QUANTUM LOGIC IMPLEMENTATION OF UNARY

ARITHMETIC OPERATIONS WITH INHERITANCE

Approved by:

_____

Dr. Mitchell A. Thornton, Advisor

_____

Dr. David Matula

_____

Dr. Sukumaran Nair

QUANTUM LOGIC IMPLEMENTATION OF UNARY ARITHMETIC OPERATIONS WITH INHERITANCE

A Thesis Presented to the Graduate Faculty of the

School of Engineering

Southern Methodist University

in

Partial Fulfillment of the Requirements

for the degree of

Master of Science in Computer Engineering

with a

Major in Computer Engineering

by

Laura Spenner

(B.S.Cp.E, Southern Methodist University)

May 17, 2008

Spenner, Laura B.S.Cp.E., Southern Methodist University, 2006

Quantum Logic Implementation of Unary
Arithmetic Operations with Inheritance

Advisor: Professor Mitchell A. Thornton

Master of Science conferred May 17, 2008

Thesis completed April 17, 2008

An investigation into the architecture of multiple-valued quantum logic arithmetic circuits is the subject of this research. In order to efficiently design multiple-valued quantum logic circuits for the realization of unary arithmetic operations with inheritance, universal sets of permutation gates are found and used. A universal gate set is defined as a small set of gates that can produce all possible functions. Prior to the development of a method for verifying quintary quantum logic universal gate sets, an investigation into ternary quantum logic circuit design was performed. Multiplicative inverse, discrete logarithm, and exponentiation circuits were designed in ternary quantum logic utilizing the inheritance principle. The multiplicative inverse was selected to perform further research into quintary quantum logic circuit design. Before designing multiplicative inverse circuits in quintary quantum logic, universal gate sets of quintary quantum logic permutation gates must be found. With the total number of permutations equal to 5! = 120 for quintary quantum logic, small universal gate sets that can be used to generate those permutations are important to future quintary quantum logic design. The method used to fully determine whether or not a set of permutation gates forms a universal gate

set is to permute the order of the gates used to form the permutations and to permute the gates chosen to form the universal gate set. The identity permutation is the arbitrary starting point for forming permutations, so if a universal gate set isn't found initially, one of the previously generated permutations is used as a new starting point for testing the same set of gates. Universal gate sets of four quintary quantum logic gates are consistently achieved. One of the universal gate sets is used to successfully design multiplicative inverse circuits in quintary quantum logic exploiting the inheritance principle.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ACKNOWLEDGEMENTS

There are so many people to thank, because there are so many people who helped me achieve this goal. My advisor Dr. Thornton deserves many, many thanks for guiding me to achieve this goal. My parents also deserve my gratitude for encouraging me through all of my difficult and doubtful times. I cannot forget to thank my brother, Kent, for his help in preventing what could have been inevitable insanity. I would also like to thank my friend, classmate and peer, Kelly Hawkins, for his sense of humor and willingness to share it with me. There are others that deserve thanks for their assistance not mentioned by name. To all those nameless benefactors, thank you so much!

Chapter 1

INTRODUCTION

Current technology employs binary logic, classical bits based on transistor logic. The concept of binary bits of information is ingrained in classical computer science and engineering. A faster, cheaper way to use these bits has been sought for decades. Transistors have become smaller, faster, and more power efficient, but atomic size limits are being reached. Regardless of the physical barriers to further development, technology conscious consumers are continually demanding the fastest computing power money can buy. Improvement in classical logic technology is slowing down. Dramatic advancements must come from a different direction such as quantum logic. The qualities of bijection and inherent parallelism in quantum logic lend themselves to a tremendous increase in speed and efficiency for certain classes of algorithms. Quantum logic modeling indicates that sometime in the future, faster computing can be provided by quantum computers and quantum coprocessors [23, 24, 25].

The limits on transistor size are not the only barriers to progress for faster, more efficient computing. Classical logic is dominated by base 2, binary logic to realize circuit designs. This small radix logic does not offer as much computational power as a larger radix that multiple-valued logic can provide. Larger radices allow more values to be represented in fewer digits, among other benefits of multiple-valued logic. Research in

the use of quantum logic gates that exploits higher dimensions shows promise for practical implementations of multiple-valued logic. Recently, an Australian group published results in which radix 3 or ternary quantum logic gates that are more powerful than any of the previously published developments were constructed and tested. The key to this ground-breaking design created by the University of Queensland based group is the ability to exploit the higher dimensions existing in quantum mechanics [21]. These results regarding the ability to utilize higher dimensions to create more powerful quantum logic circuit designs lend credence to the multiple-valued quantum logic investigation performed in the research associated with this thesis.

The ability to perform arithmetic operations quickly is at the core of classical computing. Quantum computing does not change this requirement; instead, a faster way to perform necessary arithmetic operations is offered. This research demonstrates the possibilities for multiple-valued quantum logic specifically in the area of unary arithmetic operations. Unary arithmetic circuits are designed in ternary and quintary quantum logic. Before creating quintary quantum logic circuits, universal gate sets must be verified. The successful accomplishment of these tasks indicates the usefulness of research into multiple-valued circuit design.

A significant amount of background research is essential to understanding the processes and designs created and used in this project. Detailed discussion of quantum circuit topics including the representation of quantum information, the standard notation known as Dirac notation, the Hilbert space, and quantum gate structure and reversibility are presented in Chapter 2. Quantum gate cascades are also examined. These topics

form the foundation for utilizing quantum logic to design multiple-valued logic unary arithmetic circuits. Several topics in number theory, specifically the inheritance principle, residue arithmetic, relatively prime numbers, and the sieve of Eratosthenes are also included as especially pertinent to this research. This establishes the role of the number theory concepts that support the methods employed in the design and verification of arithmetic in quintary quantum logic circuits.

The background topics discussed at length in the previous chapter allow for the exploration of the method used in designing unary arithmetic operations in quantum logic and quintary quantum logic universal gate set verification in Chapter 3. Multiple-valued logic design requires the information being manipulated to be encoded into digit sets. Descriptions of the digit sets employed for the designs included and method of encoding into the digit sets are a vital part of the design process. Before quintary quantum logic designs for unary arithmetic operations can be created, a more thorough understanding of multiple-valued logic is achieved by investigation of ternary quantum logic. These results can be examined in the designs presented as part of this chapter.

The methods for verifying both the quintary and ternary quantum logic designs are the focus of Chapter 4. The two verification methods utilized, exhaustive simulation and a relatively new QMDD-based verification method, along with the designs for quintary quantum logic multiplicative inverse circuits are part of this chapter. Observations corresponding to the quintary quantum logic designs are included.

Discussion in Chapter 5 relates to the interesting results yielded by the research, the significance of which is highlighted. As with many research topics, while very good

results are achieved, the exploration into the topic brings new areas of further research to light. Such topics are also included in this chapter.

Chapter 2

BACKGROUND

This chapter discusses background topics that support this research. First, the notion of quantum circuit cascades and supporting mathematical concepts such as Dirac notation and Hilbert vector spaces are described. Next, quantum logic gates and their representation are presented. Finally, the mathematical inheritance principle is described that is exploited in designing quantum logic unary arithmetic circuits. Inheritance is a property that is used in the implementation of the unary arithmetic multiplicative inverse and discrete logarithm quantum circuits. In order to understand the construction of the numerical tables used to specify the multiplicative inverse pairs, some basic concepts from number theory are reviewed.

2.1 Quantum Circuits

Classical logic relies on binary representation of information using electronic circuits. Quantum logic and quantum computing do not function in the same way. There are several concepts that must be addressed to fully understand quantum logic and the design of quantum circuits as will be discussed later in this work.

2.1.1 Quantum Information Representation

The smallest unit for representing information in classical logic is the bit. The quantum logic counterpart of the bit is the quantum bit, also known as a qubit. The terms bit and qubit imply a binary system, the term quantum digit in a *d*-ary system, also known as qudits, have many properties and qualities that classical bits do not have based on physical and mathematical constraints of quantum mechanics. [15]

2.1.2 Dirac Notation

Qudits are able to have a state in the sense that classical bits may have a state. These qudit states are described by vectors, not as scalar values as in classical logic. The standard notation used for the vector states of a qudit, as established in quantum mechanics, is known as Dirac notation. The vector notation known as *ket vectors* appears as $|0\rangle$, $|1\rangle$ and $|2\rangle$ to represent a ternary system for example. Ket vectors are column vectors, the values of the ket vectors in a ternary system are $|0\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, $|1\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$ and $|2\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$. The transpose of the ket vectors is referred to as a *bra vector*. The relationship between ket and bra vectors is the Hermitian conjugate. The bra vector values appear as $\langle 0|$, $\langle 1|$ and $\langle 2|$ to continue with the established example. The bra vectors are row vectors with values $\langle 0| = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$, $\langle 1| = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$ and $\langle 2| = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$ [14]. The states

described by Dirac notation are used to form superpositions, also known as linear combinations, of the form $|\Phi\rangle = \alpha|0\rangle + \beta|1\rangle$ [15].

Dirac notation can also be used to represent inner and outer products of vectors. An inner product for $|a\rangle$ and $|b\rangle$ in Dirac notation is $\langle a|b\rangle = \begin{bmatrix} a_0^* & a_1^* \cdots a_n^* \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_n \end{bmatrix} = a_0^* b_0 + a_1^* b_1 + \cdots + a_n^* b_n$. This product results in a complex scalar value in general. When a vector space has an inner product, it is known as an inner product space. An outer product for $|a\rangle$ and $|b\rangle$ in standard quantum mechanical notation is a matrix calculated by the equation $|a\rangle\langle b| = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_0^* & b_1^* \cdots b_n^* \end{bmatrix} = \begin{bmatrix} a_0 b_0^* & a_0 b_1^* \cdots a_0 b_n^* \\ a_1 b_0^* & a_1 b_1^* \cdots a_1 b_n^* \\ \vdots & \vdots \quad \ddots \\ a_n b_0^* & a_n b_1^* \cdots a_n b_n^* \end{bmatrix}$.

2.1.3 Hilbert Space

Quantum logic circuits are based on quantum mechanics and mathematics. A Hilbert space is a concept from mathematics that is explained here before a discussion of quantum circuits can proceed.

Definition: Hilbert Space - denoted as $\mathbf{H}_n$, is a complex vector space that can be *n*-dimensional with an inner product. $\mathbf{H}_n$ is isomorphic with $\mathbf{C}^n$, the *n*-dimensional vector space over the field of complex numbers.

The inner product may also be referred to as the scalar product because a Hilbert space covers the field of complex numbers. A Hilbert space over *n*-dimensions can be specified with *n* basis vectors. Furthermore, vectors in the Hilbert space $\mathbf{H}_n$ are generally modeled as column vectors that have *n* complex components. For example, vectors in $\mathbf{H}_2$ have two complex components and the space spanned by two basis-state vectors.

It should also be noted that a Hilbert space is isomorphic with the field of complex scalar numbers. This isomorphism means that a Hilbert space, $\mathbf{H}_n$, is bijective with $\mathbf{C}^n$. The property of being bijective in this case may be described as allowing a one-to-one mapping between states in a Hilbert space. This is a useful quality for designing unary arithmetic circuits that have the property of bijection. [14]

2.1.4 Quantum Gates

Quantum gates can be thought of as the building blocks for a quantum circuit and they are conveniently modeled as a mathematical transformation describing the manipulation of the quantum state of some particle. The focus will be on the mathematical system in this case. The mathematical system takes the form of transfer matrices whose products with initial vector states form the resulting vector states. The general form of the transformation may be represented as $|\psi_b\rangle = \mathbf{U} \mid \psi_a\rangle$ [14]. For the previous example using two basis states and allowing $\mathbf{U}_{10} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, then $\mathbf{U}_{10} |0\rangle = |1\rangle$. In the case of ternary quantum logic design, the single qudit gates and their associated matrices have been reasonably established in [4, 5, 6, 7, 8, 9, 10]. In Table 2.1, the six

quantum gates represent all permutations of the basis vectors for a ternary system $|0\rangle$, $|1\rangle$, and $|2\rangle$ [19]. Permutations generated by each gate are made easier to recognize in the table by denoting the transfer matrix, $\mathbf{U}_{ijk} = |i\rangle\langle 0| + |j\rangle\langle 1| + |k\rangle\langle 2|$, simply as (i, j, k). This gate set is the basis of the design of the multiplicative inverse and discrete logarithm circuits.

Table 2.1: Six Quantum Ternary Gates

Name	**Permutation (i, j, k)**	**Symbol**	**Reference**
Identity	(0,1,2)	I	-
Modulo-add by 1	(1,2,0)	C1	[4,5,7]
Modulo-add by 2	(2,0,1)	C2	[4,5]
Negation	(2,1,0)	N	[5,7]
Neg. Mod. 1	(0,2,1)	NC1	[5,6]
Neg. Mod. 2	(1,0,2)	NC2	[5,6]

The quantum gate denoted as Modulo – add by 1 in Table 1 forms the matrix $\mathbf{U}_{120}$.

$$\mathbf{U}_{120} = |1\rangle\langle 0| + |2\rangle\langle 1| + |0\rangle\langle 2| = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}\begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{U}_{120} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The transfer matrix $\mathbf{U}_{120}$ can now be used to calculate the resultant vector when it is multiplied with a qudit vector. For the example, $|1\rangle$ can be multiplied with $\mathbf{U}_{120}$ to achieve the resulting qudit vector.

$$\mathbf{U}_{120} \otimes |1\rangle = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

2.1.5 Quantum Gates and Reversibility

The function of a quantum gate is based on mathematical computations. Mathematical computations are reversible if, and only if, there is no loss of information. Thus, quantum gates and circuits are reversible. A discussion of reversibility in gates and circuits will start with binary reversible functions.

Definition: Binary reversible function (size $n \times n$) – n inputs and n outputs, is a unique and distinct mapping from $\{0, 1\}^n$ onto $\{0, 1\}^n$.

The mapping of the $n \times n$ binary reversible function is onto and one-to-one for domain and range of $\{0, 1\}^n$. This function is a permutation of $\{0, 1\}^n$, which means that it can be represented as a $2^n \times 2^n$ permutation matrix. One example of a reversible gate is the NOT gate. A NOT gate is also a self-inverse, meaning that it can cause the reversal. Symmetric functions are not reversible, so reversible functions are not symmetric. This is due to symmetry causing distinct sets, or subsets, of inputs to map to a common output, violating the one-to-one correspondence required by reversibility. Familiar classical

logic gates, namely AND, OR, NAND, NOR, XOR, and XNOR, are symmetric and therefore not reversible. This concept is generalized for use in multiple-valued logic [20].

2.2 Quantum Gate Cascade Representation

Quantum logic circuits are constructed using cascades of quantum gates. In diagrams, lines are considered to represent wires while they are not necessarily meant to be physical wires. The lines actually represent the movement of a physical particle through space or a passage of time [15]. An example of a very simple quantum logic diagram can be found in Figure 2.1 representing a negation quantum gate.

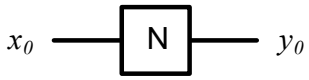

Figure 2.1: Quantum Logic Circuit for Negation of Single Input

Assuming the gate shown in Figure 2.1 is describing a negation gate for ternary logic, the matrix that describes the negation gate would appear as $\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$.

Quantum logic gates may utilize control inputs. These control inputs only allow the quantum gate to operate when the control condition is met. For example, the matrix $\mathbf{U}_{120}$, called C1 in Table 2.1, could have a control value of two. The C1 gate would not operate

on qudit x_1 unless x_0 is a value of two. Figure 2.2 contains the schematic symbol used to depict the C1 gate with a control input of 2 needed.

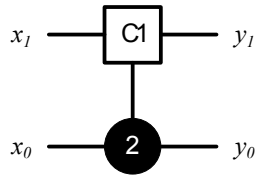

Figure 2.2: Controlled Ternary Gate Example

Assuming the gate in Figure 2.2 is once again in ternary logic, the matrix to describe the operation of this gate is $\begin{bmatrix} 1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0 \end{bmatrix}$.

The permutation gates may have as many controls as are necessary to achieve the desired logic, whether the number of controls is zero or many. For the creation of a larger circuit more than one control signal for a single quantum logic gate is necessary to achieve the desired output result. Figure 2.3 is the diagram of a simple quantum logic circuit using a multiple control input gate.

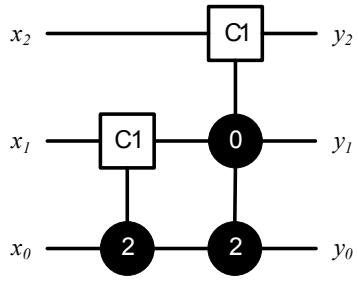

Figure 2.3: Simple Single Control and Multiple Control Gate Circuit Cascade

The quantum gate with multiple controls shown in Figure 2.3 is represented by the matrix:

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

The output y_0 has the same value as the x_0, because there is no quantum gate on that line. The output y_1 has the value of x_1 plus 1 modulo 3 when the value on x_0 is equal to 2,

but y_1 has the same value as x_1 when x_0 is not equal to 2. The output y_2 has the value x_2 plus 1 modulo 3 when the value on x_1 is equal to 0 and the value on x_0 is equal to 2, but y_2 has the same value as x_2 when x_1 is not equal to 0 and x_0 is not equal to 2.

2.3 Inheritance Principle

The property of inheritance is described in [1]. The use of this property is confined to the binary case relating to multiplicative inverses and later other functions such as the discrete logarithm and exponentiation circuits. This concept was developed with classical logic and is applied here to quantum circuits, both for the binary and the multi-valued case.

Definition: Inheritance - Let $f(a_{k-1}a_{k-2}\cdots a_0) = b_{k-1}b_{k-2}\cdots b_0$ be a one-to-one mapping of a domain of k-bit strings to a co-domain of k-bit strings for all $k \geq 1$. The function f demonstrates the property of inheritance when $f(a_{k-1}a_{k-2}\cdots a_0) = b_{k-1}b_{k-2}\cdots b_0$, and for all $k \leq n$, $f(a_{n-1}a_{n-2}\cdots a_{k-1}a_{k-2}\cdots a_0) = b_{n-1}b_{n-2}\cdots b_{k-1}b_{k-2}\cdots b_0$. Thus, the n^{th} digit, b_{n-1}, of f is only dependent on the digit values a_i where $i \leq n$.

This allows the design of a circuit based on the lower order digit solutions. Higher order digits may be ignored when designing for a particular output digit. However, it should be noted that while ordinary integer addition and multiplication satisfy inheritance, but are not one-to-one.

2.4 Residue Arithmetic

To specify the functionality of multiplicative inverse circuits, tables of multiplicative inverse pairs are created. The foundation for developing multiplicative inverse tables is based in residue arithmetic. Following [11] we employ the notation $|a|_n$ to denote the standard residue $0 \leq i \leq n\text{-}1$, such that $a \equiv i \bmod n$.

Definition: Modular multiplicative inverse - a number, a, has a multiplicative inverse, denoted as a^{-1}, if a^{-1} satisfies the equation $\left|a \times a^{-1}\right|_{p^k} = 1$, where p is the radix, or base of the number, and k is the number of digits used to represent the numbers a and a^{-1}.

The modular multiplicative inverse is clearly a unary arithmetic operation that is bijective over the set if values that have inverses and is an excellent candidate for realization in a quantum logic gate cascade. The case where p is prime is of particular interest; since all k digit integers that have non-zero low order digits will have multiplicative inverses. Furthermore, the fixed-precision multiplicative inverse function also obeys the property of inheritance. The fixed-point discrete logarithm and exponentiation functions are also bijective and exhibit inheritance.

Definition: Modular discrete logarithm – the value e that satisfies $n = |B^e|_{p^k}$, where p is the radix, k is the number of digits, and B is the logarithmic base [3, 19].

Definition: Modular exponentiation – the value n that satisfies the formula $n = |B^e|_{p^k}$, where p is the radix, k is the number of digits, B is the logarithmic base, e is the discrete logarithm [3, 19].

These unary arithmetic operations are of interest due to their ability to reduce the amount of computing power necessary for expensive arithmetic operations since multiplications become addition operations and divisions become subtraction operations. Multiplicative inverse values can also be used to implement the division operation, because multiplying by an inverse is equivalent to dividing by the original number.

2.5 Relatively Prime Numbers

In developing multiplicative inverse pairs, the concept of numbers being relatively prime to other numbers is reviewed here. The property of being relatively prime to the base or radix of the multiplicative inverse circuit is used to determine whether or not a number has a multiplicative inverse.

Definition: Relatively prime - A pair of integer values is relatively prime if their greatest common divisor is one [18].

This means that the integer pair shares no common factors except one. For the purposes of the multiplicative inverse circuit design, only numbers that are relatively prime to the radix have multiplicative inverses. If a is not relatively prime to the radix,

then a will have common factors with the p^k making it impossible to have an inverse that will allow the product to have unity residue. The possible number of multiplicative inverses that can be calculated is determined by the ratio (p – number prime factors of p) / p of all integers in the range of 0 to p^k. For the ternary case, 2 / 3 of all integers in the range of 0 to 3^k have multiplicative inverses. For the case of p = 6, only 2 / 6, or 1 / 3, of all integers in the range of 0 to 6^k. This makes the choice of p more significant. By choosing a p value that is prime, a greater ratio of the integers have multiplicative inverses.

2.6 Sieve of Eratosthenes

Eratosthenes' sieve [17] is another concept from number theory that is utilized to develop multiplicative inverse tables. One of the oldest algorithms for finding prime numbers can be altered to find numbers that are relatively prime to the radix. Any number that has a previously determined prime number as a factor, is not prime. In the case of relatively prime numbers, any number that has a prime factor of the radix as a factor is not relatively prime to the radix. In the case of p = 6, six has prime factors two and three. The numbers two and three will not have multiplicative inverses. The number four will also not have a multiplicative inverse, because it shares the prime factor two with six. [17]

2.7 Table Generation Method

Using the number theory concepts discussed, multiplicative inverse tables with user defined radices, numbers of digits, and encodings can be generated. The radix, p, and the number of digits, k, to be used are also specified parameters. This particular table generation method uses the Eratosthenes' sieve to determine which integers the system with a particular radix may have as prime factors. It is sufficient to test for numbers prime to p independent of k. These prime factors are noted for use in confirming that the integer value being considered is relatively prime to the radix.

The process is initialized with the smallest integer in the range of values determined by the radix and number of digits. That starting integer is one for the natural digit encoding and negative modulo equivalent for signed digit encoding. Before searching for a multiplicative inverse, the integer is checked against the prime factor list using the modulo operator. With no prime factors, the search for an inverse of the integer value is allowed. An integer value, starting with the same starting value of one, is multiplied by the current original integer value, then the modulus of p^k is applied. If the residue is one, the multiplicative inverse of the original value has been found.

The tables generated by this method can be used to manually design binary and multiple-valued multiplicative inverse circuits and, may in the future, be used with automated methods for designing circuits.

2.8 Table Compaction

The number of entries in a table of multiplicative inverses is determined by the formula $n = p^k \times ((p - 1) / p)$, where p is a prime radix and k is the number of digits. As an example, Table 2.2 contains the multiplicative inverse pairs for the ternary radix, $p = 3$, and four digit values. The full table contains fifty-four multiplicative inverse entries, in radix 10 and radix 3 number systems with $k = 2$ and $k = 4$ digits respectively.

Table 2.2: Multiplicative Inverses Modulo 3^4 in Decimal and Ternary Number Systems

Decimal *a*	**Decimal a^{-1}**	**Ternary *a***	**Ternary a^{-1}**
01	01	0001	0001
02	41	0002	1112
04	61	0011	2021
05	65	0012	2102
07	58	0021	2011
08	71	0022	2122
10	73	0101	2201
11	59	0102	2012
13	25	0111	0221
14	29	0112	1002
16	76	0121	2211
17	62	0122	2022
19	64	0201	2101
20	77	0202	2212
22	70	0211	2121
23	74	0212	2202
25	13	0221	0111
26	53	0222	1222
28	55	1001	2001
29	14	1002	0112
31	34	1011	1021
32	38	1012	1102
34	31	1021	1011
35	44	1022	1122

37	46	1101	1201
38	32	1102	1012
40	79	1111	2221
41	02	1112	0002
43	49	1121	1211
44	35	1122	1022
46	37	1201	1101
47	50	1202	1212
49	43	1211	1121
50	47	1212	1202
52	67	1221	2111
53	26	1222	0222
55	28	2001	1001
56	68	2002	2112
58	07	2011	0021
59	11	2012	0102
61	04	2021	0011
62	17	2022	0122
64	19	2101	0201
65	05	2102	0012
67	52	2111	1221
68	56	2112	2002
70	22	2121	0211
71	08	2122	0022
73	10	2201	0101
74	23	2202	0212
76	16	2211	0121
77	20	2212	0202
79	40	2221	1111
80	80	2222	2222

It should be observed that the multiplicative inverse is a commutative function, since $a \times a^{-1} = a^{-1} \times a$. This allows half of the table entry pairs to be removed as redundant. More table entries may be removed by recognizing that only one member of a subgroup with the same primitive generator function needs to be included [12, 17, 18]. For example, the multiplicative inverse pair (56, 68) can be used to generate the multiplicative inverse pair (07, 58). This is accomplished by finding all the prime factors

of both elements and using only the factors found to produce the other multiplicative inverse pairs. To continue the example, the prime factors of (56, 68) are {2, 2, 2, 7, 2, 2, 17}. The (07, 58) pair is generated by selecting the 7 as one element, multiplying the remaining prime factors and applying modulo 81 to achieve the second element of 58. The combined reductions allow a compact table of only 14 entries, which is given in Table 2.3.

Table 2.3: Multiplicative Inverses Modulo 3^4 With Redundant Entries Removed

Decimal *a*	**Decimal a^{-1}**	**Ternary *a***	**Ternary a^{-1}**
01	01	0001	0001
02	41	0002	1112
04	61	0011	2021
08	71	0022	2122
16	76	0121	2211
32	38	1012	1102
64	19	2101	0201
47	50	1202	1212
13	25	0111	0221
26	53	0222	1222
52	67	1221	2111
23	74	0212	2202
46	37	1201	1101
11	59	0102	2012

The ability to perform the compaction of the multiplicative inverse table also further illustrates the one-to-one relationship within the set of values for this unary arithmetic function. It should also be noted that $\left|(p^k - a)\times(p^k - a^{-1})\right|_{p^k} = 1$ can be used to generate inverses for $\left|a \times a^{-1}\right|_{p^k} = 1$.

Chapter 3

DESIGN IMPLEMENTATION

In the case of ternary logic, unary arithmetic circuits are designed that exploit the inheritance principle. Examples of ternary discrete logarithm and exponentiation circuits are also provided. In order to explore the area of quintary quantum logic implementations, a universal gate set was first determined. A method was established to test for a universal gate set and a sample multiplicative inverse circuit is designed using quintary quantum logic permutation gates.

3.1 Quantum Logic Digit Encoding

The digits of a function being used to design a quantum logic circuit may be encoded in different ways in the circuit implementations. The first encoding utilizes a digit set comprised of signed digits. The digits 0, 1, . . . , p-1 are used to represent the value, where $p > 2$. When a negative modulo equivalent encoding is used, signed digits that belong to the same modulo equivalence class as the larger positive digits are used to create the truth table associated with the circuit to be designed. For example, when $p = 3$ and a signed modulo equivalent encoding is used, the digit sets have the relationship {0, 1, 2} $\rightarrow$ {0, 1, -1}. While alternative mappings may be used, experimental results show that the resulting circuits are not as compact as the given mapping. A four digit ternary

multiplicative inverse pair using negative digit encoding, (2, -40) → (0002, -1-1-1-1) for example, uses the positive modulo equivalent digit to represent the control inputs in the design. In the case of the example multiplicative inverse pair, -1 is represented by 2 in a circuit design, but is interpreted as a signed modular equivalent by the designer.

3.2 Unary Arithmetic Quantum Circuits

As a starting point in the exploration of unary arithmetic quantum logic circuit design, the ternary base circuits of multiplicative inverses and discrete logarithms were designed. The method used for manually designing the circuits relies on exploitation of the inheritance principle. The portion of the quantum logic gate cascade that generates the least significant digit output is designed first. The least significant digit for ternary multiplicative inverse does not require any operation to be performed, it remains the same. The next digit may use the lower order digit as a control input. However, the ternary multiplicative inverse sees the same digit change for both values ending in one and two. The gates do not use control digits in this case. The numbers that end in zero need not be considered for any of the digits, because they are in the same modulo equivalence class as three, having no multiplicative inverse. The higher order digits do use control digits to perform the proper transformations. The number of control digits and gates used to realize the circuits can be reduced by finding groups of control qudits that cause the same permutation of the original values.

The four digit ternary quantum logic multiplicative inverse circuit using a modular digit value encoding will be used as an example to explain the design process. The

principle of inheritance is exploited in the design of the unary arithmetic operations. This means that the designer must first examine the lowest order digit values in the table. The multiplicative inverse table has no entries that have a lowest order value of 0, because a number with a zero in the lowest order digit has the radix as a root and cannot be relatively prime to the base. After examining the lowest order digit values, it is observed that the digits do not change from input to output. Figure 3.1 illustrates what this circuit would look like for the single qudit.

Figure 3.1: Ternary Multiplicative Inverse Circuit Design for a Single Qudit

The next digit higher order digit is examined to determine the changes from input to output. It is determined that the output seen is that of an "N" and "C1" gates for all inputs to achieve the correct output. Figure 3.2 is the diagram realizing this addition to the ternary quantum logic circuit design of a multiplicative inverse of two qudits.

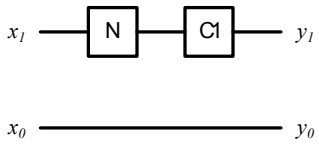

Figure 3.2: Ternary Multiplicative Inverse Circuit Design for Two Qudits

The next higher order digit is examined so that the design may be extended to include the third digit. The pattern detected in the indicated output shows that to obtain the correct output qudit, the "N" gate is the only gate necessary for all input values. A "C1" gate with a control input from y_1 of 0, a "C2" gate with control inputs y_1 of 2 and y_0 of 1, and a "C2" gate with control inputs y_1 of 1 and y_0 of 1 complete the design. Figure 3.3 is a circuit design of this description of the three digit circuit.

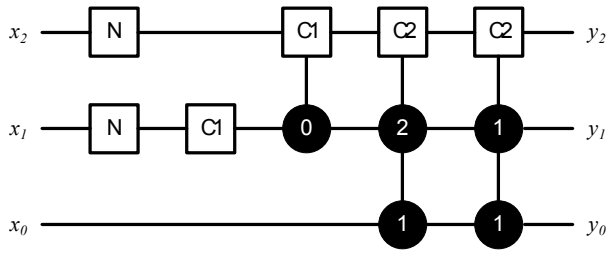

Figure 3.3: Ternary Multiplicative Inverse for Three Qudits

The last digit is now examined to complete the design of the four digit ternary multiplicative inverse with modular digit value encoding circuit. The pattern detected between the inputs and outputs at this point once again indicates that all x_3 inputs must use an "N" gate. A "C1" gate with a control input from y_1 of 0 is also necessary. Due to an inability to find functionally complete sets of inputs that see the same changes for outputs, four controlled "C2" gates with multiple control inputs are necessary to complete the design. There are two "C2" gates with two control inputs, one with control inputs y_2 of 2 and y_1 of 1, the other with control inputs y_2 of 0 and y_1 of 2. There are two "C2" gates with three control inputs, one with control inputs y_2 of 2, inputs y_1 of 2, and y_0 of 1,

the other with control inputs y_2 of 1, y_1 of 1, and y_0 of 1. These gates complete the design of the circuit. The four digit ternary multiplicative inverse with modular digit value encoding circuit design is revealed in Figure 3.4.

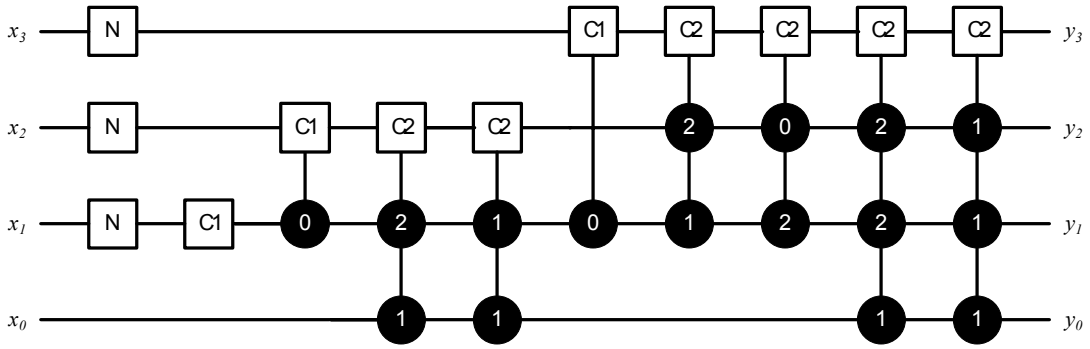

Figure 3.4: Final Multiplicative Inverse Circuit Design for Four Digit Ternary with Modular Digit Value Encoding

The use of the inheritance principle should be noted in the final design in Figure 3.4 of the multiplicative inverse circuit. This is not the only ternary circuit that was designed. The multiplicative inverse circuit corresponding to the digit natural digit set is shown in Figure 3.4.

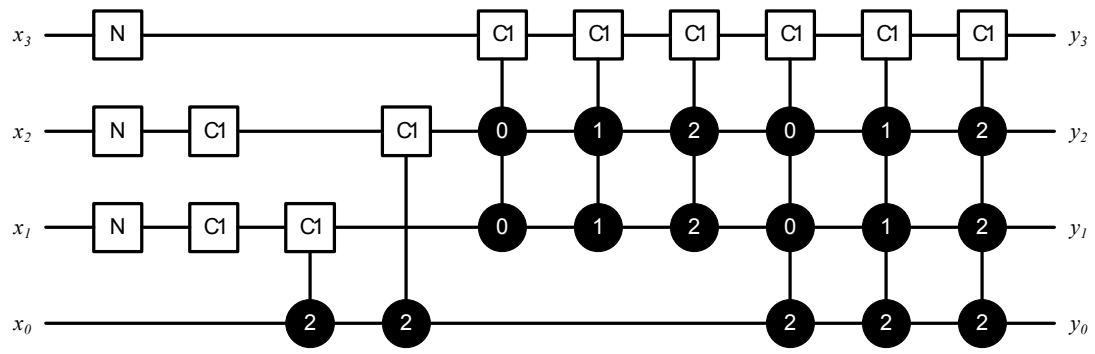

Figure 3.5: Multiplicative Inverse Circuit Design for Four Digit Ternary Logic

One other digit encoding was used in the design of ternary multiplicative inverse circuits. The encoding, $\{0, 1, 2\} \rightarrow \{-1, 0, 1\}$, is known as the balanced encoding. In Figure 3.6, it should be observed that the balanced encoding based circuit requires more gates to realize the multiplicative inverse operation than the preceding two encodings.

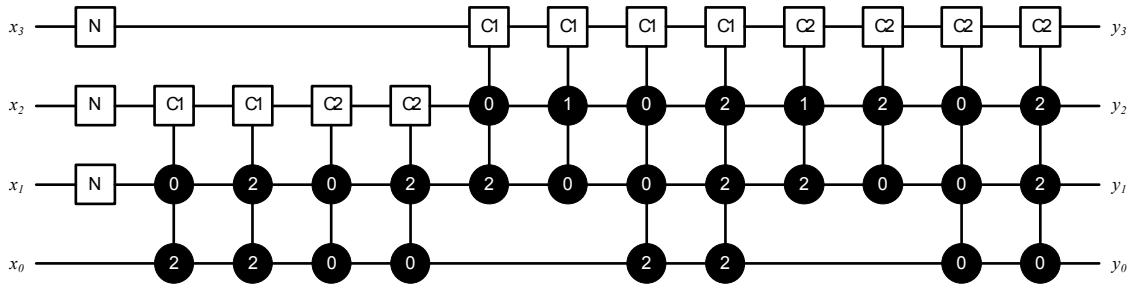

Figure 3.6: Multiplicative Inverse Circuit for Four Digit Ternary Logic with Balanced Encoding

The investigation into unary arithmetic operations in the area of ternary quantum logic was not confined to the multiplicative inverse [3]. Circuits were designed for the three digit discrete logarithm. One of the discrete logarithm circuits was designed manually and can be found in Figure 3.7.

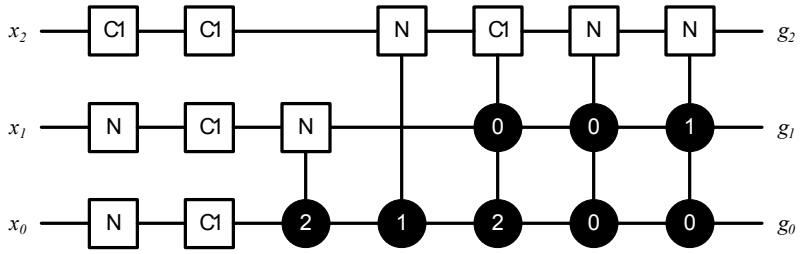

Figure 3.7: Discrete Logarithm for Three Digit Ternary Logic

Another discrete logarithm circuit design was created using an automated method [22]. The circuit design generated by the automated method is labeled as Figure 3.8 below.

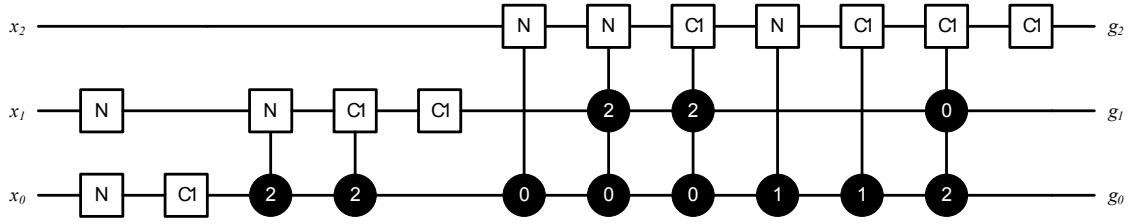

Figure 3.8: Discrete Logarithm for Three Digit Ternary Logic Designed with Automated Method

The automated method uses more gates to realize the circuit than the manually designed circuit. This is an interesting result that will assist those working with automated circuit synthesis in improvement of their process. The last ternary circuit designed is a three digit exponentiation circuit. The reversibility of quantum logic can be noted in the exponentiation circuit in Figure 3.9 is the reversal of the discrete logarithm.

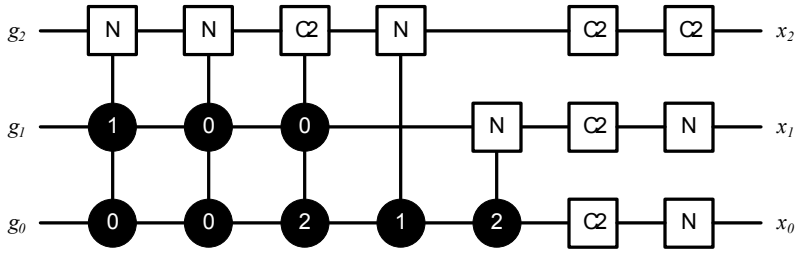

Figure 3.9: Exponentiation for Three Digit Ternary Logic

It should be noticed that the "C1" gates in the discrete logarithm circuit in Figure 3.7 must be exchanged for "C2" gates in the reversal circuit to correctly realize the exponentiation circuit in Figure 3.9 since the matrix representing the "C1" gate realizes the Identity matrix when multiplied by the matrix representing the "C2" gate.

3.3 Quintary Quantum Logic Universal Gate Set

Universal gate sets are a small collection of gates whose operations form a functionally complete set over the algebra used to model the circuit function. Hence, a universal gate set can be used to construct any possible logic circuit. In accordance with the standard notation in mathematical combinatorics, the set of all permutation gates is

denoted as Π with elements that are individual permutation gates denoted as π_{ijklm} with the *ijklm* subscript indicating the permutation the gate performs in the mathematical notation $\begin{pmatrix} 0\,1\,2\,3\,4 \\ i\;j\;k\;l\;m \end{pmatrix}$. Observations about the nature of base five permutations allow for the development of a technique to determine whether or not universal gates sets are achieved by a set of gates.

3.3.1 Quintary Quantum Logic Gate Characteristics

The total number of possible permutation gates over radix 5 is 5! = 120. This number of gates is somewhat unruly when contemplating the design of quintary logic for unary arithmetic operations. Finding a smaller universal gate set is necessary to make circuit design a more realistic endeavor.

One particular permutation gate that has characteristics that allow for the development of many other permutation gates is π_{43210}. This permutation gate allows any permutation to be reversed. This reversion means that only half, or sixty, of the total permutations must be found to have a universal gate set. As an example, the matrix π_{43210} is calculated using the equation $\pi_{43210} = |i\rangle\langle 0| + |j\rangle\langle 1| + |k\rangle\langle 2| + |l\rangle\langle 3| + |m\rangle\langle 4|$.

$$\pi_{43210} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}[1\,0\,0\,0\,0] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}[0\,1\,0\,0\,0] + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}[0\,0\,1\,0\,0] + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}[0\,0\,0\,1\,0] + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}[0\,0\,0\,0\,1]$$

$$\pi_{43210} = \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&1 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&1 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 1&0&0&0&0 \end{bmatrix}$$

Figure 3.5 demonstrates the representation utilized in circuit design diagrams in this thesis for π_{43210}.

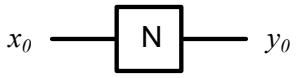

Figure 3.10: π_{43210} Gate Representation

Another permutation gate that has characteristics seen in many permutations is Π_{01243}. This permutation gate is seen frequently in the ordered permutation list. The frequency with which this gate is seen means that only half of the previous sixty gates need to be found, or thirty gates. It should also be noted that permutation gates with a similar permutation structure, for example: π_{01243} or π_{10234}, are seen with equivalent frequency. This allows more varied universal gate sets to be found. These matrices are also created by the previously described equation. As an example,

$$\pi_{01243} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} [1\,0\,0\,0\,0] + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} [0\,1\,0\,0\,0] \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} [0\,0\,1\,0\,0] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} [0\,0\,0\,1\,0] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} [0\,0\,0\,0\,1]$$

$$\pi_{01243} = \begin{bmatrix} 1&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&1&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&1 \\ 0&0&0&1&0 \end{bmatrix}$$

Figure 3.6 demonstrates the representation utilized in circuit design diagrams in this thesis for π_{01243}.

Figure 3.11: π_{01243} Gate Representation

Yet another permutation that has characteristics that can be used to determine the remaining permutations necessary for a universal gate set is π_{01432}. This permutation is seen slightly less frequently than the previous two gates. This gate is observed in two thirds of the remaining permutations, leaving only ten permutations to be found. Matrices of this type are also created by the equation used for the previous examples.

$$\pi_{01432} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} [10000] + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} [01000] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} [00100] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} [00010] + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} [00001]$$

$$\pi_{01432} = \begin{bmatrix} 1&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&1 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&1 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \end{bmatrix}$$

The permutation determined by this matrix calculation would appear similarly to Figure 3.5 and Figure 3.6.

The few remaining permutations may be found with different permutation gates. For the purposes of this example, π_{03214} will be used to find the final ten missing permutations. This final matrix can be created as well.

$$\pi_{03214} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} [10000] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} [01000] + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} [00100] + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} [00010] + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} [00001]$$

$$\pi_{03214} = \begin{bmatrix} 1&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&1 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0 \\ 0&0&0&1&0 \\ 0&0&1&0&0 \\ 0&1&0&0&0 \\ 0&0&0&0&1 \end{bmatrix}$$

Exhaustively producing all four gate quintary universal gate sets from the set of fifteen basic quintary gates discussed in the next section is a complex problem with 15 choose 4, approximately 1365, possible sets that must be verified with factorial verification time for each. This is due to the permutation necessary to prove the validity or invalidity of a universal gate set. A process was developed to search a set of five gates

for four gate quintary universal gate sets. By reducing the number of gates used to perform the verification procedure, the complexity of the problem is greatly reduced.

3.3.2 Quintary Quantum Logic Universal Gate Set Verification

The universal gate set is defined as a set of gates that when applied together cyclically generate every permutation gate that is not part of the gate set. An array containing five values can be easily manipulated in functions to represent the changes performed by using a particular gate. Each permutation must be checked in a list to determine whether or not every permutation is achieved. A rank function is used to check the permutations, the algorithm for which has been published in [18]. The manner in which the rank function is implemented is relatively simple. The rank function uses a formula that realizes an eventual result through recursion. The permutation array, number of elements left in the permutation array, and the last element that was removed from the permutation array are used in this recursive portion of the algorithm. If it is not the first time that the function has been called, the most recently removed element is used to make the remaining elements compatible with the new permutation array. This is achieved by subtracting one from each element that is greater in value than the most recently removed element. At this point the recursion is introduced with a return statement for when the array still has elements and a return statement for the case when there is only one element remaining. In the return statement for multiple elements, the formula requires that the first element in the permutation array minus one is multiplied by the factorial of the number of elements left in the array minus one. In the return statement, the previous

calculation is added to the next instance of the function that is called reflecting the removal of the first element. This produces a value that reflects the position of the current permutation in relation to the rest of the permutations in an ordered permutation list. For example, if the permutation created is {1, 2, 0, 4, 3}, then the permutation sent to the rank function is {2, 3, 1, 5, 4}, the rank function would be calculate the position by the following equation.

$$\text{rank position} = [(2-1) \times (4!)] + \text{rank}\{2, 1, 4, 3\}$$
$$[(2-1) \times (3!)] + \text{rank}\{1, 3, 2\}$$
$$[(1-1) \times (2!)] + \text{rank}\{2, 1\}$$
$$[(2-1) \times (1!)] + \text{rank}\{1\}$$
$$0$$

$$\text{rank position} = [1 \times 24] + [1 \times 6] + [0 \times 2] + [1 \times 1] + 0 = 24 + 6 + 0 + 1 + 0 = 31$$

This rank calculation is used with a list of possible permutations to determine which permutations have been observed and which have yet to be achieved.

The assumption that all gates in a universal gate set should be applied an equal number of times to achieve all possible permutations is used in this case. The order in which the gates were applied should be varied. Applying the gates in different orders allows more permutations to be achieved. This behavior can be observed with a simple example. When the permutation gate, known as π_{01243}, is modified by another permutation gate, π_{01432}, the result is the permutation gate π_{01342}. When the order of these two gates is reversed, the resulting permutation gate is π_{01423}. This observation indicates

that varying the order of the gates may allow a universal gate set with fewer gates to be found.

It was determined that a quintary universal gate set can be made with five gates using the simple order variation of rotating through the set of gates. By permuting the order of the permutation gates, a slightly smaller universal gate set is realized. To perform this test, a table of the twenty-four permutations of four numbers to set the order in which gates will be called is used. A table of the five combinations of four gates that consist of a universal set is also used. These tables are used together as a fast way to test for universal gate sets. This allows the simulation of the combination of the gates in a particular order while denoting which permutations are seen using the previously discussed ranking procedure and permutation list. It cycles through the possible order of the gates, then checks the permutation list for permutations that are not included. If any permutation is not produced by the four gates, it is not a universal gate set. It is then cycled through the other possible groups of four gates to check for universal gate sets, repeating the process.

There is one other way that this method ensures that a set of gates is a universal gate set. If all permutations aren't realized in the previously described steps, the process is started again with the same gates utilizing a permutation that has already been generated as the starting permutation. It does this by finding the index of a permutation that has already been generated by the set of gates. When the index is found, the starting permutation is taken from a five digit permutation table. If using the permutations previously generated by the gates does not help generate the missing permutations, the

gates do not form a universal gate set. An outline of the operation of this method is provided by the flowchart in Figure 3.12.

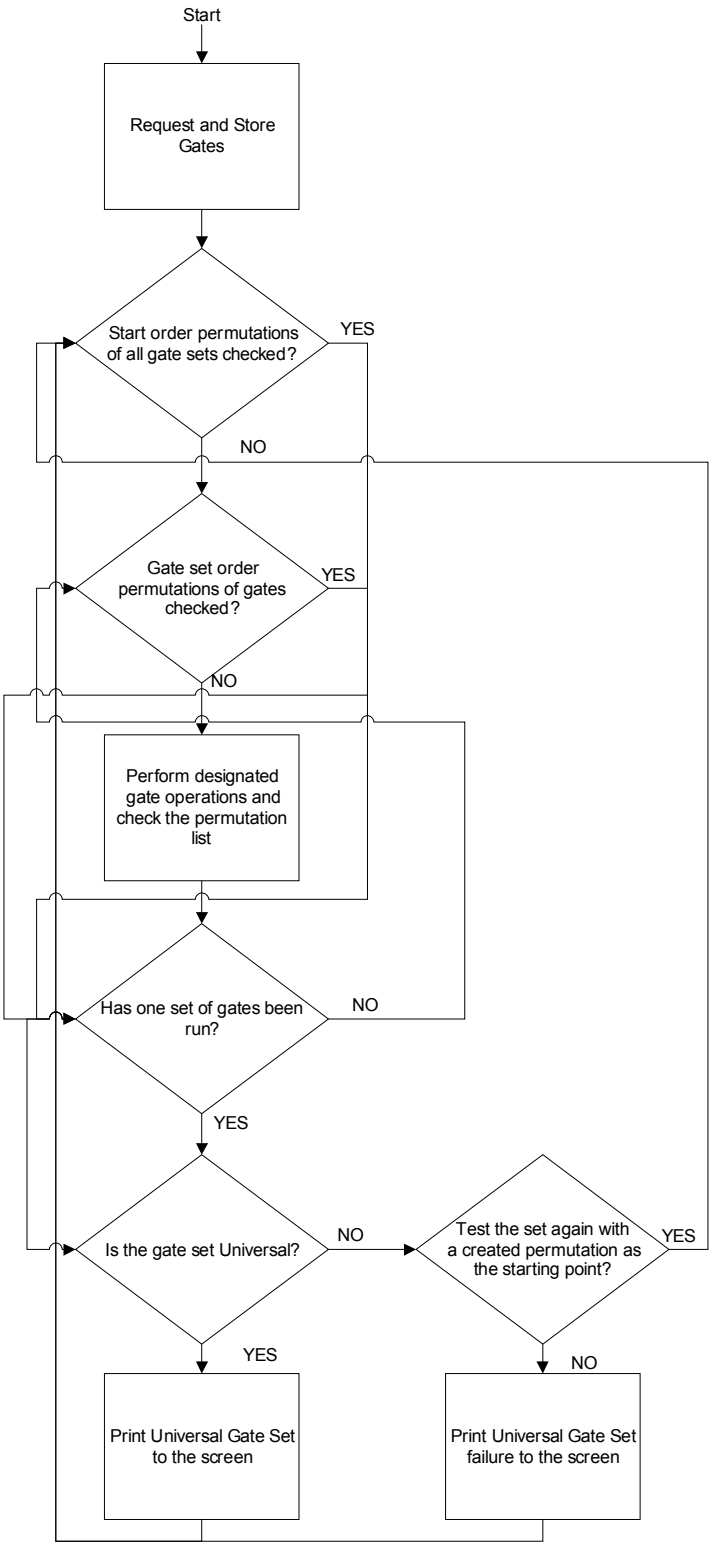

Figure 3.12: Flowchart of Universal Gate Set Verification Method

The method developed utilizes a letter and number classification for the testing of the gate sets. In Table 3.1, the permutation gates in the set of permutation gates available for use with this method are detailed.

Table 3.1: Quintary Quantum Logic Permutation Gates

Name in Program	**Permutation** ***(i, j, k, l, m)***	**Permutation Symbol**
N	(4, 3, 2, 1, 0)	π_{43210}
C1	(1, 2, 3, 4, 0)	π_{12340}
C2	(2, 3, 4, 0, 1)	π_{23401}
C3	(3, 4, 0, 1, 2)	π_{34012}
C4	(4, 0, 1, 2, 3)	π_{40123}
S1	(1, 0, 2, 3, 4)	π_{10234}
S2	(0, 2, 1, 3, 4)	π_{02134}
S3	(0, 1, 3, 2, 4)	π_{01324}
S4	(0, 1, 2, 4, 3)	π_{01243}
L1	(2, 1, 0, 3, 4)	π_{21034}
L2	(0, 3, 2, 1, 4)	π_{03214}
L3	(0, 1, 4, 3, 2)	π_{01432}
E1	(3, 1, 2, 0, 4)	π_{31204}
E2	(0, 4, 2, 3, 1)	π_{04231}
E3	(4, 1, 2, 3, 0)	π_{41230}

Table 3.1 should be used to determine which gates are chosen for verifying sets of four permutation gates forming universal gate sets from sets of five permutation gates. Permutation gates in Table 3.1 with the same starting character perform similar permutations.

3.4 Quintary Quantum Circuit Design

Using three-digit quintary radix tables for the multiplicative inverse and the basic permutation gates established for use in finding the quintary quantum logic universal gate set, quintary quantum logic circuits can be designed. The first circuit design did not utilize the negative modulo equivalent encoding feature of the multiplicative inverse table generation technique. The circuit design in Figure 3.13 was developed with the quintary quantum logic gate set {N, C1, S3, L1}

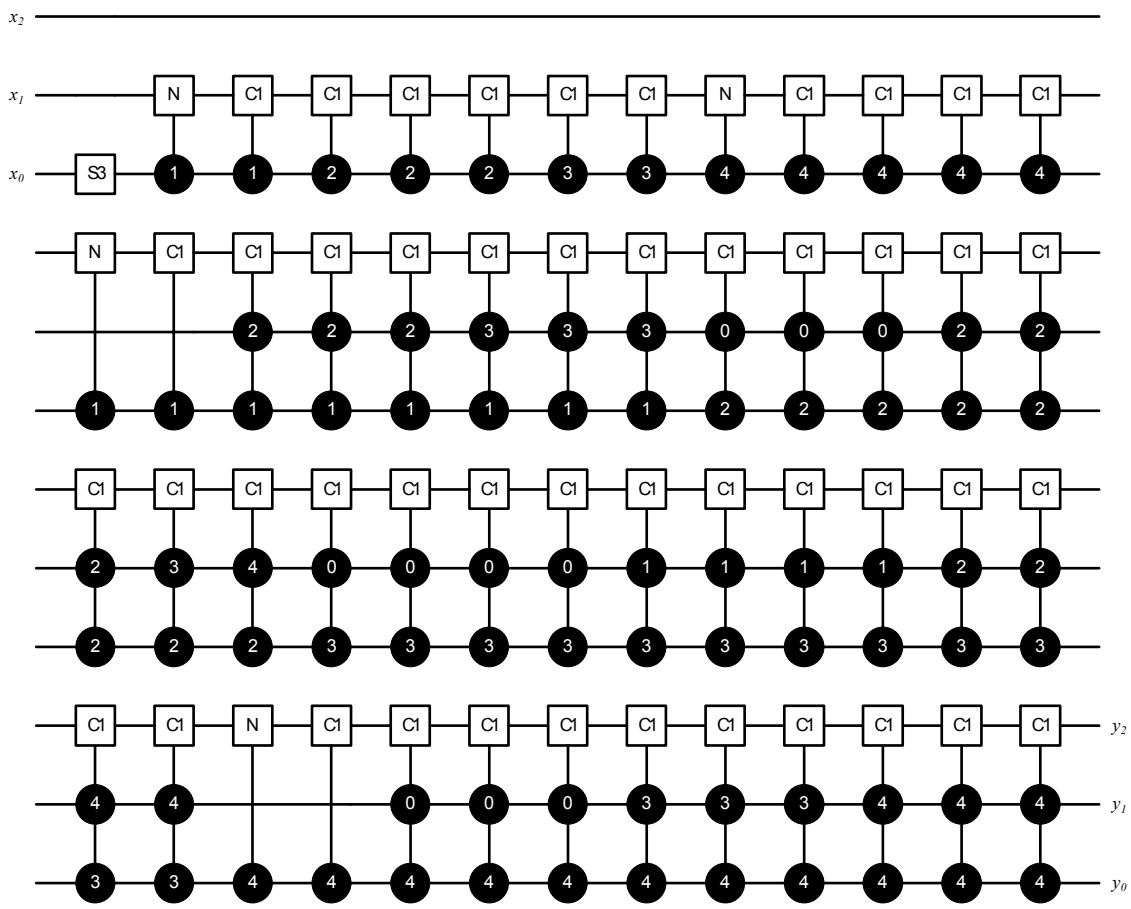

Figure 3.13: Multiplicative Inverse Circuit Modulo 5^3

In Figure 3.13, the most frequently occurring quintary quantum logic gate type is of a rotational type. The gates denoted with "C" for the letter in the name use matrices that can be described as "adding" the numeral in the name and using modulo of the base to find the resulting value. It should be observed that of the fifty-two gates used in this design, only five are not "C" gates.

Using the table generation process with the negative modulo equivalent option exercised, another three digit radix five table was created for designing a second circuit.

This circuit follows the same design concepts used to create the previous multiplicative inverse quintary quantum logic circuit.

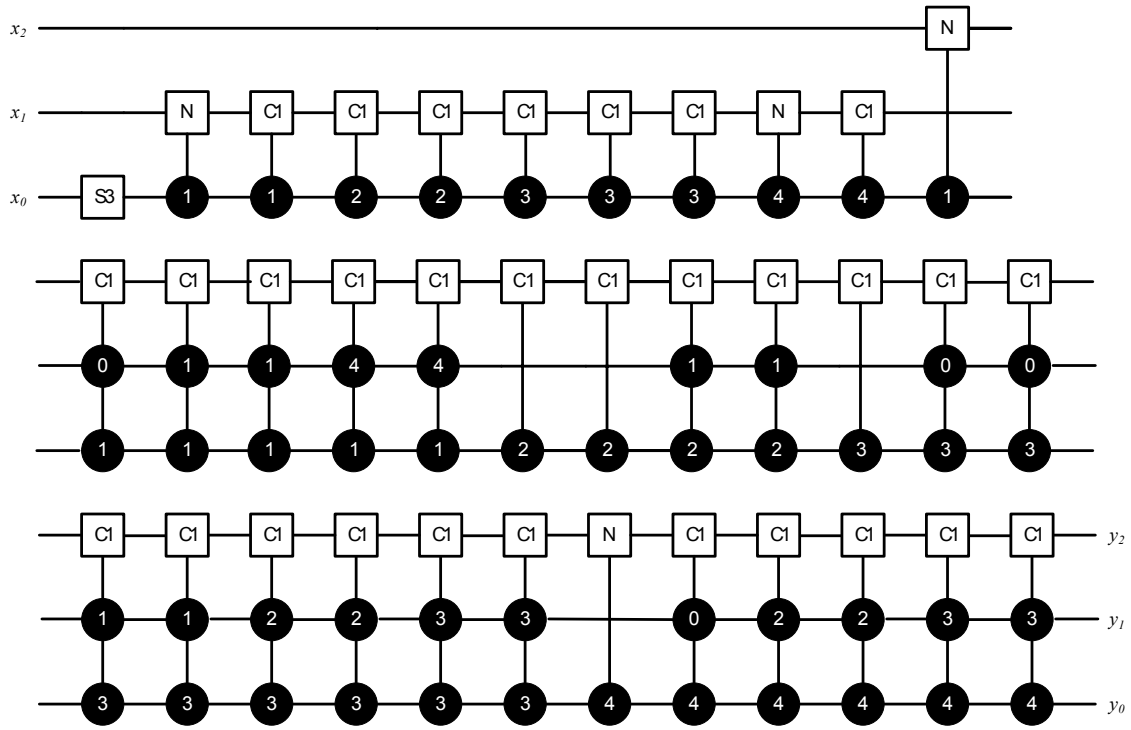

Figure 3.14: Multiplicative Inverse Circuit Modulo 5^3 with Negative Modulo Equivalent Encoding

In Figure 3.14, as in Figure 3.13, the most frequently occurring quintary quantum logic gate type is the "C" gate. In the case of the quintary negative modulo equivalent encoded multiplicative inverse circuit design, five of the thirty-five gates are not "C" type gates. It should also be noted that the negative modulo equivalent encoded circuit uses fewer gates to realize the resulting output than the normally encoded circuit.

Chapter 4

CIRCUIT VERIFICATION

4.1 Design Validation Methodology

Two distinct methods of quantum logic circuit verification were used in this research. The first verification method was simulation of the proposed quantum logic circuits. The second verification method was a quantum circuit verification tool utilizing functional equivalence checking.

4.1.1 Design Simulation

The designs for circuits were verified using simulation of the circuit designs. Each input was simulated with relation to the changes that the gates in the quantum logic circuit would achieve. The outputs were then compared to the known correct output resulting digits. The design is correct when the output digits are the same for circuit and the known values.

The simulation method may be employed with any user defined radix, while there are some limitations to other verification methods. It is necessary to use this method to verify the quintary quantum logic circuit designs due to the current constraints on other quantum logic verification methods.

4.1.2 QMDD-Based Verification

Quantum Multiple-valued Decision Diagrams, or QMDDs as they are commonly known, are the only decision diagram structure that can be used with qudit quantum logic [13]. QMDDs are directed acyclic graphs that must adhere to the following properties:

1) A single terminal vertex with an associated value of 1. Terminal vertex means that it has no outgoing edges.
2) Some number of nonterminal vertices, each has a label determined by p^2-valued selection variable. The nonterminal vertices have p^2 outgoing edges, denoted as $e_0, e_1, \ldots, e_{p^2-1}$.
3) There is one start vertex that has only one incoming edge that has no source vertex itself.
4) Each of the edges in the QMDD have a complex-valued weight associate with it. This includes the edge to the start vertex with no source.
5) Selection variables are ordered and satisfy two rules:
    a. On each of the paths from start vertex to terminal vertex, each selection variable will appear only once.
    b. A nonterminal vertex, labeled as x_i, has an edge pointing to another nonterminal vertex, labeled as $x_j, j < i$ or the terminal vertex. Thus, x_k, for the largest value of k, labels the start vertex, while x_0 is closest to the terminal vertex.

6) None of the nonterminal vertices is redundant, meaning that no nonterminal vertex may have all p^2 outgoing edges pointing to the same vertex with the same weights.
7) The outgoing edges of the nonterminal vertices are normalized so that on any edge the largest weight on any edge is 1.
8) Nonterminal vertices must be unique, meaning that no two nonterminal vertices labeled as the same x_i have the same outgoing edges, defined by the same weight and destination.

There is an extreme case of a QMDD with only a start vertex and no nonterminal vertices; however, due to the dependency of this case on a constant valued matrix, it was not necessary to examine this particular type of QMDD for this research [20].

In the case of ternary quantum logic, the circuit designs are verified using a system that parses two circuits into a QMDD. Due to the canonical nature of QMDDs, when two different quantum logic circuit cascades are parsed into QMDDs with same structure, the circuits are functionally equivalent [22].

4.2 Design Results

The design of the quintary multiplicative inverse circuit was accomplished manually using a multiplicative inverse table generated by the method described previously and a universal gate set determined by the technique described. The property of inheritance was exploited in this design. This allowed each successive higher-order digit to be designed for without re-evaluating the previous design. These designs serve to

demonstrate the ability to design quintary quantum logic unary arithmetic operations exploiting the inheritance principle.

Chapter 5

CONCLUSIONS AND FUTURE RESEARCH

5.1 Conclusions

This research of the design of multiple-valued quantum logic arithmetic circuits, led to the development of automated varied radix and digit set multiplicative inverse table is a step forward in the capability of designing more quantum logic circuits for future use. The unary arithmetic operations explored in this research exploit a major characteristic inherent in quantum logic, bijection. The use of bijective operations and residue arithmetic allow the development of concise design solutions.

The ability to find and prove a four permutation gate universal gate set in quintary quantum logic is a significant development. The property of inheritance and small universal gate sets are exploited to great effect in the design of the quintary quantum logic multiplicative inverse circuit.

5.2 Future Quantum Logic Development

While this research has yielded interesting results, there are many more areas to pursue in this topic. Only one unary arithmetic function was studied more extensively by examination in the realm of quintary logic. Other unary arithmetic operations should be

investigated in different logic bases. This would allow for a determination of the most efficient multi-valued logic base for performing these arithmetic operations.

An automated synthesis tool for creating quantum logic circuits is a logical step in the progress of quantum logic research. Such tools exist for work with classical logic, creating an automated synthesis tool for quantum logic would help to increase the number of circuits that are designed. Automated synthesis tools can also be used to help increase the efficiency of circuit designs.

5.3 Future Universal Gate Set Development

Universal gate sets are not confined to quintary quantum logic. Any quantum logic base with permutation gates will have universal gate sets. Finding those universal gate sets is a problem that becomes more complex as the value of the base increases. To examine the universal gate set problem for sectary quantum logic, quantum logic with a base of seven, the total number of permutations expands to 7!, or 5,040 total permutations. A consistent and efficient method for finding universal gate sets regardless of base is an area for further development and research.

The universal gate sets should also, in future, be expanded to include quantum logic gates that are not defined as permutation gates. An example of one of these gates is the swap gate [7]. The controlled swap gate is not defined by a permutation matrix, but instead swaps the assigned qudits.

REFERENCES

[1] D. W. Matula, A, Fit-Florea, and M. A. Thornton, Lookup Table Structures for Multiplicative Inverses Modulo 2^k, Proceedings of the *IEEE Symposium on Computer Arithmetic*, June 27-29, 2005, pp. 130-135.

[2] L. Li, A. Fit-Florea, M. A. Thornton, and D. W. Matula, Performance Evaluation of a Novel Table Lookup Method and Architecture for Integer Functions, Proceedings of the *IEEE International Conference on Application-specific Systems, Architectures, and Processors*, September 11-13, 2006, pp. 99-104.

[3] D. W. Matula, Discrete Log Number Systems, *SMU Internal Report*, 2007.

[4] D. Goodman, M. A. Thornton, D. Y. Feinstein, and D. M. Miller, Quantum Logic Circuit Simulation Based on the QMDD Data Structure, Proceedings of the *Workshop on Applications of the Reed-Muller Expansion in Circuit Design and Representations and Methodology of Future Computing Technology*, May 16, 2007, pp. 99-105.

[5] D. Gottesman, Fault-Tolerant Quantum Computation with Higher-Dimensional Systems, 1998, arXiv:quant-ph/9802007.

[6] D. M. Miller, D. Maslov, and G. Dueck, Synthesis of Quantum Multiple-Valued Circuits, *Journal of Multiple-Valued Logic and Soft Computing*, vol. 12, no. 5-6, 2006, pp. 431-450.

[7] G. Yang, X. Song, and M. Perkowski, Realizing Ternary Quantum Switching Networks without Ancilla Bits, *J. Phys. A: Math. Gen.*, vol. 38, no. 44, November 2005, arXiv:quant-ph/0509192v1.

[8] A. DeVos, B. Raa, and L. Storme, Generating the Group of Reversible Logic Gates, *J. Phys. A.: Math. Gen*., vol. 35, no. 33, August 2002, pp. 7063-7078.

[9] J. Daboul, X. Wang, and B. C. Sanders, Quantum Gates on Hybrid Qudits, *J. Phys. A.: Math. Gen*., vol. 36, no. 14, 2003, pp. 2525-2536, arXiv:quant-ph/0211185v1.

[10] A. Muthukrishnan and C. R. Stroud Jr., Multi-Valued Logic Gates for Quantum Computation, *Phys. Rev. A* preprint, June 2000, arXiv:quant-ph/0002033v2.

[11] S. Szabó and R. I. Tanaka, **Residue Arithmetic and Its Application to Computer Technology**, McGraw-Hill Book Co., 1967.

[12] G. H. Hardy and E. M. Wright, **An Introduction to the Theory of Numbers**, Oxford Science Publications, 5th ed., 1979.

[13] D. M. Miller and M. A. Thornton, QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits, Proceedings of the *IEEE Int. Symp. on Multiple-Valued Logic*, May 2006, CD, 6 pp.

[14] Dan C. Marinescu and Gabriela M. Marinescu, **Approaching Quantum Computing**, Pearson Prentice Hall, 2005.

[15] Micheal A. Nielson and Isaac L. Chuang, **Quantum Computation and Quantum Information**, Cambridge University Press, 2000.

[16] Mika Hirvensalo, **Quantum Computing**, Springer-Verlag, 2nd Edition, 2004.

[17] Calvin T. Long, **Elementary Introduction to Number Theory**, D. C. Heath and Company, 1965.

[18] William Judson LeVeque, **Topics in Number Theory**, Volume I, Addison-Wesley Publishing Company, Inc., 1956.

[19] M. A. Thornton, L. Spenner, D. W. Matula, and D. M. Miller, Quantum Logic Implementation of Unary Arithmetic Operators, IEEE International Symposium on Multiple-Valued Logic (ISMVL), May 22-23, 2008, (to appear).

[20] D. M. Miller and M. A. Thornton, **Multiple-Valued Logic Concepts and Representations**, Morgan & Claypool Publishers, San Rafael, California, ISBN 10-1598291904 (hardcopy), 10-1598291912 (eBook), January 2008.

[21] B. P. Lanyon, M. Barbieri, M. P. Almeida, T. Jennewein, T. C. Ralph, K. J. Resch, G. J. Pryde, J. L. O'Brien, A. Gilchrist & A. G. White, Quantum computing using shortcuts through higher dimensions, 2 April 2008, arXiv:0804.0272v1[quant-ph].

[22] D. M. Miller, personal communication, October 2007.

[23] D. Deutsch, Quantum Theory, the Turing-Church Principle and the Universal Quantum Computer, *Proc. R. Soc. London A*, 400:97-117, 1985.

[24] D. Deutsch and R. Jozsa, Rapid Solution of Problems by Quantum Computations, *Proc. R. Soc. London A*, 439:553-558, 1992.

[25] P. W. Shor, Algorithms for Quantum Computation: Discrete Logarithm and Factoring, *Proc. 35 Annual Symp. on Foundations of Computer Science*, 124-134, IEEE Press, 1994.