuit[i] = identity[i];
		for (; count > 0; count--)
		{
			x1 = rand()%N;// get a number between 0 and N
			x2 = (x1 + rand()%(N-1) + 1)%N;// get a different number between 0 and N
			if(rand()%2)
			{//randomly use a cnot
				circuit[x1] ^= circuit[x2];
			}
			else
			{//randomly swap wires
				circuit[x1] ^= circuit[x2];
				circuit[x2] ^= circuit[x1];
				circuit[x1] ^= circuit[x2];
			}
		}
}

// "Algorithm 1" by Patel, Markov, Hayes, uses long-range gates
int Algorithm_1_by_Patel_Markov_Hayes(int N, uint64_t * circuit, bool displayprogress) //returns gate
count
{
```

```
uint64_t rowmask, one = 1;//handles section grouping
int m, col, maxcol, row, targetrow, sub_row_pattern;
int count, diagonal_one;
count = 0;
if (displayprogress)
DisplayAlgorithm1Progress(N, circuit, 0, 0, 0);
//first calculate m and row mask
if (N < 32)
{
        m = 2;
        rowmask = 0xFFFFFFFFFFFFFFFC;
}
else
{
        m = 3;
        rowmask = 0xFFFFFFFFFFFFFFF8;
}

col = 0;
while (col < N - 1) {
        maxcol = col + m; //variable to mark width of section
        if (maxcol > N - 1)
        maxcol = N - 1;
        //Step A
        for (row = col; row < N - 1; row++){
                sub_row_pattern = circuit[row] & ~rowmask;
                if (sub_row_pattern){//only search nonzero sub rows
                        for (targetrow = row + 1; targetrow < N; targetrow++){
                                if (sub_row_pattern == (circuit[targetrow] & ~rowmask)){
                                        circuit[targetrow] ^= circuit[row];
                                        if (targetrow == row + 1)
                                        count++;
                                        else
                                        count += ((targetrow - row) << 2) - 4;// cost of
nearest neighbor conversion

                                        if (displayprogress)
                                        DisplayAlgorithm1Progress(N, circuit, count, row,
targetrow);
                                }
                        }
                }
        }
        for (; col < maxcol; col++){//Step B
                if (circuit[col] & one)
                diagonal_one = TRUE;
                else
                diagonal_one = FALSE;
                for (targetrow = col + 1; targetrow < N; targetrow++){
                        if (circuit[targetrow] & one){
                                if (!diagonal_one) {
                                        diagonal_one = TRUE;
                                        circuit[col] ^= circuit[targetrow];
                                        if (targetrow == col + 1)
                                        count++;
```

```
                    else
                    count += ((targetrow - col) << 2) - 4;// cost of nearest
neighbor conversion

                    if (displayprogress)
                    DisplayAlgorithm1Progress(N, circuit, count,
targetrow, col);
                }
                //Step C
                circuit[targetrow] ^= circuit[col];
                if (targetrow == col + 1)
                count++;
                else
                count += ((targetrow - col) << 2) - 4;// cost of nearest
neighbor conversion
                if (displayprogress)
                DisplayAlgorithm1Progress(N, circuit, count, col, targetrow);
            }
        }
        //shift test flag for next iteration
        one <<= 1;
    }
    rowmask <<= m;
  }

  return count;
}

void DisplayAlgorithm1Progress(int N, uint64_t * circuit, int gatecount, int controlrow,
int targetrow){
  uint64_t one = 1;
  int i = 0, j;
  char A='A', chr;

  if (!gatecount)
  printf ("Initial Circuit\n");
  else
  printf ("After CNOT(%d -> %d), Nearest Neighbor Gate Count = %d\n",
  controlrow, targetrow, gatecount);

  for (i = 0; i < N; ++i) {
    for(j=0; j < N; j++){
      if (circuit[i] & (one<<j))
      chr = (char)j + A;
      else
      chr = ' ';
      printf("%c ", chr);//Output appropriate variable
    }
    printf("\n");//end of row
  }
  printf("\n");//end with an extra blank line
}

void CopyCircuit(int N, uint64_t * source, uint64_t * destination){
  for (int i = 0; i < N; i++)
```

```
			destination[i] = source[i];
}

int Long_Range_Gaussian_CNOT_Synthesis(int N, uint64_t * circuit, bool displayprogress){ //returns gate count
			uint64_t one = 1;//handles section grouping
			int col, targetrow;
			int count, diagonal_one;
			count = 0;
			if (displayprogress)
			DisplayAlgorithm1Progress(N, circuit, 0, 0, 0);
			//first calculate m and row mask

			col = 0;
			while (col < N - 1) {
						if (circuit[col] & one)
						diagonal_one = TRUE;
						else
						diagonal_one = FALSE;
						for (targetrow = col + 1; targetrow < N; targetrow++){
									if (circuit[targetrow] & one){
												if (!diagonal_one) {
															diagonal_one = TRUE;
															circuit[col] ^= circuit[targetrow];
															if (targetrow == col + 1)
															count++;
															else
															count += ((targetrow - col) << 2) - 4;// cost of nearest neighbor conversion

															if (displayprogress)
															DisplayAlgorithm1Progress(N, circuit, count, targetrow, col);
												}
												circuit[targetrow] ^= circuit[col];
												if (targetrow == col + 1)
												count++;
												else
												count += ((targetrow - col) << 2) - 4;// cost of nearest neighbor conversion
												if (displayprogress)
												DisplayAlgorithm1Progress(N, circuit, count, col, targetrow);
									}
						}
						col++;
						one <<= 1;
			}

			while (col > 0) {
						for (targetrow = col - 1; targetrow >= 0; targetrow--){
									if (circuit[targetrow] & one){
												circuit[targetrow] ^= circuit[col];
												if (targetrow == col - 1)
												count++;
												else
```

```
					count += ((-targetrow + col) << 2) - 4;// cost of nearest neighbor
conversion
					if (displayprogress)
					DisplayAlgorithm1Progress(N, circuit, count, col, targetrow);
				}
			}
			col--;
			one >>= 1;
		}

		return count;
}

int LNNGE_UTM(int N, uint64_t * inputcircuit, int *cnotlist)
{	//returns gate count
		int totalgates = 0, column = 0, row;
		uint64_t circuit[N], flag;

		for (int i = 0; i < N; i++) //use copy of circuit
		circuit[i] = inputcircuit[i];
		for (; column < N - 1; column++)	//first phase of Gaussian Elimination
		{
			flag = (uint64_t)1 << column;
			for (row = N - 1; row > column; row--)
			if (circuit[row] & flag)
			{
				if (circuit[row - 1] & flag)
				{
					cnotlist[totalgates] = (row - 1);	//CNOT down gate
					totalgates++;
					circuit[row] ^= circuit[row - 1];
				}
				else
				{
					cnotlist[totalgates] = -(row);	//CNOT up gate
					totalgates++;
					cnotlist[totalgates] = (row - 1);	//CNOT down gate
					totalgates++;
					circuit[row - 1] ^= circuit[row];
					circuit[row] ^= circuit[row - 1];
				}
			}
		}

		for (; column > 0; column--)
		{	//second phase of Gaussian Elimination
			flag = (uint64_t)1 << column;
			for (row = 0; row < column && !(circuit[row] & flag); row++)
			;	//search for top instance of variable associated with the column
			if (row != column)
			{	//First extend "1"'s up
				for (int rowhelper = column; rowhelper - 1 > row; rowhelper--)
				{
					if (!(circuit[rowhelper - 1] & flag))
```

```
                        {
                                cnotlist[totalgates] = -(rowhelper);    //CNOT up gate
                                totalgates++;
                                circuit[rowhelper - 1] ^= circuit[rowhelper];
                        }
                }
                for (; ++row <= column;)
                {   //Next eliminate "1"'s
                        cnotlist[totalgates] = -(row);  //CNOT up gate
                        totalgates++;
                        circuit[row - 1] ^= circuit[row];
                }
            }
        }
        cnotlist[totalgates] = INVALID;
        return totalgates;
}

int LNNGE_LTM(int N, uint64_t * inputcircuit, int * cnotlist){//returns gate count
        int totalgates = 0, column, row;
        uint64_t circuit[N], flag;

        for (int i = 0; i < N; i++) //use copy of circuit
        circuit[i] = inputcircuit[i];
        for (column = N-1; column > 0; column--) //first phase of Gaussian Elimination
        {
            flag = (uint64_t)1 << column;
            for (row = 0; row < column; row++)
            if(circuit[row] & flag)
            {
                if(!(circuit[row + 1] & flag))
                {
                        cnotlist[totalgates] = row; //CNOT down gate
                        totalgates++;
                        circuit[row + 1] ^= circuit[row];
                }
                cnotlist[totalgates] = -(row + 1); //CNOT up gate
                totalgates++;
                circuit[row] ^= circuit[row + 1];
                //      Display(circuit, lastpenalty, totalgates, 0, row);
            }
        }
        for (;column < N-1; column++)
        {   //second phase of Gaussian Elimination
            flag = (uint64_t)1 << column;
            for (row = N-1; row > column && !(circuit[row] & flag); row--)
            ;   //search for lowest instance of variable associated with the column
            if (row != column)
            {   //First extend "1"'s down
                for (int rowhelper = column; rowhelper + 1 < row; rowhelper++)
                {
                        if (!(circuit[rowhelper + 1] & flag))
                        {
                                cnotlist[totalgates] = rowhelper; //CNOT down gate
```

```
                    totalgates++;
                    circuit[rowhelper + 1] ^= circuit[rowhelper];
                }
            }
            for (;--row >= column;)
            {   //Next eliminate "1"'s
                cnotlist[totalgates] = row; //CNOT down gate
                totalgates++;
                circuit[row + 1] ^= circuit[row];
            }
        }
    }

    cnotlist[totalgates] = INVALID;
    return totalgates;
}

int LNNGED_UTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth){//returns gate count
    int totalgates = 0, column = 0, row;
    uint64_t circuit[N], flag;
    if (depth == 0)
    return LNNGE_UTM(N, inputcircuit, cnotlist);

    for (int i = 0; i < N; i++) //use copy of circuit
    circuit[i] = inputcircuit[i];
    for (;column < N-1; column++)
    {       //first phase of Gaussian Elimination
        flag = (uint64_t)1 << column;
        for (row = N-1; row > column; row--)
        if(circuit[row] & flag)
        {
            //now check if for another instance of this variable
            //on the row above, necessitating a CNOT down gate
            if(circuit[row - 1] & flag)
            {
                cnotlist[totalgates] = (row - 1); //CNOT down gate
                totalgates++;
                circuit[row] ^= circuit[row - 1];

            }
            else
            {
                //Check for higher instance of variable in the same column
                //i.e. row[0] represents a wire that physically is higher
                //than row[1]
                int rowabove = row - 2, instancefound = FALSE;
                while (!instancefound && rowabove >= column)
                if (circuit[rowabove] & flag)
                instancefound = TRUE;

                else
                rowabove--;
                if (!instancefound)
                {
```

```
                cnotlist[totalgates] = -(row); //CNOT up gate
                totalgates++;
                cnotlist[totalgates] = (row - 1); //CNOT down gate
                totalgates++;
                circuit[row - 1] ^= circuit[row];
                circuit[row] ^= circuit[row - 1];
            }
            else
            {//choose best heuristic and adjust row
                int minimumheuristic, mincnotdown = 0, cnotdown = 0, rown,
cd, temp;

                //first set minimumheuristic to all CNOT up
                for (rown = row; rown - 1> rowabove; rown--)
                circuit[rown - 1] ^= circuit[rown];
                minimumheuristic = LNNGED_UTM(N, circuit,
cnotlist+totalgates, depth - 1);

                //compare against rest
                for (cnotdown = 1; cnotdown < row - rowabove;
cnotdown++)
                {       // compute deltas, find cost, and ultimately restore
                        circuit[rowabove + cnotdown] ^= circuit[rowabove +
cnotdown + 1];
                        circuit[rowabove + cnotdown] ^= circuit[rowabove +
cnotdown - 1];
                        temp = LNNGED_UTM(N, circuit,
cnotlist+totalgates, depth - 1);
                        if (temp < minimumheuristic){
                                minimumheuristic = temp;
                                mincnotdown = cnotdown;
                        }
                }//restore circuit
                for (rown = row; rown - 1 > rowabove; rown--)
                circuit[rown - 1] ^= circuit[rown - 2];

                //choose best
                cnotdown = mincnotdown;
                for (cd = 0; cd < cnotdown; cd++)
                {
                        cnotlist[totalgates] = (rowabove + cd); //CNOT down
gate
                        totalgates++;
                        circuit[rowabove + 1 + cd] ^= circuit[rowabove +
cd];
                }
                for (rown = row; rown - 1> rowabove+cnotdown; rown--)
                {
                        cnotlist[totalgates] = -(rown); //CNOT up gate
                        totalgates++;
                        circuit[rown - 1] ^= circuit[rown];
                }

                row++; //adjustment so row calculation starts over
            }
```

```
					}
				}
			}
			for (; column > 0; column--)
			{	//second phase of Gaussian Elimination
				flag = (uint64_t)1 << column;
				for (row = 0; row < column && !(circuit[row] & flag); row++)
				;	//search for top instance of variable associated with the column
				if (row != column)
				{	//First extend "1"'s up
					for (int rowhelper = column; rowhelper - 1 > row; rowhelper--)
					{
						if (!(circuit[rowhelper - 1] & flag))
						{
							cnotlist[totalgates] = -(rowhelper);	//CNOT up gate
							totalgates++;
							circuit[rowhelper - 1] ^= circuit[rowhelper];
						}
					}
					for (; ++row <= column;)
					{	//Next eliminate "1"'s
						cnotlist[totalgates] = -(row);	//CNOT up gate
						totalgates++;
						circuit[row - 1] ^= circuit[row];
					}
				}
			}
			cnotlist[totalgates] = INVALID;
			return totalgates;
}

int LNNGED_LTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth){//returns gate count
			int totalgates = 0, column, row;
			uint64_t circuit[N], flag;
			if (depth == 0)
			return LNNGE_LTM(N, inputcircuit, cnotlist);

			for (int i = 0; i < N; i++) //use copy of circuit
			circuit[i] = inputcircuit[i];
			for (column = N-1; column > 0; column--)
			{		//first phase of Gaussian Elimination
				flag = (uint64_t)1 << column;
				for (row = 0; row < column; row++)
				if(circuit[row] & flag)
				{
					//now check if for another instance of this variable
					//on the row above, necessitating a CNOT up gate
					if(circuit[row+1] & flag)
					{
						cnotlist[totalgates] = -(row+1); //CNOT up gate
						totalgates++;
						circuit[row] ^= circuit[row+1];

					}
```

```
else
{
	//Check for lower instance of variable in the same column
	int rowbelow = row + 2, instancefound = FALSE;
	while (!instancefound && rowbelow <= column)
	if (circuit[rowbelow] & flag)
	instancefound = TRUE;

	else
	rowbelow++;
	if (!instancefound)
	{
		cnotlist[totalgates] = row; //CNOT down gate
		totalgates++;
		cnotlist[totalgates] = -(row+1); //CNOT up gate
		totalgates++;
		circuit[row+1] ^= circuit[row];
		circuit[row] ^= circuit[row+1];
	}
	else
	{//choose best heuristic and adjust row
		int minimumheuristic, mincnotup = 0, cnotup = 0, rown, cu,
temp;
		//first set minimumheuristic to all CNOT down
		for (rown = row; rown + 1 < rowbelow; rown++)
		circuit[rown + 1] ^= circuit[rown];
		minimumheuristic = LNNGED_LTM(N, circuit,
cnotlist+totalgates, depth - 1);


		//compare against rest
		for (cnotup = 1; cnotup < rowbelow - row; cnotup++)
		{
			// compute deltas, find cost, and eventually restore
circuit
			circuit[rowbelow - cnotup] ^= circuit[rowbelow -
cnotup + 1];
			circuit[rowbelow - cnotup] ^= circuit[rowbelow -
cnotup - 1];
			temp = LNNGED_LTM(N, circuit,
cnotlist+totalgates, depth - 1);
			if (temp < minimumheuristic)
			{
				minimumheuristic = temp;
				mincnotup = cnotup;
			}
		}

		//restore circuit
		for (rown = row; rown + 1 < rowbelow; rown++)
		circuit[rown + 1] ^= circuit[rown + 2];

		//choose best
		cnotup = mincnotup;
		for (cu = 0; cu < cnotup; cu++)
```

```
                                        {
                                                cnotlist[totalgates] = -(rowbelow - cu); //CNOT up
gate
                                                totalgates++;
                                                circuit[rowbelow - 1 - cu] ^= circuit[rowbelow - cu];
                                        }
                                        for (rown = row; rown + 1 < rowbelow - cnotup; rown++)
                                        {
                                                cnotlist[totalgates] = rown; //CNOT down gate
                                                totalgates++;
                                                circuit[rown+1] ^= circuit[rown];
                                        }

                                        row--;//adjustment so row calculation starts over
                                }
                        }
                }
        }
        for (;column < N-1; column++)
        {   //second phase of Gaussian Elimination
                flag = (uint64_t)1 << column;
                for (row = N-1; row > column && !(circuit[row] & flag); row--)
                ;   //search for lowest instance of variable associated with the column
                if (row != column)
                {   //First extend "1"'s down
                        for (int rowhelper = column; rowhelper + 1 < row; rowhelper++)
                        {
                                if (!(circuit[rowhelper + 1] & flag))
                                {
                                        cnotlist[totalgates] = rowhelper; //CNOT down gate
                                        totalgates++;
                                        circuit[rowhelper + 1] ^= circuit[rowhelper];
                                }
                        }
                        for (;--row >= column;)
                        {   //Next eliminate "1"'s
                                cnotlist[totalgates] = row; //CNOT down gate
                                totalgates++;
                                circuit[row + 1] ^= circuit[row];
                        }
                }
        }

        cnotlist[totalgates] = INVALID;
        return totalgates;
}


void Display(int N, uint64_t * circuit, uint64_t cost, int gates, int depth, int cnot) {

        uint64_t one = 1;
        int i=0, j, cnotcontrol, cnottarget;
        char A='A', chr;
        //return;
```

```
		if (cnot != INVALID)
		{
			if (cnot>=0)
			{
				cnotcontrol=cnot;
				cnottarget=cnot+1;
			}
			else
			{
				cnotcontrol=-cnot;
				cnottarget=-cnot-1;
			}
			printf ("After CNOT(%d -> %d): ",cnotcontrol,cnottarget);
		}
		printf("Cost %lld, Total Gates %d, Depth %d\n", cost, gates, depth);
		for (;i<N;++i) {
			for(j=0;j<depth;j++)
			printf("    ");//indentation based on depth
			for(j=0;j<N;j++){
				if (circuit[i]&(one<<j))
				chr = (char)j+A;
				else
				chr = ' ';

				printf("%c ",chr);//Output appropriate variable
			}
			printf("\n");//end of row
		}
		printf("\n");//end with extra blank line
		;
}

// Linear Nearest Neighbor Alternating Elimination, solve for upper diagonal first
int LNNAE_U(int N, uint64_t * inputcircuit, int * cnotlist) //returns gate count
{
		int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
		uint64_t circuit[N], transposedcircuit[N], flag;

		for (int i = 0; i < N; i++) //use copy of circuit
		circuit[i] = inputcircuit[i];
		for (flag = 1, column = 0; column < N - 1; column++, flag <<= 1)	//first phase of Gaussian 
Elimination
		{
			for (row = N - 1; row > column; row--)
			{
				if (circuit[row] & flag)
				{
					if (!(circuit[row - 1] & flag))
					{
						cnotlist[totalgates] = -(row);	//CNOT up gate
						totalgates++;
						circuit[row - 1] ^= circuit[row];
					}
					cnotlist[totalgates] = (row - 1);	//CNOT down gate
```

```
					totalgates++;
					circuit[row] ^= circuit[row - 1];
				}
			}
			//1. Transpose circuit
			//2. Use forward substitution and backwards elimination on column
			//3. Transpose back
			TransposeCircuit(N, circuit, transposedcircuit);
			for (row = N - 1; row > column; row--)
			{
				if (transposedcircuit[row] & flag)
				{
					if (!(transposedcircuit[row - 1] & flag))
					{
						transposedcnotlist[transposedtotalgates] = -(row);  //CNOT up
gate
						transposedtotalgates++;
						transposedcircuit[row - 1] ^= transposedcircuit[row];
					}
					transposedcnotlist[transposedtotalgates] = (row - 1);   //CNOT down
gate
					transposedtotalgates++;
					transposedcircuit[row] ^= transposedcircuit[row - 1];
				}
			}
			TransposeCircuit(N, transposedcircuit, circuit);
		}
		//Terminate both gate lists and combine
		cnotlist[totalgates] = INVALID;
		transposedcnotlist[transposedtotalgates] = INVALID;
		ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);

		return totalgates + transposedtotalgates;
}

// Linear Nearest Neighbor Alternating Elimination, solve for lower diagonal first
int LNNAE_L(int N, uint64_t * inputcircuit, int * cnotlist) //returns gate count
{
		int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
		uint64_t circuit[N], transposedcircuit[N], flag;

		for (int i = 0; i < N; i++) //use copy of circuit
		circuit[i] = inputcircuit[i];
		for (flag = (uint64_t)1 << (N - 1), column = N - 1; column > 0; column--, flag >>= 1)    //first
phase of Gaussian Elimination
		{
			for (row = 0; row < column; row++)
			{
				if(circuit[row] & flag)
				{
					if(!(circuit[row + 1] & flag))
					{
						cnotlist[totalgates] = row; //CNOT down gate
						totalgates++;
```

```
                    circuit[row + 1] ^= circuit[row];
                }
                cnotlist[totalgates] = -(row + 1); //CNOT up gate
                totalgates++;
                circuit[row] ^= circuit[row + 1];
            }
        }
        //1. Transpose circuit
        //2. Use forward substitution and backwards elimination on column
        //3. Transpose back
        TransposeCircuit(N, circuit, transposedcircuit);
        for (row = 0; row < column; row++)
        {
            if(transposedcircuit[row] & flag)
            {
                if(!(transposedcircuit[row + 1] & flag))
                {
                    transposedcnotlist[transposedtotalgates] = row; //CNOT down
gate

                    transposedtotalgates++;
                    transposedcircuit[row + 1] ^= transposedcircuit[row];
                }
                transposedcnotlist[transposedtotalgates] = -(row + 1); //CNOT up gate
                transposedtotalgates++;
                transposedcircuit[row] ^= transposedcircuit[row + 1];
            }
        }
        TransposeCircuit(N, transposedcircuit, circuit);
    }
    //Terminate both gate lists and combine
    cnotlist[totalgates] = INVALID;
    transposedcnotlist[transposedtotalgates] = INVALID;
    ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);
    return totalgates + transposedtotalgates;
}



// Linear Nearest Neighbor Alternating Elimination with Depth, solve for upper diagonal first
int LNNAED_U(int N, uint64_t * inputcircuit, int * cnotlist, int depth) //returns gate count
{
    int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
    uint64_t circuit[N], transposedcircuit[N], flag;
    if (depth == 0)
    return LNNAE_U(N, inputcircuit, cnotlist);

    for (int i = 0; i < N; i++) //use copy of circuit
    circuit[i] = inputcircuit[i];
    for (flag = 1, column = 0; column < N - 1; column++, flag <<= 1)    //first phase of Gaussian
Elimination
    {
        for (row = N-1; row > column; row--)
        if(circuit[row] & flag)
        {
```

```
//now check if for another instance of this variable
//on the row above, necessitating a CNOT down gate
if(circuit[row - 1] & flag)
{
        cnotlist[totalgates] = (row - 1); //CNOT down gate
        totalgates++;
        circuit[row] ^= circuit[row - 1];

}
else
{
        //Check for higher instance of variable in the same column
        //i.e. row[0] represents a wire that physically is higher
        //than row[1]
        int rowabove = row - 2, instancefound = FALSE;
        while (!instancefound && rowabove >= column)
        if (circuit[rowabove] & flag)
        instancefound = TRUE;

        else
        rowabove--;
        if (!instancefound)
        {
                do
                {
                        cnotlist[totalgates] = -(row); //CNOT up gate
                        totalgates++;
                        cnotlist[totalgates] = (row - 1); //CNOT down gate
                        totalgates++;
                        circuit[row - 1] ^= circuit[row];
                        circuit[row] ^= circuit[row - 1];
                }
                while (--row > column);
        }
        else
        {//choose best heuristic and adjust row
                int minimumheuristic, mincnotdown = 0, cnotdown = 0, rown,
cd, temp;
                //first set minimumheuristic to all CNOT up
                for (rown = row; rown - 1> rowabove; rown--)
                circuit[rown - 1] ^= circuit[rown];
                minimumheuristic = LNNAED_U(N, circuit, cnotlist +
totalgates, depth - 1);

                //compare against rest
                for (cnotdown = 1; cnotdown < row - rowabove;
cnotdown++)
                {       // compute deltas, find cost, and ultimately restore
                        circuit[rowabove + cnotdown] ^= circuit[rowabove +
cnotdown + 1];
                        circuit[rowabove + cnotdown] ^= circuit[rowabove +
cnotdown - 1];
                        temp = LNNAED_U(N, circuit, cnotlist + totalgates,
depth - 1);
```

```
                        if (temp < minimumheuristic){
                            minimumheuristic = temp;
                            mincnotdown = cnotdown;
                        }
                    }//restore circuit
                    for (rown = row; rown - 1 > rowabove; rown--)
                    circuit[rown - 1] ^= circuit[rown - 2];

                    //choose best
                    cnotdown = mincnotdown;
                    for (cd = 0; cd < cnotdown; cd++)
                    {
                        cnotlist[totalgates] = (rowabove + cd); //CNOT down
gate

                        totalgates++;
                        circuit[rowabove + 1 + cd] ^= circuit[rowabove +
cd];
                    }
                    for (rown = row; rown - 1> rowabove+cnotdown; rown--)
                    {
                        cnotlist[totalgates] = -(rown); //CNOT up gate
                        totalgates++;
                        circuit[rown - 1] ^= circuit[rown];
                    }

                    row++; //adjustment so row calculation starts over
                }
            }
        }
        //1. Transpose circuit
        //2. Use forward substitution and backwards elimination on column
        //3. Transpose back
        TransposeCircuit(N, circuit, transposedcircuit);

        for (row = N-1; row > column; row--)
        if(transposedcircuit[row] & flag)
        {
            //now check if for another instance of this variable
            //on the row above, necessitating a CNOT down gate
            if(transposedcircuit[row - 1] & flag)
            {
                transposedcnotlist[transposedtotalgates] = (row - 1); //CNOT down
gate

                transposedtotalgates++;
                transposedcircuit[row] ^= transposedcircuit[row - 1];

            }
            else
            {
                //Check for higher instance of variable in the same column
                //i.e. row[0] represents a wire that physically is higher
                //than row[1]
                int rowabove = row - 2, instancefound = FALSE;
                while (!instancefound && rowabove >= column)
```

```
if (transposedcircuit[rowabove] & flag)
instancefound = TRUE;

else
rowabove--;
if (!instancefound)
{
	do
	{
		transposedcnotlist[transposedtotalgates] = -(row);
//CNOT up gate

		transposedtotalgates++;
		transposedcnotlist[transposedtotalgates] = (row - 1);
//CNOT down gate

		transposedtotalgates++;
		transposedcircuit[row - 1] ^= transposedcircuit[row];
		transposedcircuit[row] ^= transposedcircuit[row - 1];
	}
	while (--row > column);
}
else
{//choose best heuristic and adjust row
	int minimumheuristic, mincnotdown = 0, cnotdown = 0, rown,
cd, temp;
	//first set minimumheuristic to all CNOT up
	for (rown = row; rown - 1> rowabove; rown--)
	transposedcircuit[rown - 1] ^= transposedcircuit[rown];
	//In order to keep all operations consistent recursive function
	//calls need to use the non-transposed circuit
	TransposeCircuit(N, transposedcircuit, circuit);
	minimumheuristic = LNNAED_U(N, circuit, cnotlist +
totalgates, depth - 1);

	//compare against rest
	for (cnotdown = 1; cnotdown < row - rowabove;
cnotdown++)
	{		// compute deltas, find cost, and ultimately restore
		transposedcircuit[rowabove + cnotdown] ^=
transposedcircuit[rowabove + cnotdown + 1];
		transposedcircuit[rowabove + cnotdown] ^=
transposedcircuit[rowabove + cnotdown - 1];
		TransposeCircuit(N, transposedcircuit, circuit);
		temp = LNNAED_U(N, circuit, cnotlist + totalgates,
depth - 1);
		if (temp < minimumheuristic){
			minimumheuristic = temp;
			mincnotdown = cnotdown;
		}
	}//restore transposedcircuit
	for (rown = row; rown - 1 > rowabove; rown--)
	transposedcircuit[rown - 1] ^= transposedcircuit[rown - 2];

	//choose best
	cnotdown = mincnotdown;
```

```
                                        for (cd = 0; cd < cnotdown; cd++)
                                        {
                                                transposedcnotlist[transposedtotalgates] = (rowabove
+ cd); //CNOT down gate
                                                transposedtotalgates++;
                                                transposedcircuit[rowabove + 1 + cd] ^=
transposedcircuit[rowabove + cd];
                                        }
                                        for (rown = row; rown - 1> rowabove+cnotdown; rown--)
                                        {
                                                transposedcnotlist[transposedtotalgates] = -(rown);
//CNOT up gate
                                                transposedtotalgates++;
                                                transposedcircuit[rown - 1] ^=
transposedcircuit[rown];
                                        }

                                        row++; //adjustment so row calculation starts over
                                }
                        }
                }
                TransposeCircuit(N, transposedcircuit, circuit);
        }
        //Terminate both gate lists and combine
        cnotlist[totalgates] = INVALID;
        transposedcnotlist[transposedtotalgates] = INVALID;
        ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);

        return totalgates + transposedtotalgates;
}

// Linear Nearest Neighbor Alternating Elimination with Depth, solve for lower diagonal first
int LNNAED_L(int N, uint64_t * inputcircuit, int * cnotlist, int depth) //returns gate count
{
        int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
        uint64_t circuit[N], transposedcircuit[N], flag;

        if (depth == 0)
        return LNNAE_L(N, inputcircuit, cnotlist);
        for (int i = 0; i < N; i++) //use copy of circuit
        circuit[i] = inputcircuit[i];
        for (flag = (uint64_t)1 << (N - 1), column = N - 1; column > 0; column--, flag >>= 1)    //first
phase of Gaussian Elimination
        {
                for (row = 0; row < column; row++)
                if(circuit[row] & flag)
                {
                        //now check if for another instance of this variable
                        //on the row above, necessitating a CNOT up gate
                        if(circuit[row+1] & flag)
                        {
                                cnotlist[totalgates] = -(row+1); //CNOT up gate
                                totalgates++;
```

```
                                circuit[row] ^= circuit[row+1];

                        }
                        else
                        {
                                //Check for lower instance of variable in the same column
                                int rowbelow = row + 2, instancefound = FALSE;
                                while (!instancefound && rowbelow <= column)
                                if (circuit[rowbelow] & flag)
                                instancefound = TRUE;

                                else
                                rowbelow++;
                                if (!instancefound)
                                {
                                        do
                                        {
                                                cnotlist[totalgates] = row; //CNOT down gate
                                                totalgates++;
                                                cnotlist[totalgates] = -(row+1); //CNOT up gate
                                                totalgates++;
                                                circuit[row+1] ^= circuit[row];
                                                circuit[row] ^= circuit[row+1];
                                        }
                                        while (++row < column);
                                }
                                else
                                {//choose best heuristic and adjust row
                                        int minimumheuristic, mincnotup = 0, cnotup = 0, rown, cu,
temp;
                                        //first set minimumheuristic to all CNOT down
                                        for (rown = row; rown + 1 < rowbelow; rown++)
                                        circuit[rown + 1] ^= circuit[rown];
                                        minimumheuristic = LNNAED_L(N, circuit, cnotlist +
totalgates, depth - 1);

                                        //compare against rest
                                        for (cnotup = 1; cnotup < rowbelow - row; cnotup++)
                                        {
                                                // compute deltas, find cost, and eventually restore
circuit
                                                circuit[rowbelow - cnotup] ^= circuit[rowbelow -
cnotup + 1];
                                                circuit[rowbelow - cnotup] ^= circuit[rowbelow -
cnotup - 1];
                                                temp = LNNAED_L(N, circuit, cnotlist + totalgates,
depth - 1);
                                                if (temp < minimumheuristic)
                                                {
                                                        minimumheuristic = temp;
                                                        mincnotup = cnotup;
                                                }
                                        }
```

```
//restore circuit
for (rown = row; rown + 1 < rowbelow; rown++)
circuit[rown + 1] ^= circuit[rown + 2];

//choose best
cnotup = mincnotup;
for (cu = 0; cu < cnotup; cu++)
{
	cnotlist[totalgates] = -(rowbelow - cu); //CNOT up
gate

	totalgates++;
	circuit[rowbelow - 1 - cu] ^= circuit[rowbelow - cu];
}
for (rown = row; rown + 1 < rowbelow - cnotup; rown++)
{
	cnotlist[totalgates] = rown; //CNOT down gate
	totalgates++;
	circuit[rown+1] ^= circuit[rown];
}

row--;//adjustment so row calculation starts over
			}
		}
	}
	//1. Transpose circuit
	//2. Use forward substitution and backwards elimination on column
	//3. Transpose back
	TransposeCircuit(N, circuit, transposedcircuit);
	for (row = 0; row < column; row++)
	if(transposedcircuit[row] & flag)
	{
		//now check if for another instance of this variable
		//on the row above, necessitating a CNOT up gate
		if(transposedcircuit[row+1] & flag)
		{
			transposedcnotlist[transposedtotalgates] = -(row+1); //CNOT up gate
			transposedtotalgates++;
			transposedcircuit[row] ^= transposedcircuit[row+1];

		}
		else
		{
			//Check for lower instance of variable in the same column
			int rowbelow = row + 2, instancefound = FALSE;
			while (!instancefound && rowbelow <= column)
			if (transposedcircuit[rowbelow] & flag)
			instancefound = TRUE;

			else
			rowbelow++;
			if (!instancefound)
			{
				do
				{
```

```
transposedcnotlist[transposedtotalgates] = row;
//CNOT down gate
                                transposedtotalgates++;
                                transposedcnotlist[transposedtotalgates] = -(row+1);
//CNOT up gate
                                transposedtotalgates++;
                                transposedcircuit[row+1] ^= transposedcircuit[row];
                                transposedcircuit[row] ^= transposedcircuit[row+1];
                        }
                        while (++row < column);
                }
                else
                {//choose best heuristic and adjust row
                        int minimumheuristic, mincnotup = 0, cnotup = 0, rown, cu,
temp;
                        //first set minimumheuristic to all CNOT down
                        for (rown = row; rown + 1 < rowbelow; rown++)
                        transposedcircuit[rown + 1] ^= transposedcircuit[rown];
                        TransposeCircuit(N, transposedcircuit, circuit);
                        minimumheuristic = LNNAED_L(N, circuit, cnotlist +
totalgates, depth - 1);

                        //compare against rest
                        for (cnotup = 1; cnotup < rowbelow - row; cnotup++)
                        {
                                // compute deltas, find cost, and eventually restore
transposedcircuit
                                transposedcircuit[rowbelow - cnotup] ^=
transposedcircuit[rowbelow - cnotup + 1];
                                transposedcircuit[rowbelow - cnotup] ^=
transposedcircuit[rowbelow - cnotup - 1];
                                TransposeCircuit(N, transposedcircuit, circuit);
                                temp = LNNAED_L(N, circuit, cnotlist + totalgates,
depth - 1);
                                if (temp < minimumheuristic)
                                {
                                        minimumheuristic = temp;
                                        mincnotup = cnotup;
                                }
                        }

                        //restore transposedcircuit
                        for (rown = row; rown + 1 < rowbelow; rown++)
                        transposedcircuit[rown + 1] ^= transposedcircuit[rown + 2];

                        //choose best
                        cnotup = mincnotup;
                        for (cu = 0; cu < cnotup; cu++)
                        {
                                transposedcnotlist[transposedtotalgates] = -
(rowbelow - cu); //CNOT up gate
                                transposedtotalgates++;
                                transposedcircuit[rowbelow - 1 - cu] ^=
transposedcircuit[rowbelow - cu];
```

```
                                                }
                                                for (rown = row; rown + 1 < rowbelow - cnotup; rown++)
                                                {
                                                        transposedcnotlist[transposedtotalgates] = rown;
//CNOT down gate
                                                        transposedtotalgates++;
                                                        transposedcircuit[rown+1] ^=
transposedcircuit[rown];
                                                }

                                                row--;//adjustment so row calculation starts over
                                        }
                                }
                        }
                        TransposeCircuit(N, transposedcircuit, circuit);
                }
                //Terminate both gate lists and combine
                cnotlist[totalgates] = INVALID;
                transposedcnotlist[transposedtotalgates] = INVALID;
                ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);
                return totalgates + transposedtotalgates;
}
```

LinearReversibleCircuitDatabase5x5.c

```
//LinearReversibleCircuitDatabase5x5.c
//Notes: minimum maximum marker write_count smart

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>

//work on 5x5 proof of concept
#define INCOMPLETE_A (64) //marker to indicate this circuit needs to be searched
#define INCOMPLETE_B (128) //marker to indicate this circuit needs to be searched
#define IDENTITY_CIRCUIT (0xf) //special value for terminal circuit in database searches
#define LOWEST_GATE (-4)
#define HIGHEST_GATE (3)
#define GATE_OFFSET (5) //gates are saved to the database with GATE_OFFSET added
#define NMAX (64)
#define INVALID (128)
#define ApplyCNOT(a, c) if (c < 0) a[-c-1] ^= a[-c]; else a[c+1] ^= a[c]
#define FALSE (0)
#define TRUE (1)
#define CNOTLISTSIZE (4*NMAX*NMAX)

static unsigned int LNNGED4optimaltotal = 0;
static unsigned int LNNAED4optimaltotal = 0;
static unsigned char *buffer;
static unsigned char *counts;
static unsigned control_mask[5] = {0x1f, 0x3e0, 0x7c00, 0xf8000, 0x1f00000};
static uint64_t identity64[64];
static unsigned int verification_counter = 0;
void Initialize64(void);
```

```
bool CircuitsAreEquivalent(int N, uint64_t * circuitA, uint64_t * circuitB);
//Returns a count of equivalent optimal circuits given problem "circuit"
//Returns 0 if circuit is not reversible
unsigned long long CountEquivalentOptimalCircuits5x5(unsigned circuit);

//Do not call this function directly as it is only used as a helper
//function by the public function
unsigned long long PrivateCountEquivalentOptimalCircuits5x5(unsigned circuit);

void _5x5_verify(int i);
void _5x5_comparisons(int i);

void TransposeCircuit(int N, uint64_t * source, uint64_t * destination);
void ReverseandTransposeCNOTList(int * source, int * destination); //Transposes a INVALID-terminated
gate list
// Linear Nearest Neighbor Gaussian Elimination using the "upper triangle matrix" approach
int LNNGE_UTM(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Gaussian Elimination using the "lower triangle matrix" approach
int LNNGE_LTM(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Gaussian Elimination with Depth using the "upper triangle matrix" approach
int LNNGED_UTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
// Linear Nearest Neighbor Gaussian Elimination with Depth using the "lower triangle matrix" approach
int LNNGED_LTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
// Linear Nearest Neighbor Alternating Elimination, solve for upper diagonal first
int LNNAE_U(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Alternating Elimination, solve for lower diagonal first
int LNNAE_L(int N, uint64_t * inputcircuit, int * cnotlist); //returns gate count
// Linear Nearest Neighbor Alternating Elimination with Depth, solve for upper diagonal first
int LNNAED_U(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
// Linear Nearest Neighbor Alternating Elimination with Depth, solve for lower diagonal first
int LNNAED_L(int N, uint64_t * inputcircuit, int * cnotlist, int depth); //returns gate count
void CopyCircuit(int N, uint64_t * source, uint64_t * destination);

int main (void)
{
   unsigned long write_count, minimum_a, maximum_a, minimum_b, maximum_b;
   unsigned long identity, circuit, next_circuit, control;
   int gate; //follows CNOT gate encoding where negative values indicate
   //CNOT up and positive values indicate CNOT down
   int iteration = 1;
   //identity is a function of N, and in this case N=4
   //                                          5=(N+1) 10=2(N+1) 15=3(N+1)
   identity = (1<<0) + (1<<6) + (1<<12) + (1<<18) + (1<<24);
   buffer = malloc(1<<25);
   counts = malloc(1<<25);
   if (!buffer || !counts)
   {
      puts("memory allocation failure");
      return 0;
   }
   //mark the identity matrix so it gets searched
   buffer[identity] = IDENTITY_CIRCUIT | INCOMPLETE_A;
   counts[identity] = 0;
```

```
//Set starting minimum and maximum to index of identity matrix
minimum_a = identity;
maximum_a = identity;

do
{
   write_count = 0;
   minimum_b = identity;//Reset minimum and maximum for recalculation
   maximum_b = identity;
   for (circuit = minimum_a; circuit <= maximum_a; circuit++)
   {
      if (buffer[circuit] & INCOMPLETE_A)//if circuit has not been searched
      {
         buffer[circuit] ^= INCOMPLETE_A;//clear incomplete flag
         //Search all nearby circuits except predecessor circuit
         for (gate = LOWEST_GATE; gate <= HIGHEST_GATE; gate++)
            if (gate != (int)buffer[circuit] - GATE_OFFSET)
            {
               if (gate >= 0)
               {
                  control = circuit & control_mask[gate];
                  next_circuit = circuit ^ control << 5;
               }
               else
               {
                  control = circuit & control_mask[-gate];
                  next_circuit = circuit ^ control >> 5;
               }
               if (buffer[next_circuit] == 0)//if circuit is unknown
               {
                  //Now that it is known that this circuit is reversible and needs to be marked as incomplete
                  buffer[next_circuit] = (gate + GATE_OFFSET) | INCOMPLETE_B;
                  counts[next_circuit] = iteration;
                  write_count++;
                  //adjust minimum and maximum if necessary
                  if (minimum_b > next_circuit)
                     minimum_b = next_circuit;
                  else if (maximum_b < next_circuit)
                     maximum_b = next_circuit;
               }
            }
      }
   }
   if (write_count == 0)//Break if search is complete
      break;
   printf ("iteration = %2d, write_count = %ld\n", iteration++, write_count);
   write_count=0;
   //Second iteration
   minimum_a = identity;//Reset minimum and maximum for recalculation
   maximum_a = identity;
   for (circuit = minimum_b; circuit <= maximum_b; circuit++)
   {
      if (buffer[circuit] & INCOMPLETE_B)//if circuit has not been searched
      {
```

```
            buffer[circuit] ^= INCOMPLETE_B;//clear incomplete flag
            //Search all nearby circuits except predecessor circuit
            for (gate = LOWEST_GATE; gate <= HIGHEST_GATE; gate++)
               if (gate != (int)buffer[circuit] - GATE_OFFSET)
               {
                  if (gate >= 0)
                  {
                     control = circuit & control_mask[gate];
                     next_circuit = circuit ^ control << 5;
                  }
                  else
                  {
                     control = circuit & control_mask[-gate];
                     next_circuit = circuit ^ control >> 5;
                  }
                  if (buffer[next_circuit] == 0)//if circuit is unknown
                  {
                     //Now that it is known that this circuit is reversible and needs to be marked as incomplete
                     buffer[next_circuit] = (gate + GATE_OFFSET) | INCOMPLETE_A;
                     write_count++;
                     counts[next_circuit] = iteration;
                     //adjust minimum and maximum if necessary
                     if (minimum_a > next_circuit)
                        minimum_a = next_circuit;
                     else if (maximum_a < next_circuit)
                        maximum_a = next_circuit;
                  }
               }
         }
      }
      printf ("iteration = %2d, write_count = %ld\n", iteration++, write_count);
   }
   while (write_count > 0);

   for (int i = 0; i< 1<<25; i++)
      if (buffer[i])
         _5x5_verify(i);
   if (verification_counter == 9999360)//expected value from equation
      //(2^5-1)*(2^5-2)*(2^5-4)*(2^5-8)*(2^5-16)
   {
      puts("Database Verified.");

      //FILE * f = fopen("5x5LNN_LRC.dat","wb");
      //fwrite(buffer, 1<<25, 1, f);
      //fclose(f);
      int temp = 0;
      for (int i = 0; i< 1<<25; i++)
      {
         if (buffer[i])
         {
            _5x5_comparisons(i);
         }
      }
      printf("LNNGED depth = 4 optimal total: %lld\n", LNNGED4optimaltotal);
```

```
      printf("LNNAED depth = 4 optimal total: %lld\n", LNNAED4optimaltotal);
      printf("Optimal total 9999360\n");
   }
   else
      printf("%lld errors in database detected", 9999360 -
            verification_counter);
   return 0;
}

unsigned long long CountEquivalentOptimalCircuits5x5(unsigned circuit)
//Returns a count of equivalent optimal circuits given problem "circuit"
//Returns 0 if circuit is not reversible
{
   if (buffer[circuit] == 0)
      return 0;
   return PrivateCountEquivalentOptimalCircuits5x5(circuit);
}

unsigned long long PrivateCountEquivalentOptimalCircuits5x5(unsigned circuit)
{
   unsigned long long count = 0;
   unsigned control, next_circuit;
   int gate;
   if (buffer[circuit] == IDENTITY_CIRCUIT)
      return 1;
   //Recursively add up all counts of equivalent circuits
   for (gate = LOWEST_GATE; gate <= HIGHEST_GATE; gate++)
   {
      if (gate >= 0)
      {
         control = circuit & control_mask[gate];
         next_circuit = circuit ^ control << 5;
      }
      else
      {
         control = circuit & control_mask[-gate];
         next_circuit = circuit ^ control >> 5;
      }
      if (counts[circuit] - 1 == counts[next_circuit])
         count += PrivateCountEquivalentOptimalCircuits5x5(next_circuit);
   }
   return count;
}

void Initialize64(void) {
   uint64_t one = 1;
   for (int i = 0; i < NMAX; i++) {
      identity64[i] = one << i;
   }
}

int VerifyCNOTList(int N, uint64_t * inputcircuit, int * cnotlist) //true return means verified, false fails
{
   int i;
```

```
    uint64_t circuit[NMAX];
    Initialize64();
    for (i = 0; i < N; i++) //use copy of circuit
        circuit[i] = identity64[i];
    for (i = 0; cnotlist[i] != INVALID; i++)
        ApplyCNOT(circuit, cnotlist[i]);
    return CircuitsAreEquivalent(N, circuit, inputcircuit);
}

bool CircuitsAreEquivalent(int N, uint64_t * circuitA, uint64_t * circuitB)
{
    for (int i = 0; i < N; i++)
        if (circuitA[i] != circuitB[i])
            return false;
    return true;
}

void _5x5_verify(int i)
{
    int caution = 25;//maximum optimal CNOT list length
    while(buffer[i] != IDENTITY_CIRCUIT)
    {
        if (!buffer[i])//not reversible
        {
            printf("Encountered database error... exiting");
            exit(1);
        }
        if (caution-- == 0)//check for CNOT list getting longer than maximum for 5x5
            return;//this circuit will not be counted but subsequent tests can continue
        if (buffer[i] - GATE_OFFSET>= 0)
        {
            i ^= (i & control_mask[buffer[i] - GATE_OFFSET]) << 5;
        }
        else
        {
            i ^= (i & control_mask[-(buffer[i] - GATE_OFFSET)]) >> 5;
        }
    }
    verification_counter++;
}

void _5x5_comparisons(int i)
{
    int optimalcount = 0;//maximum optimal CNOT list length
    int LNNGEDcount = 0;
    int LNNAEDcount = 0;
    uint64_t circuit[5], transposedcircuit[5], inversetransposedcircuit[5];
    uint64_t inversecircuit[5] = {1, 2, 4, 8, 16};
    int cnotlist [100];
    int temp;

    circuit[0] = i & 0x1f;
    circuit[1] = (i >> 5) & 0x1f;
    circuit[2] = (i >> 10) & 0x1f;
```

```
circuit[3] = (i >> 15) & 0x1f;
circuit[4] = (i >> 20) & 0x1f;

while(buffer[i] != IDENTITY_CIRCUIT)
{
   optimalcount++;
   if (buffer[i] - GATE_OFFSET>= 0)
   {
      i ^= (i & control_mask[buffer[i] - GATE_OFFSET]) << 5;
   }
   else
   {
      i ^= (i & control_mask[-(buffer[i] - GATE_OFFSET)]) >> 5;
   }
}

//Prepare transposed matrix for future function calls
TransposeCircuit(5, circuit, transposedcircuit);
LNNGE_UTM(5, circuit, cnotlist);
for(temp = 0; cnotlist[temp] != INVALID; temp++)
   ApplyCNOT(inversecircuit, cnotlist[temp]);
TransposeCircuit(5, inversecircuit, inversetransposedcircuit);

//LNNGED best of 8 approaches
LNNGEDcount = LNNGED_UTM(5,circuit, cnotlist, 4);
temp = LNNGED_LTM(5,circuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
temp = LNNGED_UTM(5,transposedcircuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
temp = LNNGED_LTM(5,transposedcircuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
temp = LNNGED_UTM(5,inversecircuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
temp = LNNGED_LTM(5,inversecircuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
temp = LNNGED_UTM(5,inversetransposedcircuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
temp = LNNGED_LTM(5,inversetransposedcircuit, cnotlist, 4);
if (LNNGEDcount > temp)
   LNNGEDcount = temp;
if (LNNGEDcount == optimalcount)
   LNNGED4optimaltotal++;

//LNNAED best of 8 approaches
LNNAEDcount = LNNAED_U(5,circuit, cnotlist, 4);
//printf("%d,", LNNAEDcount);
temp = LNNAED_L(5,circuit, cnotlist, 4);
if (LNNAEDcount > temp)
```

```
        LNNAEDcount = temp;
    //printf("%d,", LNNAEDcount);
    temp = LNNAED_U(5,transposedcircuit, cnotlist, 4);
    if (LNNAEDcount > temp)
        LNNAEDcount = temp;
    temp = LNNAED_L(5,transposedcircuit, cnotlist, 4);
    if (LNNAEDcount > temp)
        LNNAEDcount = temp;
    //printf("%d,", LNNAEDcount);
    temp = LNNAED_U(5,inversecircuit, cnotlist, 4);
    if (LNNAEDcount > temp)
        LNNAEDcount = temp;
    //printf("%d,", LNNAEDcount);
    temp = LNNAED_L(5,inversecircuit, cnotlist, 4);
    if (LNNAEDcount > temp)
        LNNAEDcount = temp;
    //printf("%d,", LNNAEDcount);
    temp = LNNAED_U(5,inversetransposedcircuit, cnotlist, 4);
    if (LNNAEDcount > temp)
        LNNAEDcount = temp;
    temp = LNNAED_L(5,inversetransposedcircuit, cnotlist, 4);
    if (LNNAEDcount > temp)
        LNNAEDcount = temp;
    if (LNNAEDcount == optimalcount)
        LNNAED4optimaltotal++;
}

void TransposeCircuit(int N, uint64_t * source, uint64_t * destination) {
    uint64_t destinationflag = 1, sourceflag;
    for (int i=0; i < N; i++)
        destination[i] = 0;
    for (int destinationcolumn = 0; destinationcolumn < N; destinationcolumn++)
    {
        sourceflag = 1;
        for (int sourcecolumn = 0; sourcecolumn < N; sourcecolumn++)
        {
            if (source[destinationcolumn] & sourceflag)
                destination[sourcecolumn] |= destinationflag;
            sourceflag<<=1;
        }
        destinationflag<<=1;
    }
}

void ReverseandTransposeCNOTList(int * source, int * destination) {
    int lower = 0, higher = 0;
    while(source[higher] != INVALID)
        higher++;
    destination[higher] = INVALID;
    while(source[lower] != INVALID)
        destination[--higher] = -(source[lower++]+1);
}

int LNNGE_UTM(int N, uint64_t * inputcircuit, int *cnotlist)
```

```
{   //returns gate count
    int totalgates = 0, column = 0, row;
    uint64_t circuit[N], flag;

    for (int i = 0; i < N; i++) //use copy of circuit
       circuit[i] = inputcircuit[i];
    for (; column < N - 1; column++)    //first phase of Gaussian Elimination
    {
       flag = (uint64_t)1 << column;
       for (row = N - 1; row > column; row--)
          if (circuit[row] & flag)
          {
             if (circuit[row - 1] & flag)
             {
                cnotlist[totalgates] = (row - 1);   //CNOT down gate
                totalgates++;
                circuit[row] ^= circuit[row - 1];
             }
             else
             {
                cnotlist[totalgates] = -(row);  //CNOT up gate
                totalgates++;
                cnotlist[totalgates] = (row - 1);   //CNOT down gate
                totalgates++;
                circuit[row - 1] ^= circuit[row];
                circuit[row] ^= circuit[row - 1];
             }
          }
    }

    for (; column > 0; column--)
    {   //second phase of Gaussian Elimination
       flag = (uint64_t)1 << column;
       for (row = 0; row < column && !(circuit[row] & flag); row++)
          ;  //search for top instance of variable associated with the column
       if (row != column)
       {  //First extend "1"'s up
          for (int rowhelper = column; rowhelper - 1 > row; rowhelper--)
          {
             if (!(circuit[rowhelper - 1] & flag))
             {
                cnotlist[totalgates] = -(rowhelper);    //CNOT up gate
                totalgates++;
                circuit[rowhelper - 1] ^= circuit[rowhelper];
             }
          }
          for (; ++row <= column;)
          {  //Next eliminate "1"'s
             cnotlist[totalgates] = -(row);  //CNOT up gate
             totalgates++;
             circuit[row - 1] ^= circuit[row];
          }
       }
    }
```

```
    cnotlist[totalgates] = INVALID;
    return totalgates;
}

int LNNGE_LTM(int N, uint64_t * inputcircuit, int * cnotlist) { //returns gate count
    int totalgates = 0, column, row;
    uint64_t circuit[N], flag;

    for (int i = 0; i < N; i++) //use copy of circuit
        circuit[i] = inputcircuit[i];
    for (column = N-1; column > 0; column--) //first phase of Gaussian Elimination
    {
        flag = (uint64_t)1 << column;
        for (row = 0; row < column; row++)
            if(circuit[row] & flag)
            {
                if(!(circuit[row + 1] & flag))
                {
                    cnotlist[totalgates] = row; //CNOT down gate
                    totalgates++;
                    circuit[row + 1] ^= circuit[row];
                }
                cnotlist[totalgates] = -(row + 1); //CNOT up gate
                totalgates++;
                circuit[row] ^= circuit[row + 1];
                //          Display(circuit, lastpenalty, totalgates, 0, row);
            }
    }
    for (; column < N-1; column++)
    {   //second phase of Gaussian Elimination
        flag = (uint64_t)1 << column;
        for (row = N-1; row > column && !(circuit[row] & flag); row--)
            ;   //search for lowest instance of variable associated with the column
        if (row != column)
        {   //First extend "1"'s down
            for (int rowhelper = column; rowhelper + 1 < row; rowhelper++)
            {
                if (!(circuit[rowhelper + 1] & flag))
                {
                    cnotlist[totalgates] = rowhelper; //CNOT down gate
                    totalgates++;
                    circuit[rowhelper + 1] ^= circuit[rowhelper];
                }
            }
            for (; --row >= column;)
            {   //Next eliminate "1"'s
                cnotlist[totalgates] = row; //CNOT down gate
                totalgates++;
                circuit[row + 1] ^= circuit[row];
            }
        }
    }

    cnotlist[totalgates] = INVALID;
```

```
   return totalgates;
}

int LNNGED_UTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth) { //returns gate count
   int totalgates = 0, column = 0, row;
   uint64_t circuit[N], flag;
   if (depth == 0)
      return LNNGE_UTM(N, inputcircuit, cnotlist);

   for (int i = 0; i < N; i++) //use copy of circuit
      circuit[i] = inputcircuit[i];
   for (; column < N-1; column++)
   {  //first phase of Gaussian Elimination
      flag = (uint64_t)1 << column;
      for (row = N-1; row > column; row--)
         if(circuit[row] & flag)
         {
            //now check if for another instance of this variable
            //on the row above, necessitating a CNOT down gate
            if(circuit[row - 1] & flag)
            {
               cnotlist[totalgates] = (row - 1); //CNOT down gate
               totalgates++;
               circuit[row] ^= circuit[row - 1];
            }
            else
            {
               //Check for higher instance of variable in the same column
               //i.e. row[0] represents a wire that physically is higher
               //than row[1]
               int rowabove = row - 2, instancefound = FALSE;
               while (!instancefound && rowabove >= column)
                  if (circuit[rowabove] & flag)
                     instancefound = TRUE;
                  else
                     rowabove--;
               if (!instancefound)
               {
                  cnotlist[totalgates] = -(row); //CNOT up gate
                  totalgates++;
                  cnotlist[totalgates] = (row - 1); //CNOT down gate
                  totalgates++;
                  circuit[row - 1] ^= circuit[row];
                  circuit[row] ^= circuit[row - 1];
               }
               else
               {  //choose best heuristic and adjust row
                  int minimumheuristic, mincnotdown = 0, cnotdown = 0, rown, cd, temp;
                  //first set minimumheuristic to all CNOT up
                  for (rown = row; rown - 1> rowabove; rown--)
                     circuit[rown - 1] ^= circuit[rown];
                  minimumheuristic = LNNGED_UTM(N, circuit, cnotlist+totalgates, depth - 1);

                  //compare against rest
```

```
                for (cnotdown = 1; cnotdown < row - rowabove; cnotdown++)
                {   // compute deltas, find cost, and ultimately restore
                    circuit[rowabove + cnotdown] ^= circuit[rowabove + cnotdown + 1];
                    circuit[rowabove + cnotdown] ^= circuit[rowabove + cnotdown - 1];
                    temp = LNNGED_UTM(N, circuit, cnotlist+totalgates, depth - 1);
                    if (temp < minimumheuristic) {
                        minimumheuristic = temp;
                        mincnotdown = cnotdown;
                    }
                }//restore circuit
                for (rown = row; rown - 1 > rowabove; rown--)
                    circuit[rown - 1] ^= circuit[rown - 2];

                //choose best
                cnotdown = mincnotdown;
                for (cd = 0; cd < cnotdown; cd++)
                {
                    cnotlist[totalgates] = (rowabove + cd); //CNOT down gate
                    totalgates++;
                    circuit[rowabove + 1 + cd] ^= circuit[rowabove + cd];
                }
                for (rown = row; rown - 1> rowabove+cnotdown; rown--)
                {
                    cnotlist[totalgates] = -(rown); //CNOT up gate
                    totalgates++;
                    circuit[rown - 1] ^= circuit[rown];
                }

                row++; //adjustment so row calculation starts over
            }
        }
    }
}
for (; column > 0; column--)
{   //second phase of Gaussian Elimination
    flag = (uint64_t)1 << column;
    for (row = 0; row < column && !(circuit[row] & flag); row++)
        ;   //search for top instance of variable associated with the column
    if (row != column)
    {   //First extend "1"'s up
        for (int rowhelper = column; rowhelper - 1 > row; rowhelper--)
        {
            if (!(circuit[rowhelper - 1] & flag))
            {
                cnotlist[totalgates] = -(rowhelper);    //CNOT up gate
                totalgates++;
                circuit[rowhelper - 1] ^= circuit[rowhelper];
            }
        }
        for (; ++row <= column;)
        {   //Next eliminate "1"'s
            cnotlist[totalgates] = -(row);  //CNOT up gate
            totalgates++;
            circuit[row - 1] ^= circuit[row];
```

```
            }
        }
    }
    cnotlist[totalgates] = INVALID;
    return totalgates;
}

int LNNGED_LTM(int N, uint64_t * inputcircuit, int * cnotlist, int depth) { //returns gate count
    int totalgates = 0, column, row;
    uint64_t circuit[N], flag;
    if (depth == 0)
        return LNNGE_LTM(N, inputcircuit, cnotlist);

    for (int i = 0; i < N; i++) //use copy of circuit
        circuit[i] = inputcircuit[i];
    for (column = N-1; column > 0; column--)
    {   //first phase of Gaussian Elimination
        flag = (uint64_t)1 << column;
        for (row = 0; row < column; row++)
            if(circuit[row] & flag)
            {
                //now check if for another instance of this variable
                //on the row above, necessitating a CNOT up gate
                if(circuit[row+1] & flag)
                {
                    cnotlist[totalgates] = -(row+1); //CNOT up gate
                    totalgates++;
                    circuit[row] ^= circuit[row+1];
                }
                else
                {
                    //Check for lower instance of variable in the same column
                    int rowbelow = row + 2, instancefound = FALSE;
                    while (!instancefound && rowbelow <= column)
                        if (circuit[rowbelow] & flag)
                            instancefound = TRUE;
                        else
                            rowbelow++;
                    if (!instancefound)
                    {
                        cnotlist[totalgates] = row; //CNOT down gate
                        totalgates++;
                        cnotlist[totalgates] = -(row+1); //CNOT up gate
                        totalgates++;
                        circuit[row+1] ^= circuit[row];
                        circuit[row] ^= circuit[row+1];
                    }
                    else
                    {   //choose best heuristic and adjust row
                        int minimumheuristic, mincnotup = 0, cnotup = 0, rown, cu, temp;
                        //first set minimumheuristic to all CNOT down
                        for (rown = row; rown + 1 < rowbelow; rown++)
                            circuit[rown + 1] ^= circuit[rown];
                        minimumheuristic = LNNGED_LTM(N, circuit, cnotlist+totalgates, depth - 1);
```

```
          //compare against rest
          for (cnotup = 1; cnotup < rowbelow - row; cnotup++)
          {
            // compute deltas, find cost, and eventually restore circuit
            circuit[rowbelow - cnotup] ^= circuit[rowbelow - cnotup + 1];
            circuit[rowbelow - cnotup] ^= circuit[rowbelow - cnotup - 1];
            temp = LNNGED_LTM(N, circuit, cnotlist+totalgates, depth - 1);
            if (temp < minimumheuristic)
            {
              minimumheuristic = temp;
              mincnotup = cnotup;
            }
          }

          //restore circuit
          for (rown = row; rown + 1 < rowbelow; rown++)
            circuit[rown + 1] ^= circuit[rown + 2];

          //choose best
          cnotup = mincnotup;
          for (cu = 0; cu < cnotup; cu++)
          {
            cnotlist[totalgates] = -(rowbelow - cu); //CNOT up gate
            totalgates++;
            circuit[rowbelow - 1 - cu] ^= circuit[rowbelow - cu];
          }
          for (rown = row; rown + 1 < rowbelow - cnotup; rown++)
          {
            cnotlist[totalgates] = rown; //CNOT down gate
            totalgates++;
            circuit[rown+1] ^= circuit[rown];
          }

          row--;//adjustment so row calculation starts over
        }
      }
    }
}
for (; column < N-1; column++)
{   //second phase of Gaussian Elimination
  flag = (uint64_t)1 << column;
  for (row = N-1; row > column && !(circuit[row] & flag); row--)
    ;  //search for lowest instance of variable associated with the column
  if (row != column)
  {  //First extend "1"'s down
    for (int rowhelper = column; rowhelper + 1 < row; rowhelper++)
    {
      if (!(circuit[rowhelper + 1] & flag))
      {
        cnotlist[totalgates] = rowhelper; //CNOT down gate
        totalgates++;
        circuit[rowhelper + 1] ^= circuit[rowhelper];
      }
```

```
            }
            for (; --row >= column;)
            {   //Next eliminate "1"'s
                cnotlist[totalgates] = row; //CNOT down gate
                totalgates++;
                circuit[row + 1] ^= circuit[row];
            }
        }
    }

    cnotlist[totalgates] = INVALID;
    return totalgates;
}


void Display(int N, uint64_t * circuit, uint64_t cost, int gates, int depth, int cnot) {

    uint64_t one = 1;
    int i=0, j, cnotcontrol, cnottarget;
    char A='A', chr;
    //return;
    if (cnot != INVALID)
    {
        if (cnot>=0)
        {
            cnotcontrol=cnot;
            cnottarget=cnot+1;
        }
        else
        {
            cnotcontrol=-cnot;
            cnottarget=-cnot-1;
        }
        printf ("After CNOT(%d -> %d): ",cnotcontrol,cnottarget);
    }
    printf("Cost %lld, Total Gates %d, Depth %d\n", cost, gates, depth);
    for (; i<N; ++i) {
        for(j=0; j<depth; j++)
            printf("    ");//indentation based on depth
        for(j=0; j<N; j++) {
            if (circuit[i]&(one<<j))
                chr = (char)j+A;
            else
                chr = ' ';

            printf("%c ",chr);//Output appropriate variable
        }
        printf("\n");//end of row
    }
    printf("\n");//end with extra blank line
    ;
}

// Linear Nearest Neighbor Alternating Elimination, solve for upper diagonal first
```

```
int LNNAE_U(int N, uint64_t * inputcircuit, int * cnotlist) //returns gate count
{
   int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
   uint64_t circuit[N], transposedcircuit[N], flag;

   for (int i = 0; i < N; i++) //use copy of circuit
      circuit[i] = inputcircuit[i];
   for (flag = 1, column = 0; column < N - 1; column++, flag <<= 1)    //first phase of Gaussian
Elimination
   {
      for (row = N - 1; row > column; row--)
      {
         if (circuit[row] & flag)
         {
            if (!(circuit[row - 1] & flag))
            {
               cnotlist[totalgates] = -(row);  //CNOT up gate
               totalgates++;
               circuit[row - 1] ^= circuit[row];
            }
            cnotlist[totalgates] = (row - 1);   //CNOT down gate
            totalgates++;
            circuit[row] ^= circuit[row - 1];
         }
      }
      //1. Transpose circuit
      //2. Use forward substitution and backwards elimination on column
      //3. Transpose back
      TransposeCircuit(N, circuit, transposedcircuit);
      for (row = N - 1; row > column; row--)
      {
         if (transposedcircuit[row] & flag)
         {
            if (!(transposedcircuit[row - 1] & flag))
            {
               transposedcnotlist[transposedtotalgates] = -(row);  //CNOT up gate
               transposedtotalgates++;
               transposedcircuit[row - 1] ^= transposedcircuit[row];
            }
            transposedcnotlist[transposedtotalgates] = (row - 1);   //CNOT down gate
            transposedtotalgates++;
            transposedcircuit[row] ^= transposedcircuit[row - 1];
         }
      }
      TransposeCircuit(N, transposedcircuit, circuit);
   }
   //Terminate both gate lists and combine
   cnotlist[totalgates] = INVALID;
   transposedcnotlist[transposedtotalgates] = INVALID;
   ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);

   return totalgates + transposedtotalgates;
}
```

```
// Linear Nearest Neighbor Alternating Elimination, solve for lower diagonal first
int LNNAE_L(int N, uint64_t * inputcircuit, int * cnotlist) //returns gate count
{
   int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
   uint64_t circuit[N], transposedcircuit[N], flag;

   for (int i = 0; i < N; i++) //use copy of circuit
      circuit[i] = inputcircuit[i];
   for (flag = (uint64_t)1 << (N - 1), column = N - 1; column > 0; column--, flag >>= 1)    //first phase of
Gaussian Elimination
   {
      for (row = 0; row < column; row++)
      {
         if(circuit[row] & flag)
         {
            if(!(circuit[row + 1] & flag))
            {
               cnotlist[totalgates] = row; //CNOT down gate
               totalgates++;
               circuit[row + 1] ^= circuit[row];
            }
            cnotlist[totalgates] = -(row + 1); //CNOT up gate
            totalgates++;
            circuit[row] ^= circuit[row + 1];
         }
      }
      //1. Transpose circuit
      //2. Use forward substitution and backwards elimination on column
      //3. Transpose back
      TransposeCircuit(N, circuit, transposedcircuit);
      for (row = 0; row < column; row++)
      {
         if(transposedcircuit[row] & flag)
         {
            if(!(transposedcircuit[row + 1] & flag))
            {
               transposedcnotlist[transposedtotalgates] = row; //CNOT down gate
               transposedtotalgates++;
               transposedcircuit[row + 1] ^= transposedcircuit[row];
            }
            transposedcnotlist[transposedtotalgates] = -(row + 1); //CNOT up gate
            transposedtotalgates++;
            transposedcircuit[row] ^= transposedcircuit[row + 1];
         }
      }
      TransposeCircuit(N, transposedcircuit, circuit);
   }
   //Terminate both gate lists and combine
   cnotlist[totalgates] = INVALID;
   transposedcnotlist[transposedtotalgates] = INVALID;
   ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);
   return totalgates + transposedtotalgates;
}
```

```
// Linear Nearest Neighbor Alternating Elimination with Depth, solve for upper diagonal first
int LNNAED_U(int N, uint64_t * inputcircuit, int * cnotlist, int depth) //returns gate count
{
   int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
   uint64_t circuit[N], transposedcircuit[N], flag;
   if (depth == 0)
      return LNNAE_U(N, inputcircuit, cnotlist);

   for (int i = 0; i < N; i++) //use copy of circuit
      circuit[i] = inputcircuit[i];
   for (flag = 1, column = 0; column < N - 1; column++, flag <<= 1)    //first phase of Gaussian
Elimination
   {
      for (row = N-1; row > column; row--)
         if(circuit[row] & flag)
         {
            //now check if for another instance of this variable
            //on the row above, necessitating a CNOT down gate
            if(circuit[row - 1] & flag)
            {
               cnotlist[totalgates] = (row - 1); //CNOT down gate
               totalgates++;
               circuit[row] ^= circuit[row - 1];
            }
            else
            {
               //Check for higher instance of variable in the same column
               //i.e. row[0] represents a wire that physically is higher
               //than row[1]
               int rowabove = row - 2, instancefound = FALSE;
               while (!instancefound && rowabove >= column)
                  if (circuit[rowabove] & flag)
                     instancefound = TRUE;
                  else
                     rowabove--;
               if (!instancefound)
               {
                  do
                  {
                     cnotlist[totalgates] = -(row); //CNOT up gate
                     totalgates++;
                     cnotlist[totalgates] = (row - 1); //CNOT down gate
                     totalgates++;
                     circuit[row - 1] ^= circuit[row];
                     circuit[row] ^= circuit[row - 1];
                  }
                  while (--row > column);
               }
               else
               {  //choose best heuristic and adjust row
                  int minimumheuristic, mincnotdown = 0, cnotdown = 0, rown, cd, temp;
                  //first set minimumheuristic to all CNOT up
```

```
            for (rown = row; rown - 1> rowabove; rown--)
               circuit[rown - 1] ^= circuit[rown];
            minimumheuristic = LNNAED_U(N, circuit, cnotlist + totalgates, depth - 1);

            //compare against rest
            for (cnotdown = 1; cnotdown < row - rowabove; cnotdown++)
            {  // compute deltas, find cost, and ultimately restore
               circuit[rowabove + cnotdown] ^= circuit[rowabove + cnotdown + 1];
               circuit[rowabove + cnotdown] ^= circuit[rowabove + cnotdown - 1];
               temp = LNNAED_U(N, circuit, cnotlist + totalgates, depth - 1);
               if (temp < minimumheuristic) {
                  minimumheuristic = temp;
                  mincnotdown = cnotdown;
               }
            }//restore circuit
            for (rown = row; rown - 1 > rowabove; rown--)
               circuit[rown - 1] ^= circuit[rown - 2];

            //choose best
            cnotdown = mincnotdown;
            for (cd = 0; cd < cnotdown; cd++)
            {
               cnotlist[totalgates] = (rowabove + cd); //CNOT down gate
               totalgates++;
               circuit[rowabove + 1 + cd] ^= circuit[rowabove + cd];
            }
            for (rown = row; rown - 1> rowabove+cnotdown; rown--)
            {
               cnotlist[totalgates] = -(rown); //CNOT up gate
               totalgates++;
               circuit[rown - 1] ^= circuit[rown];
            }

            row++; //adjustment so row calculation starts over
         }
      }
   }
//1. Transpose circuit
//2. Use forward substitution and backwards elimination on column
//3. Transpose back
TransposeCircuit(N, circuit, transposedcircuit);

for (row = N-1; row > column; row--)
   if(transposedcircuit[row] & flag)
   {
      //now check if for another instance of this variable
      //on the row above, necessitating a CNOT down gate
      if(transposedcircuit[row - 1] & flag)
      {
         transposedcnotlist[transposedtotalgates] = (row - 1); //CNOT down gate
         transposedtotalgates++;
         transposedcircuit[row] ^= transposedcircuit[row - 1];
      }
      else
```

```
        {
          //Check for higher instance of variable in the same column
          //i.e. row[0] represents a wire that physically is higher
          //than row[1]
          int rowabove = row - 2, instancefound = FALSE;
          while (!instancefound && rowabove >= column)
            if (transposedcircuit[rowabove] & flag)
              instancefound = TRUE;
            else
              rowabove--;
          if (!instancefound)
          {
            do
            {
              transposedcnotlist[transposedtotalgates] = -(row); //CNOT up gate
              transposedtotalgates++;
              transposedcnotlist[transposedtotalgates] = (row - 1); //CNOT down gate
              transposedtotalgates++;
              transposedcircuit[row - 1] ^= transposedcircuit[row];
              transposedcircuit[row] ^= transposedcircuit[row - 1];
            }
            while (--row > column);
          }
          else
          {   //choose best heuristic and adjust row
            int minimumheuristic, mincnotdown = 0, cnotdown = 0, rown, cd, temp;
            //first set minimumheuristic to all CNOT up
            for (rown = row; rown - 1> rowabove; rown--)
              transposedcircuit[rown - 1] ^= transposedcircuit[rown];
            //In order to keep all operations consistent recursive function
            //calls need to use the non-transposed circuit
            TransposeCircuit(N, transposedcircuit, circuit);
            minimumheuristic = LNNAED_U(N, circuit, cnotlist + totalgates, depth - 1);

            //compare against rest
            for (cnotdown = 1; cnotdown < row - rowabove; cnotdown++)
            {   // compute deltas, find cost, and ultimately restore
              transposedcircuit[rowabove + cnotdown] ^= transposedcircuit[rowabove + cnotdown +
1];

              transposedcircuit[rowabove + cnotdown] ^= transposedcircuit[rowabove + cnotdown -
1];

              TransposeCircuit(N, transposedcircuit, circuit);
              temp = LNNAED_U(N, circuit, cnotlist + totalgates, depth - 1);
              if (temp < minimumheuristic) {
                minimumheuristic = temp;
                mincnotdown = cnotdown;
              }
            }//restore transposedcircuit
            for (rown = row; rown - 1 > rowabove; rown--)
              transposedcircuit[rown - 1] ^= transposedcircuit[rown - 2];

            //choose best
            cnotdown = mincnotdown;
            for (cd = 0; cd < cnotdown; cd++)
```

```
                    {
                        transposedcnotlist[transposedtotalgates] = (rowabove + cd); //CNOT down gate
                        transposedtotalgates++;
                        transposedcircuit[rowabove + 1 + cd] ^= transposedcircuit[rowabove + cd];
                    }
                    for (rown = row; rown - 1> rowabove+cnotdown; rown--)
                    {
                        transposedcnotlist[transposedtotalgates] = -(rown); //CNOT up gate
                        transposedtotalgates++;
                        transposedcircuit[rown - 1] ^= transposedcircuit[rown];
                    }

                    row++; //adjustment so row calculation starts over
                }
            }
        }
        TransposeCircuit(N, transposedcircuit, circuit);
    }
    //Terminate both gate lists and combine
    cnotlist[totalgates] = INVALID;
    transposedcnotlist[transposedtotalgates] = INVALID;
    ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);

    return totalgates + transposedtotalgates;
}

// Linear Nearest Neighbor Alternating Elimination with Depth, solve for lower diagonal first
int LNNAED_L(int N, uint64_t * inputcircuit, int * cnotlist, int depth) //returns gate count
{
    int totalgates = 0, transposedtotalgates = 0, column, row, transposedcnotlist[CNOTLISTSIZE];
    uint64_t circuit[N], transposedcircuit[N], flag;

    if (depth == 0)
        return LNNAE_L(N, inputcircuit, cnotlist);
    for (int i = 0; i < N; i++) //use copy of circuit
        circuit[i] = inputcircuit[i];
    for (flag = (uint64_t)1 << (N - 1), column = N - 1; column > 0; column--, flag >>= 1)    //first phase of
Gaussian Elimination
    {
        for (row = 0; row < column; row++)
            if(circuit[row] & flag)
            {
                //now check if for another instance of this variable
                //on the row above, necessitating a CNOT up gate
                if(circuit[row+1] & flag)
                {
                    cnotlist[totalgates] = -(row+1); //CNOT up gate
                    totalgates++;
                    circuit[row] ^= circuit[row+1];
                }
                else
                {
                    //Check for lower instance of variable in the same column
                    int rowbelow = row + 2, instancefound = FALSE;
```

```
while (!instancefound && rowbelow <= column)
   if (circuit[rowbelow] & flag)
      instancefound = TRUE;
   else
      rowbelow++;
if (!instancefound)
{
   do
   {
      cnotlist[totalgates] = row; //CNOT down gate
      totalgates++;
      cnotlist[totalgates] = -(row+1); //CNOT up gate
      totalgates++;
      circuit[row+1] ^= circuit[row];
      circuit[row] ^= circuit[row+1];
   }
   while (++row < column);
}
else
{  //choose best heuristic and adjust row
   int minimumheuristic, mincnotup = 0, cnotup = 0, rown, cu, temp;
   //first set minimumheuristic to all CNOT down
   for (rown = row; rown + 1 < rowbelow; rown++)
      circuit[rown + 1] ^= circuit[rown];
   minimumheuristic = LNNAED_L(N, circuit, cnotlist + totalgates, depth - 1);

   //compare against rest
   for (cnotup = 1; cnotup < rowbelow - row; cnotup++)
   {
      // compute deltas, find cost, and eventually restore circuit
      circuit[rowbelow - cnotup] ^= circuit[rowbelow - cnotup + 1];
      circuit[rowbelow - cnotup] ^= circuit[rowbelow - cnotup - 1];
      temp = LNNAED_L(N, circuit, cnotlist + totalgates, depth - 1);
      if (temp < minimumheuristic)
      {
         minimumheuristic = temp;
         mincnotup = cnotup;
      }
   }

   //restore circuit
   for (rown = row; rown + 1 < rowbelow; rown++)
      circuit[rown + 1] ^= circuit[rown + 2];

   //choose best
   cnotup = mincnotup;
   for (cu = 0; cu < cnotup; cu++)
   {
      cnotlist[totalgates] = -(rowbelow - cu); //CNOT up gate
      totalgates++;
      circuit[rowbelow - 1 - cu] ^= circuit[rowbelow - cu];
   }
   for (rown = row; rown + 1 < rowbelow - cnotup; rown++)
   {
```

```
                    cnotlist[totalgates] = rown; //CNOT down gate
                    totalgates++;
                    circuit[rown+1] ^= circuit[rown];
                }

                row--;//adjustment so row calculation starts over
            }
        }
    }
//1. Transpose circuit
//2. Use forward substitution and backwards elimination on column
//3. Transpose back
TransposeCircuit(N, circuit, transposedcircuit);
for (row = 0; row < column; row++)
    if(transposedcircuit[row] & flag)
    {
        //now check if for another instance of this variable
        //on the row above, necessitating a CNOT up gate
        if(transposedcircuit[row+1] & flag)
        {
            transposedcnotlist[transposedtotalgates] = -(row+1); //CNOT up gate
            transposedtotalgates++;
            transposedcircuit[row] ^= transposedcircuit[row+1];
        }
        else
        {
            //Check for lower instance of variable in the same column
            int rowbelow = row + 2, instancefound = FALSE;
            while (!instancefound && rowbelow <= column)
                if (transposedcircuit[rowbelow] & flag)
                    instancefound = TRUE;
                else
                    rowbelow++;
            if (!instancefound)
            {
                do
                {
                    transposedcnotlist[transposedtotalgates] = row; //CNOT down gate
                    transposedtotalgates++;
                    transposedcnotlist[transposedtotalgates] = -(row+1); //CNOT up gate
                    transposedtotalgates++;
                    transposedcircuit[row+1] ^= transposedcircuit[row];
                    transposedcircuit[row] ^= transposedcircuit[row+1];
                }
                while (++row < column);
            }
            else
            {   //choose best heuristic and adjust row
                int minimumheuristic, mincnotup = 0, cnotup = 0, rown, cu, temp;
                //first set minimumheuristic to all CNOT down
                for (rown = row; rown + 1 < rowbelow; rown++)
                    transposedcircuit[rown + 1] ^= transposedcircuit[rown];
                TransposeCircuit(N, transposedcircuit, circuit);
                minimumheuristic = LNNAED_L(N, circuit, cnotlist + totalgates, depth - 1);
```

```
                    //compare against rest
                    for (cnotup = 1; cnotup < rowbelow - row; cnotup++)
                    {
                        // compute deltas, find cost, and eventually restore transposedcircuit
                        transposedcircuit[rowbelow - cnotup] ^= transposedcircuit[rowbelow - cnotup + 1];
                        transposedcircuit[rowbelow - cnotup] ^= transposedcircuit[rowbelow - cnotup - 1];
                        TransposeCircuit(N, transposedcircuit, circuit);
                        temp = LNNAED_L(N, circuit, cnotlist + totalgates, depth - 1);
                        if (temp < minimumheuristic)
                        {
                            minimumheuristic = temp;
                            mincnotup = cnotup;
                        }
                    }

                    //restore transposedcircuit
                    for (rown = row; rown + 1 < rowbelow; rown++)
                        transposedcircuit[rown + 1] ^= transposedcircuit[rown + 2];

                    //choose best
                    cnotup = mincnotup;
                    for (cu = 0; cu < cnotup; cu++)
                    {
                        transposedcnotlist[transposedtotalgates] = -(rowbelow - cu); //CNOT up gate
                        transposedtotalgates++;
                        transposedcircuit[rowbelow - 1 - cu] ^= transposedcircuit[rowbelow - cu];
                    }
                    for (rown = row; rown + 1 < rowbelow - cnotup; rown++)
                    {
                        transposedcnotlist[transposedtotalgates] = rown; //CNOT down gate
                        transposedtotalgates++;
                        transposedcircuit[rown+1] ^= transposedcircuit[rown];
                    }

                    row--;//adjustment so row calculation starts over
                }
            }
        }
        TransposeCircuit(N, transposedcircuit, circuit);
    }
    //Terminate both gate lists and combine
    cnotlist[totalgates] = INVALID;
    transposedcnotlist[transposedtotalgates] = INVALID;
    ReverseandTransposeCNOTList(transposedcnotlist, cnotlist + totalgates);
    return totalgates + transposedtotalgates;
}

void CopyCircuit(int N, uint64_t * source, uint64_t * destination) {
    for (int i = 0; i < N; i++)
        destination[i] = source[i];
}
```

Appendix B: 16 Fundamental Types of Linear Reversible Circuit Synthesis.

Elimination Type	Sub-type	Input	Gate Output Sequence
Gaussian	Upper Triangle Matrix	Matrix	Reversed
		Transposed Matrix	Transposed
		Inverse Matrix	Normal
		Transposed Inverse Matrix	Reversed and Transposed
	Lower Triangle Matrix	Matrix	Reversed
		Transposed Matrix	Transposed
		Inverse Matrix	Normal
		Transposed Inverse Matrix	Reversed and Transposed
Alternating	Upper Diagonal	Matrix	Reversed
		Transposed Matrix	Transposed
		Inverse Matrix	Normal
		Transposed Inverse Matrix	Reversed and Transposed
	Lower Diagonal	Matrix	Reversed
		Transposed Matrix	Transposed
		Inverse Matrix	Normal
		Transposed Inverse Matrix	Reversed and Transposed

Appendix C: Systolic 2D Shift Register LNNAE Data Flow

The following 19 figures illustrate flow of data in the author's systolic 2D shift register LNNAE system. When intermediate values change along the edges of the matrix they appear with a brown background, and values achieve their identity matrix value they appear with a green background.

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

0	B_{42}	B_{43}	B_{44}
$\mathbf{A_{11}}$	$\mathbf{A_{12}}$	$\mathbf{A_{13}}$	$\mathbf{A_{14}}$
$\mathbf{A_{21}}$	$\mathbf{A_{22}}$	$\mathbf{A_{23}}$	$\mathbf{A_{24}}$
B_{31}	B_{32}	B_{33}	B_{34}

0	C_{32}	C_{33}	C_{34}
0	$\mathbf{B}_{42}$	$\mathbf{B}_{43}$	$\mathbf{B}_{44}$
$\mathbf{A}_{11}$	$\mathbf{A}_{12}$	$\mathbf{A}_{13}$	$\mathbf{A}_{14}$
C_{21}	C_{22}	C_{23}	C_{24}

0	D_{22}	D_{23}	D_{24}
0	$\mathbf{C_{32}}$	$\mathbf{C_{33}}$	$\mathbf{C_{34}}$
0	$\mathbf{B_{42}}$	$\mathbf{B_{43}}$	$\mathbf{B_{44}}$
1	D_{12}	D_{13}	D_{14}

E_{24}	0	$\mathbf{D_{22}}$	E_{23}
E_{34}	0	$\mathbf{C_{32}}$	E_{33}
E_{44}	0	$\mathbf{B_{42}}$	E_{43}
0	1	$\mathbf{D_{12}}$	E_{13}

F_{23}	$\mathbf{E_{24}}$	0	F_{22}
F_{33}	$\mathbf{E_{34}}$	0	F_{32}
F_{43}	$\mathbf{E_{44}}$	0	F_{42}
0	0	1	F_{12}

F_{22}	F_{23}	E_{24}	0
F_{32}	F_{33}	E_{34}	0
F_{42}	F_{43}	E_{44}	0
0	0	0	1

0	0	0	1
F_{22}	F_{23}	E_{24}	0
F_{32}	F_{33}	E_{34}	0
F_{42}	F_{43}	E_{44}	0

0	G_{43}	G_{44}	0
0	0	0	1
$\mathbf{F_{22}}$	$\mathbf{F_{23}}$	$\mathbf{E_{24}}$	0
G_{32}	G_{33}	G_{34}	0

0	H_{33}	H_{34}	0
0	$\mathbf{G_{43}}$	$\mathbf{G_{44}}$	0
0	0	0	1
1	H_{23}	H_{24}	0

0	0	H_{33}	H_{34}
0	0	G_{43}	G_{44}
1	0	0	0
0	1	H_{23}	H_{24}

I_{34}	0	0	I_{33}
I_{44}	0	0	I_{43}
0	1	0	0
0	0	1	I_{23}

I_{33}	I_{34}	0	0
I_{43}	I_{44}	0	0
0	0	1	0
0	0	0	1

0	0	0	1
I_{33}	I_{34}	0	0
I_{43}	I_{44}	0	0
0	0	1	0

0	0	1	0
0	0	0	1
I_{33}	I_{34}	0	0
I_{43}	I_{44}	0	0

0	1	0	0
0	0	1	0
0	0	0	1
1	J_{34}	0	0

0	0	1	0
0	0	0	1
1	0	0	0
0	1	J_{34}	0

0	0	0	1
1	0	0	0
0	1	0	0
0	0	1	J_{34}

1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

Appendix D: Pseudo-method Test Results

Testing linear reversible circuit synthesis on 8 wires	Average Number of CNOT gates	Time (s)
Initial Gate Search (1)	50.375	0.1785
Truncated Initial Gate Search (2)	50.5	0.9515
Best of 8 LNNAED at depth=4	51.175	0.519
Truncated Initial Gate Search (3)	52.225	0.01375
Best of 8 LNNAED at depth=1 and Best of 8 LNNAED at depth=1	52.475	0.0015
Best of 8 LNNGED at depth=4	53.275	0.00275

Testing linear reversible circuit synthesis on 16 wires	Average Number of CNOT gates	Time (s)
Truncated Initial Gate Search (4)	255.6	11.5055
Best of 8 LNNAED at depth=2	257.2	3.75825
Initial Gate Search (5)	257.425	24.0875
Truncated Initial Gate Search (6)	258.925	18.8175
Best of 8 LNNGED at depth=4	259.05	13.524
Best of 8 LNNAED at depth=1 and Best of 8 LNNAED at depth=1	267.7	0.07825

Testing linear reversible circuit synthesis on 32 wires	Average Number of CNOT gates	Time (s)
Truncated Initial Gate Search (7)	1255.35	24.082
Best of 8 LNNGED at depth 2	1258.925	5.02275
Initial Gate Search (8)	1259.025	85.83775
Best of 8 LNNAED at depth=1 and Best of 8 LNNAED at depth=1	1268.75	2.41675
Truncated Initial Gate Search (9)	1271.1	4.5465

Testing linear reversible circuit synthesis on 64 wires	Average Number of CNOT gates	Time (s)
Best of 8 LNNAED at depth=1 and Best of 8 LNNAED at depth=1	5566.125	84.25575
LNNGED depth 2	5570.575	50.4915
Initial Gate Search (10)	5611.025	33.95075

(1) Each iteration searching all functions within two CNOT gates using Best of 2 LNNGED at depth=2.

(2) Two iterations searching all functions within two CNOT gates using Best of 2 LNNAED at depth=2, then Best of 2 LNNAED at Depth=4.
(3) Two iterations searching all functions within two CNOT gates using Best of 2 LNNGED at depth=2, then Best of 2 LNNGED at Depth=4.
(4) Two iterations searching all functions within two CNOT gates using Best of 2 LNNAED at depth=1, then Best of 2 LNNAED at Depth=2.
(5) Each iteration searching all functions within two CNOT gates using Best of 2 LNNGED at depth=2.
(6) Two iterations searching all functions within two CNOT gates using Best of 2 LNNGED at depth=2, then Best of 2 LNNGED at Depth=4.
(7) Two iterations searching all functions within two CNOT gates using Best of 2 LNNGED at depth=1, then Best of 2 LNNGED at Depth=2.
(8) Each iteration searching all functions within one CNOT gate using LNNGED at depth=1.
(9) Two iterations searching all functions within two CNOT gates using Best of 2 LNNAED at depth=0, then Best of 2 LNNAED at Depth=1.
(10) Each iteration searching all functions within one CNOT gate using LNNGE.