5-26-2017

# A Method for Optimizing q-Valued Quantum Multiplexers

Justin T. Morgan
*Portland State University*

# A Method for Optimizing q-Valued Quantum Multiplexers

BY

**Justin Morgan**

*An undergraduate honors thesis submitted in partial fulfillment of the requirements for the degree of bachelor of science*

*in*

*University Honors*

*and*

*Computer Engineering*

Thesis Adviser
Marek Perkowski

Portland State University
August 17, 2017

# Method for Optimizing q-Valued Quantum Multiplexers

Justin Morgan, Tashin Saffat, Marek Perkowski

April 2017

**Abstract**

This paper presents a new approach to optimize arbitrary quantum circuits based on multi-valued Quantum Multiplexers. We define standard, and fixed polarity forms for binary valued quantum multiplexers that are analogous to the disjoint sum of products and Fixed Polarity Reed-Muller Forms for classical logic functions. Then, the method is extended to logic with an arbitrary radix. The algorithm produced requires $O(mq^m)$ butterfly transformations, where $m$ is the number of control variables, and $q$ is the radix of logic. A software script is then added and described to facilitate in the computation of larger and more complex quantum multiplexers.

## 1 Introduction

Quantum multiplexers are complex gates which, based on a given combination of values from several control variables, will select a target function to act upon a set of target variables. Initially however, Quantum multiplexers are created in what is called suboptimal form, as their control functions are in disjoint sum of products form. This logic arrangement can be optimized to reduce the amount of controls needed to produce a logically equivalent form. In this treatment, we assume that these control variables have only the basis states, so a multiplexer with $m$ $q$-valued control variables can be completely characterized by specifying which target function is selected for each of the $q$ possible states of each variable. For $m$ variables, you have $q^m$ states. Throughout this paper $m$ denotes the number of control variables. A basis state is also called the "logical" state of the qubit for binary valued case, and the basis quantum control states, ie. the values of $|1\rangle$ and $|0\rangle$, is revealed by the label of one of the two basis states. It is assumed that the initial state of the qubit is one of these basis states. Qubits can contain more complex quantum information such as superposed and entangled states. However for purposes here, labeling the control variables as $|1\rangle$, $|0\rangle$ or $|2\rangle$ for qudits of a ternary function is sufficient.

In Figure 1, a ternary quantum multiplexer with two control variables is shown. In the ternary case parameters $m = 2$ and $q = 3$, we have $q^m$ states, or $3^2 = 9$ possible states. In the paper which the current treatment is based on, [2] which formed the basis for this current paper, Tashin Saffat developed a method for implementing binary quantum multiplexers, where the control variables $C_1$ and $C_2$ are binary qubits, whereas in the implementation for

this paper, the method is extended where those control variables are ternary valued, and in general, q-valued control variables. This paper is an extension on the work Saffat has done, by adding new segments detailing the logical methods utilized to optimize these multiplexers, and the addition of several diagrams and importantly, software that can be used to compute increasingly complex versions of these quantum multiplexers via numerical analysis tools.

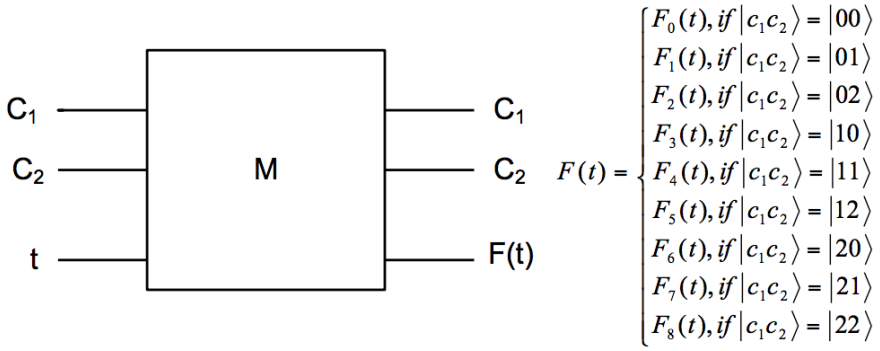
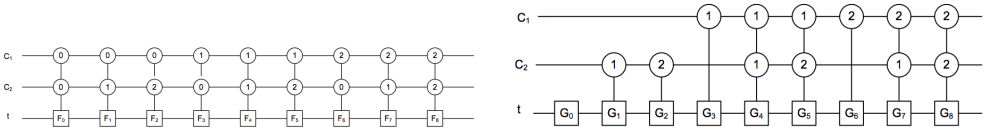



Figure 1: A Ternary quantum multiplexer

In Figure 2, a circuit for a ternary quantum multiplexer is shown, with two control variables and one target variable. Each of these target functions labeled $f_0$ through $f_8$ for each combination of inputs. These inputs, since they are ternary valued instead of the classical binary value, will be $|00\rangle, |01\rangle, |02\rangle, |10\rangle, |11\rangle, |12\rangle, |20\rangle, |21\rangle$ ,and $|22\rangle$. These combinations are also seen in Figure 1. Each combination in this multiplexer is represented in ascending order, from $F_0$ through $F_8$. Figure 3 shows the multiplexer in Figure 1 converted to positive polarity multi-valued general fixed polarity (MV GFP) form. In this paper, the method for the fixed polarity Reed-Muller (FPRM) form based on butterfly transformations is extended to arbitrary quantum multiplexers. Although in the previous research pertaining to binary quantum multiplexers, the binary GFP forms are direct generalizations of binary Fixed Polarity Polynomial Expressions (FPPE), the MV GFP forms presented in this paper do not directly generalize FPPE. These MV GFP forms exist for quantum logic with a non prime radix, radix meaning number base. [3].



(a) A Ternary quantum multiplexer (b) A Ternary Multiplexer in MV GFP form

Figure 2: A representation of DSOP form and MV GFP form Quantum Multiplexers

The objective of achieving this form is to both reduce the number of controls and the cost

of the target functions by calculating quantum multiplexers for several different polarities. The method developed in this paper is to calculate all of the initial target functions and choose the optimal polarity. Then, we must calculate the function vector from the original function's vector. To achieve this function vector, we analyze the quantum multiplexer described and shown on the left in Figure 2. This results in a system of equations that can be used to solve for vector G in terms of vector F. Figure 5 details this transformation with the complete set of equations. Though the method explicitly mentions q-valued quantum multiplexers, several examples will be used from binary valued quantum multiplexers. These binary examples with simpler systems of equations and with boolean logic provide a simplified setting for understanding the underlying mathematical properties of this method.

## 1.1 Reed-Muller Logic

Reed-Muller logic is well known for its fundamental properties in synthesizing binary, reversible and permutative quantum circuits. [1] As such, it is useful for improving the synthesis and algorithms of multi valued quantum circuits. Reed-Muller forms are canonical representations of functions (circuits) using EXORs of ANDs of arbitrary literals, where a literal is a variable that is complemented or not complemented. The AND-EXOR implementation is used for binary valued logic. Even through classic CMOS design, the AND-EXOR implementation of logic functions for arithmetic and similar applications is far more economical then AND-OR, generally requiring fewer gates and connections. These observations strengthen the case for utilizing Reed-Muller logic in this method to optimize quantum multiplexers. These representations allow the inexpensive construction of large functions. The implementation of AND-EXOR will be demonstrated with several examples, using binary valued quantum multiplexers, in addition to ternary representations. Ternary logic uses Galois field 3 logic instead of the AND-EXOR will also be presented. AND-EXOR construction seen in binary logic with Reed-Muller represents a special case of the Galois Field logic, denoted as $GF_k$. Galois Field Logic could be viewed as a generalization of the boolean logic, because Galois Field mathematical operations are applicable to multi-valued logic over any finite field.

The concept of Positive polarity and Fixed polarity Reed-Muller is now introduced. PPRMs and FPRMs are generalizations of Reed-Muller canonical forms. PPRM is exactly the same as the original Reed-Muller and Zhegalkin forms. The name has been historically changed when several new forms were invented. In PPRM, every variable is in positive polarity, or in the case of binary logic, every variable is logic high and un-complemented. In FPRM every variable has a uniform complement or non-complement, but never both. A concrete example would be where a logical expression with only $A$ or $A'$, but never both at once in the same EXOR of ANDs exist. In order to generalize the FPRM method for classical logic to quantum multiplexers, standard and fixed polarity forms are defined for quantum multiplexers that are analogous to the disjoint sum of products and FPRM forms for classical logic functions. These are called Generalized Fixed Polarity forms, or GFP. For binary valued controls on a multiplexer, the fixed polarity will be defined as GFP and for ternary and other q-valued multiplexers, the fixed polarity form will be defined as Multi-Valued

Generalized Fixed Polarity, MV-GFP.

## 1.2 A Binary Approach

A binary example will be worked through to provide a clearer understanding of the implementation and purpose of Reed-Muller forms. Figure 3 illustrates a suboptimal quantum multiplexer in disjointed sum of products. As the Karnaugh map shows, there are four possible combinations and four possible target functions. Each target function is activated by one specific combination and no other, hence the designation of disjointed sum of product. Below are a series of equations that demonstrate the transformation of disjoint sum of products to a Fixed Reed-Muller polarity form for the polarity 11 in this quantum multiplexer.

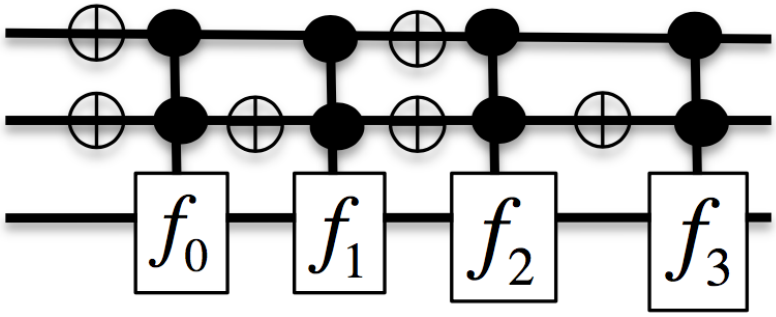



Figure 3: A quantum multiplexer constructed with doubly controlled NOT gates called also CCNOT, otherwise known as Toffoli gates.

Figure 1 shows a binary valued quantum multiplexer, constructed with controlled NOT (CNOT) gates, each of which enables a particular combination to act on a target variable. This multiplexer and its target functions are mapped with a Karnaugh map. This Karnaugh map is created based on the quantum multiplexer in a slightly different form, one that uses the CNOT gates but utilizes the notation of empty circles instead of filled ones to activate target functions. The figure begins with A and B asserted on the control lines. Immediately, the CNOT gate negates both such inputs. The notation for these multiplexers is described in Figure 4:

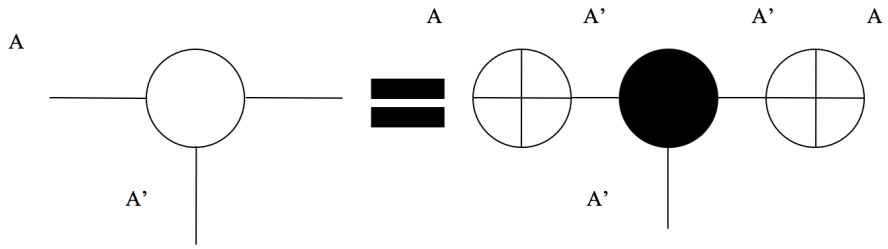
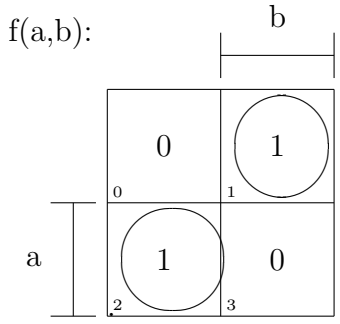



Figure 4: Notation description of activated targets and controlled NOT gates

The subfigure on the left in Figure 4 demonstrates a simple open circle. The subfigure on the right is the equivalent format when using CNOT gates. Replacing the notation of Figure 3 using the new open circle notation, allows the Karnaugh map to be better mapped and a fixed polarity form can be achieved. Working left to right, follow the control lines (Denoted A and B) through the CNOT gates and the target function lines. Every open circle means that the variable is negated before going to the target function, and then un-negated as it continues rightward (refer to Figure 4). The two control lines combine to act on the target function below it. Since we are using Reed-Muller form, this quantum multiplexer is constructed with quantum equivalents of XOR gates. These two control lines, therefore, are XOR'd together, to make $A \oplus B$. As we move through, the A control line isn't negated, so it continues to be $A'$. When it flows through the open circle notation for the next target function, it is then negated to $A$, and combines with B which is negated in this case, to make $A \oplus B$. A similar procedure occurs over the last two target functions; finally, each target function will have an associated logical function in the form of $A \oplus B$ with possible negations. Each of these four logical functions also maps to the four possible combinations with two variables. Since these are all XOR functions, recall that XOR is true when only one of the variables is positive. Thus, only the logical functions $A'B$ and $AB'$ are true. Creating the Karnaugh map for this multiplexer appears as:




The circling of the Karnaugh map results in a Sum of Products expression, which can then be formed into a logically equivalent expression in Reed-Muller Fixed polarity, for binary valued expressions. Such a mathematical calculation is done with the following set of

equations. The first equation is the quantum multiplexer from the Karnaugh map expressed in sum of products. Note the disjointed expressions, as each target function only corresponds to one particular combination of inputs. The following equations results from replacing the OR operator with the XOR operator.

$$F = A'B + AB'$$
$$F = A'B \oplus AB'$$
$$F = (1 \oplus A) \cdot B \oplus AB'$$
$$F = B \oplus AB \oplus AB'$$
$$F = B \oplus A(B \oplus B')$$
$$F = B \oplus A(1)$$
$$F = B \oplus A$$

The final equation shows the result of this transformation, this multiplexer is in polarity 11. Below we show the equations are that result from similar calculations, but for the remaining polarities. To summarize, the remaining polarities expressed in FPRM form are as follows:

$$F = A' \oplus B' \quad \text{(Polarity 00)}$$
$$F = B \oplus (A' \oplus 1) \quad \text{(Polarity 01)}$$
$$F = A \oplus (B' \oplus 1) \quad \text{(Polarity 10)}$$
$$F = A \oplus B \quad \text{(Polarity 11)}$$

These polarities are also results from the same Karnaugh map circling. Essentially, this Karnaugh map is expressed in multiple logically equivalent expressions of differing polarities, all in Fixed Polarity Reed-Muller (FPRM) form.

## 1.3 The Set of Equations

Now that the logical expressions for each polarity has been stated, the combinations of these polarities, and how they map to the target functions can be demonstrated , by forming a system of equations and an optimal representation of this circuit. First, begin by listing out the combinations of polarities, and which target function is activated by a particular combination as seen in Figure 3.

$$A'B' \cdot f_0 \oplus A'B \cdot f_1 \oplus AB' \cdot f_2 \oplus AB \cdot f_3 \quad (1)$$

This equation is a combination of all possible polarities, in literal form. Since this needs to be rewritten to fixed polarity and in Reed-Muller logic, the next few steps will convert our complemented variables and group them according to the target function they activate. This

process is called positive Davio expansion, and it will be applied to each variable that appears as a complement. Davio expansion is also sometimes referred to as Reed-Muller expansion, and it describes certain specific type of decomposition of a boolean function. Any variable that appears as negated will be expanded to a form where the variable is XOR'd with the value of 1. In addition to positive Davio expansion, there is negative Davio expansion, and Shannon expansion. Before continuing from equation (1), these expansions shall be formally defined.

Given an arbitrary logic function, $f(x_1, x_2, ..., x_n)$, where $f_0 = f(0, x_1, x_2, ..., x_n)$ and $f_1 = f(1, x_1, x_2, ...x_n)$, and $f_2 = f_0(x_1, x_2, ..., x_n) \oplus f_1(x_1, x_2, ..., x_n)$, these expansions are defined as follows:

$$f = f_0(x_1, x_2, ..., x_n) \oplus x_1[f_0(x_1, x_2, ..., x_n) \oplus f_1(x_1, x_2, ..., x_n)] \quad \text{(Positive Davio)}$$
$$f = f_1(x_1, x_2, ..., x_n) \oplus x_1'[f_0(x_1, x_2, ..., x_n) \oplus f_1(x_1, x_2, ..., x_n)] \quad \text{(Negative Davio)}$$
$$f = x_1' f_0(x_1, x_2, ..., x_n) \oplus x_1 f_1(x_1, x_2, ..., x_n) \quad \text{(Shannon Expansion)}$$

Now these expansions have been defined, the next step is to work through the problem presented in (1).

$$(1 \oplus A) \oplus (1 \oplus B) f_0 (1 \oplus A) \oplus B f_1 \oplus A \oplus (1 \oplus B) f_2 \oplus AB f_3 \tag{2}$$

$$(1 \oplus A \oplus B \oplus AB) \cdot f_0 \oplus (B \oplus AB) \cdot f_1 \oplus (A \oplus AB) \cdot f_2 \oplus AB \cdot f_3 \tag{3}$$

Equations 2 and 3 expand equation 1 by converting any complemented variable into the form $(1 \oplus A) or (1 \oplus B)$. The next step is to group them all by target function, by simplifying the equation through algebra.

$$f_0 \oplus A f_0 \oplus B f_0 \oplus AB f_0 \oplus B f_1 \oplus AB f_1 \oplus A f_2 \oplus AB f_2 \oplus AB f_3 \tag{4}$$

$$f_0 \oplus A(f_0 \oplus f_2) \oplus B(f_0 \oplus B f_1) \oplus AB(f_0 \oplus f_1 \oplus f_2 \oplus f_3) \tag{5}$$

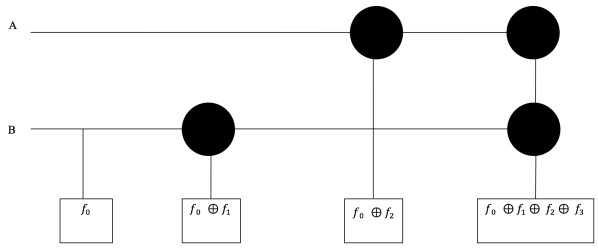
The final equation contains the variables and which target functions are activated by those variables being asserted positive (1). Note that equations (4) = (5), (5) is a simplification of (4) by combining like terms. Now, each variable combination has been represented. With no controls asserted, $f_0$ is activated. For control A, functions $f_0$, $f_2$ are activated. This continues for the remaining groups for B and AB. In the case of $A = B = 1$, all target functions will be activated, as described by equation (5). These groupings help form the system of equations that show a optimized multiplexer in Generalized Fixed Polarity form.

Take the equation set from equation 5 and it will correspond to the system of equations set up in Figure 5. The system of equations now has functions of $f$ equal to target functions of $g$. Then, solve for $g$ in terms of $f$. Start from the beginning; $f_0$ is equivalent to $g_0$. Since $f_0$ is equal to $g_0$, then $g_0$ is equal to $f_0$. Then, on the next equation, $f_1 = g1 * g0$, we substitute in $g_0$, which is now known to be equal to $f_0$. Every target function of $g$ that is solved in terms of $f$ can be then substituted in subsequent equations which allow the calculations to be complete. These complete calculations are seen on the right side of the arrows in Figure

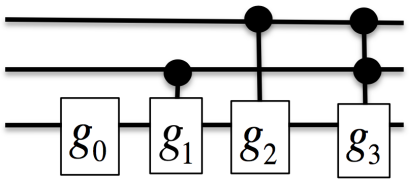
$$f_0 = g_0 \rightarrow g_0 = f_0$$
$$f_1 = g_1 \cdot g_0 \rightarrow g_1 = f_1 \cdot f_0^- 1$$
$$f_2 = g_2 \cdot g_0 \rightarrow g_2 = f_2 \cdot f_0^- 1$$
$$f_3 = g_3 \cdot g_2 \cdot g_1 \cdot g_0 \rightarrow g_3 = f_3 \cdot f_0^- 1 \cdot f_0 \cdot f_2^- 1$$

Figure 5: System of equations

5. This system of equations now forms the crux of the issue. What this method addresses is, for a given polarity, calculate the function vector of our MV GFP form multiplexer $[g_0, g_1, ...g_q^m - 1]$ in terms of the function vector from the original suboptimal multiplexer $[f_0, f1, ...f_q^m - 1]$. With the binary example, The equations that form the function vector of $f$ were calculated, and then solved for $g$ terms of the functions $f$.




(a) A Quantum Multiplexer with target functions grouped by variables which activate them.




(b) A quantum multiplexer with target functions of $g$

Figure 6: Quantum Multiplexer expressed in target functions from Positive Davio expansion and in GFP form

A physical representation of the multiplexer that results from applying a positive Davio expansion and grouping like terms, is shown in Figure 6.a. This Multiplexer was created as the result of applying equation 5. The function $f_0$ is by itself; it is not controlled by any control lines, and thus is activated in every combination. This fact is reflected in how each target function in Figure 6 has the term $f_0$. Additionally, compare Figure 6 to the equation set in Figure 5. While the system of equations could be derived through more trivial means, the process of logically converting a Karnaugh mapped Quantum Multiplexer to FPRM, then applying the Davio expansion and grouping by like terms, demonstrates how this method creates the optimized quantum multiplexers. Finally, in Figure 6.b A quantum multiplexer in GFP form is shown.

Now, as in [3] we define the transformation which maps the vectors $[f_0, f1, ...f_q^m - 1]$ to $[g_0, g_1, ...g_q^m - 1]$. This transformation is described using functional butterfly diagrams. The purpose of this paper is to present an algorithm which extends our GFP algorithm by using functional butterfly diagrams, which will break down the transformation into efficiently

calculable computations. Then, the method selects the optimal implementation based on an arbitrary cost function. The computational cost of our MV GFP method is determined by the polarity route, which is the order in which the polarities are calculated. For this paper, we let the number of butterfly transformations required to calculate a polarity be considered as the unit cost. Thus, our algorithm will require $O(m^{qm})$ butterfly transformations, where m is the number of control variables used, and q is the radix of logic, otherwise known as the number base. We have now introduced all the necessary terms and background knowledge. The important concepts of Reed-Muller logic were defined and a binary quantum multiplexer has been converted to MV GFP form using boolean algebra. The rest of the paper is organized as follows: Section 2 presents more in-depth definitions for several of the key concepts that will be used throughout this paper. Section 3 discusses the specifics of the butterfly transformations. In section 4, we present an algorithm for Generalized Fixed Polarity Method to find the optimal cost quantum multiplexer. In section 5, we discuss the overall architecture of a software implementation of this method. Finally, we conclude and address possible extensions of this research.

# 2 Terms and Definitions

## 2.1 Multivalued Generalized Fixed Polarity

The minimal Polarity route determines the order in which polarities are calculated and thus the cost of the multi-valued generalized fixed polarity method [2]. This is part of the goal of finding the circuit that reduces both the complexity of the control and the complexity of target functions. This is done by calculating target functions for different polarities. Polarities in the context of Boolean logic refer to the coefficients of a function, whether they are positive or negated. A fixed polarity is a Boolean function where the literals of a function input are only positive or negative, not mixed. Whether each bit is complemented or not, is determined by the polarity of the bit in question. [1] To demonstrate, take for example the function $G_1 = A'B + BC + ACD$. This is a simple example using OR operators. There are four possible literals. $A, B, C$, and $D$. Each of these four literals has a possible complement, $A', B', C'$ and $D'$. A fixed polarity is where we only have exclusive selection of one of these literals present in the equation solution. $G_1 = A'B + BC + ACD$ is of mixed polarity, so we define that as its default state. Changing one value $A$ to be $A'$, where $G1 = A'B + BC + A'CD$, the function is now fixed polarity. Having fixed polarity forms is important because they can be used to design easily testable circuits. In our case, fixed polarity forms require fewer products than sum of products expressions. A derivative of fixed polarity is Positive Polarity Reed-Muller form. Positive polarity is where the logical expression contains only positive literals for each variable. It is the most restrictive classification of Reed-Muller logic. In quantum computing, given a set of Boolean variables we can similarly introduce a polarity. This polarity can be implemented by inverting certain variables at the beginning of the control, which will essentially complement certain bits. These inverters are represented by a unitary matrix. Let $P_p$ represent the unitary matrix

for a polarity $p$ Now, given a multiplexer $M = [f_0, f1, ...f_{q^m} - 1]$ with control variables $C_1, C_2, C_m$, consider the implementation for a certain polarity represented $P_p$ of the control variables:

$$M = G_{q^m} - 1 \cdot G_{q^m} - 2 \cdot \cdots \cdot G_0 \cdot P_p \tag{6}$$

If given a set of variables that are q valued, ternary valued in our case, and each of those variables are limited to the basis states, we can introduce a polarity whether it is a simple binary variable or ternary variable. This polarity can be implemented by shifting certain variables at the beginning of the function. These complements form inverters which in turn can be represented by a unitary matrix. $G_i$ is a unitary matrix acting on the target variables. There are $q^m$ possible GFP forms, so a fast way to compute the transformations of Vector G from vector F is given by the calculations seen in Figure 5, and then cycle through all the possible polarities. [7] [2]

## 2.2 Unitary Block Matrix

Quantum multiplexers are represented mathematically by using Unitary Block Matrices, due to several mathematical properties of unitary matrices and quantum mechanics. A Unitary Block Matrix is a matrix where each of its entries along the diagonal are itself unitary matrices. Each of these matrices are square matrices, and all of the off diagonal matrices are zero matrices. A matrix $U$ is unitary if its conjugate transpose is also its inverse. Such that, $U * U = UU* = I$ where I is the identity matrix. Unitary matrices have significant importance in quantum mechanics and computing because they preserve norms and thus, probability amplitudes. These probability amplitudes are complex numbers used to describe behaviors of quantum systems. Quantum computations are performed when a unitary evolution operator, acting as a quantum logic gate, transforms the state of qudits in a quantum system. A unitary operator can be represented by a unitary evolution matrix, a matrix that is a quantum logic circuit that needs to be realized by a universal set of quantum logic gates. These unitary operators can then be represented by several square unitary matrices. With these unitary matrices representing the quantum logic gates, they then can be placed on the diagonal of the block matrix, forming the Unitary Block Matrix, and thus a quantum multiplexer. [7] [9]

$$M = \begin{pmatrix} f_0 & 0 & \cdots & 0 & 0 \\ 0 & f_1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & f_{2^n-2} & 0 \\ 0 & 0 & \cdots & 0 & f_{2^n-1} \end{pmatrix}$$

Figure 7: A Quantum Multiplexer represented by a unitary block matrix of arbitrary size. Note each function along the diagonal is itself square matrix.

## 2.3 Quantum Multiplexer

A quantum multiplexer is a type of gate that will, based on a set of control variables, choose a Boolean function to select an arbitrary quantum function that acts on a set of target variables. These functions can be any quantum functions, acting on any number of target variables. They are represented by unitary matrices, and the entire quantum multiplexer M can be represented as a unitary block matrix. An example of this unitary matrix is shown in Figure 7. To construct this matrix, take a q-valued quantum multiplexer and call it M. M has a set of control variables and target variables, C and T respectively. Let $C = \{c_1, c_2, ..., c_m\}$ be our set of control variables. Then, M can be thought of as a function $X : |0\rangle, |1\rangle, ... |q^m - 1\rangle \rightarrow U$ where U is the space of operations on T. Now let $X = (|i\rangle) = F_i$ . $F_i$ is the target function that is selected when the control variables are in state $|i\rangle$. We can then represent the multiplexer as an ordered set of the target functions $M = \{F_0, F_1, F_{q^m} - 1\}_C$

The chief issue with quantum multiplexers is that they often are initially designed as suboptimal or even extremely complicated when designed from the so called unitary matrices that form the basis of the quantum multiplexer operations. This research is intending to create quantum circuits with optimal, or at the least, reduced quantum multiplexers. [7] [1]

## 2.4 Ternary Quantum Logic Mathematics

In addition to the binary examples presented, ternary quantum multiplexers are also used and worked through. A ternary logic system has the logic values of $|0\rangle, |1\rangle, and |2\rangle$. These values are contained in a unit of memory called a qutrit, analogous to how classical logic uses a bit to contain information [1]. These qutrits states are represented by the three basis states, and the notation for superposition is $\alpha|0\rangle + \beta|1\rangle + \gamma|2\rangle$ where the coefficients $\alpha, \beta$, and $\gamma$ are complex numbers. The intermediate states in between the basis states are deterministic in Hilbert space before they are measured. Therefore, they cannot be non-probabilistically measured, but they can be theoretically quantified. A measurement of these intermediate states will instead reveal that the qutrit is in one of the aforementioned basis states.

## 2.5 Functional transformations and butterfly diagrams

The objective of this method from Saffat is to is describe the steps we took to transform our vector of target functions $F$ to target functions of $G$ in a GFP form multiplexer. These transformations are broken down to describe the optimal polarity route. These transformations are called butterfly transformations, and are represented on a butterfly diagram, which will be demonstrated later on. Given a group of $G$, we will define a transformation of degree $k$ over $G$ as a function $T : G^k \rightarrow G^k$ such that if $T$ takes $[x_1, x_2, ...x_k]$ to $[y_1, y_2, ...y_k]$, then $y_i$ is a fixed product of the functions $x_1, x_2, ..., x_n$. Thus, a transformation is independent of the associated group. Transformations are a generalization of matrices because if the associated group is Abelian (commutative), then the transformation is a matrix operation.

Now we introduce functional butterfly diagrams (FBD). A FBD is a mapping of a transform of one vector to another. The inputs of the transformation are on the left, and the outputs are on the right. These inputs and outputs will take the format of a matrix, the size of which is determined by the value of the selected radix. These diagrams also make an appearance in Fast Fourier and Walsh transforms and spectral method for logic synthesis.

# 3 Construction of Tmp

First, define transformations $A_p$, for zero $0 \leq p \leq q-1$. In our example with ternary, q is equal to 3, so our range in values is $0 \leq p \leq 2$. This initial step is critical because it will later serve as the basic transformation patterns in our butterfly diagram, and will more easily explain how the diagrams are mapped to each other based on the selected polarity. $Ap$ is defined as a degree $q$ transformation that takes the vector $[a_0, a_1, ...a_{q-1}]$ to $[b_0, b_1, ...b_{q-1}]$ such that $b_0 = a_{p+1}$ and $b_k = a_{k-(q-1-p)} \cdot {a_{p+1}}^{-1}$ for $k \neq 0$. We claim that

$$T_{m,p} = A_{pm} \otimes A_{pm-1} \otimes ... \otimes Ap_1 \tag{7}$$

This is then proved be induction on $m$.

## 3.1 The Base Case

First action is to establish the claim for $m = 1$. Start by calculating $T_{1,p}$. Let $M = \{F_0, F_1, ...F_{q-1}\}_c = \{G_0, G_1, ...G_{q-1}\}_{c,p}$. In order for this to be true, the multiplexers must output the same values for all inputs of the control variable. By definition, for the multiplexer $\{G_0, G_1, ...G_{q-1}\}_{c,p}, G_k, k \neq 0$, is selected by $k$ and the polarity shits the control variable by $q-1-p$, so $G_k$ is selected when the control variable is $k-(q-1-p)$. Thus, for an input of $p+1$, let $G_0 = F_{p+1}$ and let $G_k \cdot G_0 = F_k(q-1-p)$, so $G_k = F_{k-(q-1-p)}G_0^{-1} = F_{k-(q-1-p)}F_{p+1}$. Thus, for the transformation $A_p$ over $U$ takes $[F_0, F_1, ...F_{q-1}]$ to $[G_0, G_1, ...G_{q-1}]$. Now, we have $T_{1,p}$ as desired.

## 3.2 Induction

*It is shown to establish the claim for m, assuming it holds for* $m-1$. For the induction step, show that $T_{m,p} = T_{m-1,p'} \otimes T_{1,p1}$, where $p' = p - p_1 q^{m-1}$ $P'$ is the polarity $p$ without its leading digit.

Given a multiplexer $M = \{F_0, F_1, ...F_{q-1}\}_c$ with control variables $C_1, C_2, ..., C_m$ and $t$ target variables, it can be reinterpreted as a multiplexer with one control variable, $C_1$ and $tm-1$ target variables. If we let $C'$ be the set of control variables without $C_1$, this means $M = \{M_0, M_1, ...M_{q-1}\}_{\{c1\}}$, where each of $M_k$ is a multiplexer of $m-1$ control variables that can be expressed as:

$$M_k = \{F_{kq^{m-1}}, F_{kq^{m-1}} + 1, ..., F_{(k+1)q^{m-1}} - 1\}_{C'} \tag{8}$$

Now, in order to show that $T_{m,p} = T_{m-1,p'} \otimes T_{1,p1}$, we realize $M$ in two steps. First, we implement $M$ as a multiplexer with one control variable in GFP form. That is,

$$M = \{M'_0, M'_1, ..., M'_{q-1}\}_{\{c1\},p1} \tag{9}$$

Now, it will be shown that each of $M'_k$ is a multiplexer, so now it will be shown how to decompose each of the $M'_k$ into GFP form with polarity $p'$. This is shown in Figure 2.b for ternary quantum multiplexer with two control variables. First, we show that each of $M'_k$ is a multiplexer. Note that the composition of two multiplexers, is another multiplexer with target functions that are the composition of the original multiplexers target functions. Also, the inverse of a multiplexer is another multiplexer whose target functions are the inverses of the original multiplexers target functions. As long as the quantum multiplexer represented by the unitary block matrix $A$ is an $nxn$ square matrix, and contains $q$ submatrices only along the diagonal, then the inverse of quantum multiplexer (represented as a unitary block matrix $A$) is another block matrix $B$ with every the inverse of every sub matrix of $A$ inverted. This theorem only applies to diagonal block matrices. Other types of block matrices with triangular matrices, non diagonal matrices and such, have different rules on how they are inverted.

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & i & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -i \end{pmatrix} \quad M^{-1} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{2}{\sqrt{2}} & \frac{2}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{2}{\sqrt{2}} & \frac{-2}{\sqrt{2}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{2}{\sqrt{2}} & \frac{-2}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{2}{\sqrt{2}} & \frac{2}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -i & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -i \end{pmatrix}$$

(a) A diagonal block matrix (b) Inverted diagonal block matrix

Figure 8: Example of an inverted, diagonal block matrix

Thus, since the transformation $T_{1,p1}$ takes $[M_0, M_1, ...M_{q-1}]$ to $[M'_0, M'_1, ...M'_{q-1}]$ each of the functions $M'_k$ is a multiplexer. Now, let:

$$M_k{}' = \{F_{kq^{m-1}}{}', F_{kq^{m-1}+1}{}', \cdots, F_{(k+1)q^{m-1}-1}{}'\}_{C'} \tag{10}$$

Then, we can easily compute $[F_0{}', F_1{}', \cdots, F_{q^m-1}{}']$ from $[F_0, F_1, ..., F_{q^m-1}]$. Once again, because of the way multiplexers compose

$$T_{1,p1}([F_k, F_{k+q}{}^{m-1}, F_{k+2q}{}^{m-1}, ..., F_{k+(q-1)q}{}^{m-1}]) = [F_k{}', F_{k+q^{m-1}}{}', F_{k+2q^{m-1}}{}', \cdots, F_{k+(q-1)q^{m-1}}{}'] \tag{11}$$

for all $0 \leq k \leq q^{m-1} - 1$. Finally, we can realize $M = \{G_0, G_1, \cdots, G_{q^m-1}\}_{c,p}$ from $\{M'_0, M'_1, ...M'_{q-1}\}_{\{c1\},p1}$, by representing each of the multiplexers $M_k{}'$ in GFP form. Thus, we have:

$$M_k{}' = \{F_{kq^{m-1}}{}', F_{kq^{m-1}+1}{}', \cdots, F_{(k+1)q^{m-1}-1}{}'\}_{C'} = \{G_{kq^{m-1}}, G_{kq^{m-1}+1}, \cdots, G_{(k+1)q^{m-1}-1}\}_{c',p'} \quad (12)$$

$$T_{m-1,p'}([F_{kq^{m-1}}{}', F_{kq^{m-1}+1}{}', \cdots, F_{(k+1)q^{m-1}-1}{}']) = [G_{kq^{m-1}}, G_{kq^{m-1}+1}, \cdots, G_{(k+1)q^{m-1}-1}] \quad (13)$$

Therefore, by equation 11 and 13, $T_{m-1,p'} \otimes T_{1,p1}$ takes $[F_0, F_1, ..., F_{q^m-1}]$ to $[G_0, G_1, ..., G_{q^m-1}]$ and we have $T_{m,p} = T_{m-1,p'} \otimes T_{1,p1}$ as desired. This is depicted in Figure 2 for a ternary multiplexer with two control variables.

# 4 The Algorithm for Computing General Fixed Polarity Forms of a Quantum Multiplexer

Now, there is a method to compute a single polarity of $\{G_0, G_1, G_{q^m-1}\}$ from $\{F_0, F_1, ..., F_{q^m-1}\}$ we can develop the algorithm needed to compute the optimal polarity. There are four steps to follow:

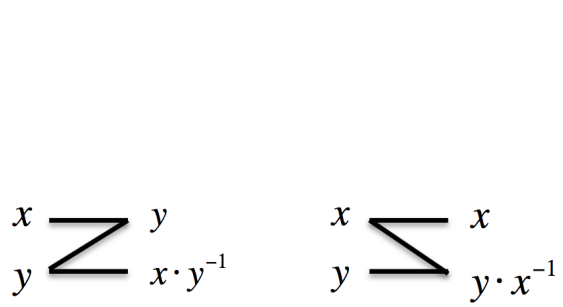
1. Initialize $v = [F0, F1, Fqm - 1]$ and $p' = 0$
2. For all $p$, with $0 \leq p \leq q^m - 1$ $p$ is the polarity that is being calculated, and the amount of polarities depends on the radix of logic and amount of control variables.

   Calculate $Vp = A_{pm} \otimes A_{pm-1} \otimes ... \otimes Ap_1(v)$, where $V_p$ is the procedure where functional butterfly diagrams are computed.

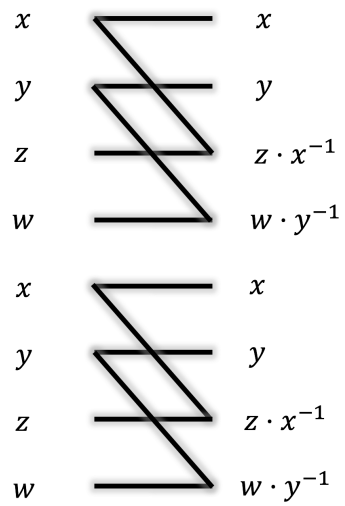
   Calculate the cost of $Vp, C(vp)$ where C is cost function.
3. If $C(vp) < C(vp)$, then $p \rightarrow p$
4. The optimal polarity is then $p'$. Let $[G_0, G_1, ...G_{q^m} - 1] = vp'$
5. Finally, implement the quantum multiplexer as $\{G_0, G_1, G_{q^m} - 1\}_{c,p}$

To demonstrate the GFP method, the notation for the functional butterfly diagrams is shown in Figure 8. These notations are for the binary example. Each one represents a butterfly transformation that corresponds to a transformation of polarities. Since this is a binary quantum multiplexer, there are two such notations. For each $A^0$ and $A^1$. $A^0$ and $A^1$ are found by applying the proofs made in section 3 with the $T_{m,p}$ equations. For this binary example, $T_{m,p} = {A^{p1}}_{m,1} \cdot {A^{p1}}_{m,2} \cdot ... \cdot {A^{pm}}_{m,m}$ These then lead to the determination that these transformations are based on the equations $A^0 = P^{0^{-1}}$ which takes $[a, b]$ to $[b, a \cdot b^-1]$, and $A^1 = P^{1^{-1}}$ takes $[a, b]$ to $[a, b \cdot a^-1]$. In the notation there is a portion of the butterfly pattern. On the left is the values X and Y, and on the right is the result of a matrix multiplication. Now, using this notation we can give an example of the GFP method. Let the Quantum Multiplexer M be $(F_0, F_1, F_2, F_3)_m$ be represented by the matrices such as the following for instance: $F_0 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $F_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}$, $F_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$, $F_3 = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}$.

These matrices are determined by a process that is outlined in other works, for it is out of scope of this paper. These certain matrices are only one such example for a multiplexer. They may represent Pauli gates, Hadamard gates, and other types of quantum gates. In a real world application, there could potentially be thousands of these matrices in these examples. They are the result of matrix rotational operations, and apply to both binary valued and ternary valued logic. A binary quantum multiplexer target function will take the form of a 2x2 matrix, a ternary valued target function will be a 3x3 matrix, and so on. The process of determining the matrices which represent the functions of this Quantum Multiplexer is outlined in [7]. The butterfly diagrams presented here are also commonly used in Fourier fast transforms and Walsh transforms.




(a) Notation for $A^0, A^1$




(b) Notation for ${A^1}_{3,2}$

Figure 9: A representation of notations for Functional butterfly diagrams in a binary valued Quantum Multiplexer

In Figure 9 there are the patterns represented by the transformations found in the previous equations. The X and Y here are Matrices, which again are determined to represent the target functions $F_0$ through $F_{q^m} - 1$. The lines demonstrate a matrix operation on the original which produces some result. In some cases, it is quite simple. The straight line from $X$ to $Y$ means the result will be $Y$. On the bottom, $Y$ goes to $X * Y^{-1}$, which is the multiplication of matrix $X$ and matrix $Y^{-1}$. There are two such patterns here in Figure 9. Each represents the diagram of $A^0$ and $A^1$ respectively. The right subfigure in Figure 9 is a notation for the entire multiplexer. The complete list of polarities for a binary valued quantum multiplexer is seen in the large Figure 11 on the next page. The way these butterfly diagrams operate is based on patterns. On a Binary valued quantum multiplexer, everything is stretched by 2. For subsequently larger radices, the factor of stretching will be equal to that radix. A ternary stretches by 3, and so on. These adjustments have to be made based on the amount of control variables present. In several of the examples shown in each of these

figures, there are only two control variables. If $m = 3$ or higher, the pattern would appear more like in Figure 12. Take for example the notation for $A^1$ in the left subfigure. This pattern is applied to the right subfigure. Since it is binary valued, the pattern "stretches" vertically; in the notation the line travels diagonally downward by one space. In this full notation however, it travels two spaces down. Another example will be shown in ternary valued logic to demonstrate this pattern. Since it is observed binary valued matrices will stretch by double the amount of their original notation, logically we can assume in ternary valued matrices for quantum multiplexers, it will stretch by a factor of 3. This fact will be shown here in Figure 12.

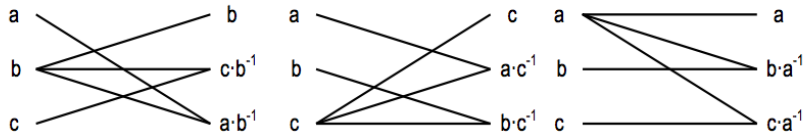
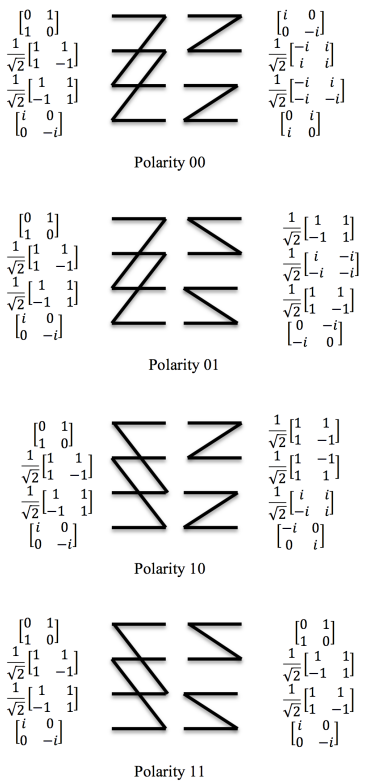
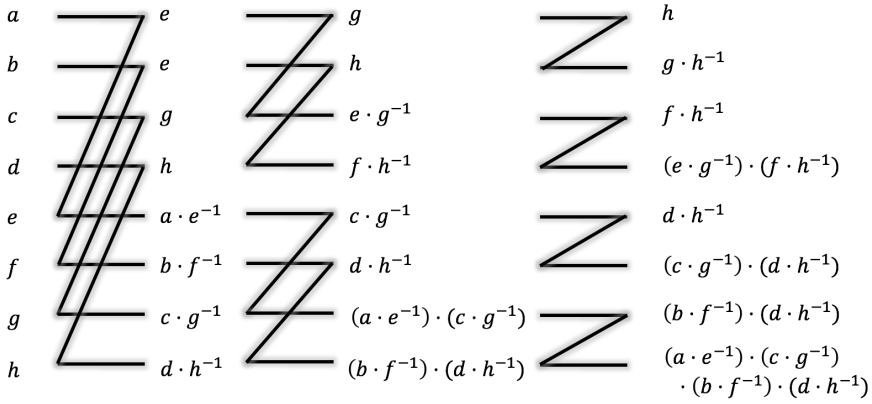



Figure 10: A representation of notations for Functional butterfly diagrams in a Ternary valued Quantum Multiplexer, $A^0, A^1, A^2$

Figure 10 demonstrates the same type of notations seen in figure 9, but this time for Ternary valued quantum Multiplexers. There are three such matrices in each notation, reflecting the ternary value, in comparison to the Binary example's two matrices seen in Figure 9. With increasing radix sizes, there will be more notations for the transforms, and the complexity of such diagrams will grow. With more control variables, the variation in the patterns will also increase via vertical stretching. This increasing complexity created a need for such a tool that can compute them correctly and quickly, and hopefully expanded to see the transforms for larger, arbitrary radices with many control variables.

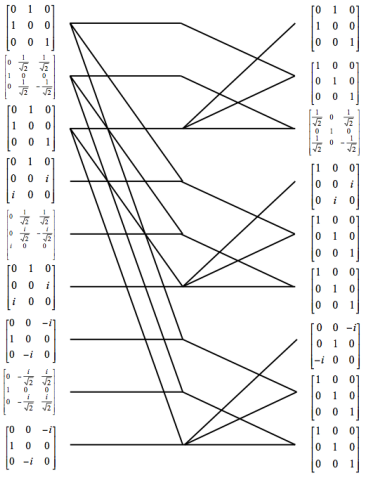



(a) A Binary valued quantum multiplexer represented by the butterfly diagram computations of $V_0, V_1, V_2, V_3$

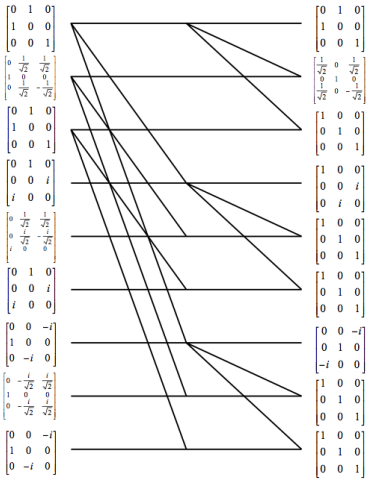



(b) A representation of polarity 000 for a three control variable binary quantum mux, note the increasing amount of matrices required and stretching of transformation patterns seen in Figure 9. Each variable represents a square, 2x2 unitary matrix.

Figure 11: A representation of Multiplexers in Functional butterfly diagrams for a binary valued Quantum Multiplexer of 2 and 3 control variables.



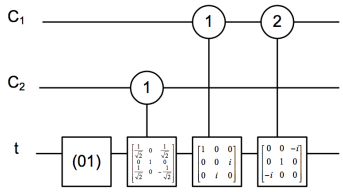
(a) Ternary valued quantum multiplexer in polarity 21.



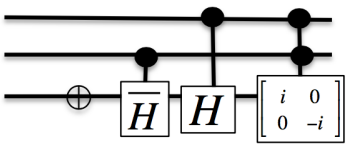
(b) Ternary valued quantum multiplexer in polarity 22.

Figure 12: A representation of notations for Functional butterfly diagrams in a Ternary valued Quantum Multiplexer.

Figure 11 contains examples of what a binary valued quantum multiplexer appears while represented as a butterfly diagram. Another example is for ternary. Ternary follows the same line of patterns but with an added complexity. These patterns are described in Figure 10. Here, there are only 2 such polarities calculated for the multiplexer, but there are many more. For sake of brevity the other polarities in the ternary multiplexer have been omitted. Based on the cost function outlined earlier, with the minimal amount of transformations, polarity 22 is our selected optimal polarity for ternary valued quantum multiplexers. Likewise, for binary valued multiplexer in Figure 11, our selected optimal polarity based on the cost function is 11. Below is an example of the reduced and optimal multiplexer for a both ternary valued and binary valued quantum multiplexers.




(a) The optimal Polarity $22_3$, for ternary valued multiplexers, using the GFP method and implemented as a multiplexer.




(b) The optimal Polarity of 11. Note the H stands for Hadamard, which are the gates represented by several of the matrices in this polarity.

Figure 13: A representation of notations for Functional butterfly diagrams in a Ternary valued Quantum Multiplexer

# 5 The Software Component

One of the main extensions of this method, of which was originally described in [2] is the addition of a software component. Because of the intricate mathematical computations required to create these functional butterfly diagrams, and the exponentially increasing amount of polarities needed to calculate them, doing so without the aid of numerical analysis tool can be very error prone or otherwise impossible. To aid in the computation of these polarities, and also put a visible tool for this method that has been created and proved, a script written in MATLAB has been developed. MATLAB is a proprietary, multi-paradigm numerical computing environment developed by Mathworks. While MATLAB is not a free tool, monetary cost or software philosophy otherwise, using other programming languages, namely C and C#, were quickly ruled out due to the need for external packages to handle matrix manipulation, and general complexity due to the increased overhead to create and manipulate matrices. MATLAB was selected because of its core strength in matrix manipulation, and its weakly typed variable interface allows for mixing and matching of values in matrices. This is crucial because as an example, several matrices may initially be composed of double values, but the resulting computations may result in a matrix with complex numbers. Weakly typed pertains to the type system of a programming language, where the compiler must determine the contents of variables, and either loosely or strictly enforce types, such as integer, floating point, or a character string. Weakly typed programming is much more simpler for this type of script, compared to a explicitly typed language. Explicitly typed is far more strict, as the programmer must explicitly state the kind of data contained within the variable, rather than the compiler implicitly defining it.

The script in its current iteration takes in the inputs of $q$ and $m$, which is our radix and our number of control variables. Based on these variables, the script can quickly determine the number of polarities it will need to calculate. This value is then run through the main loop, which will parse each index of a decimal to a binary converted character array. It traverses through this character array in big endian format, as it begins from the most significant bit. Each bit location needs a different pattern, as evidenced by the several examples presented in the previous section. As the control variable count $m$ increases, the layout of the butterfly patterns needs to be adjusted. The most significant bit in our case is stretched by $q$ times the original pattern. Using the transformation equations, these patterns can be computed and then our cell array of matrices (which at this current time are composed of rotated Hadamard matrices) will be sent to an external function which will determine the correct order of transformations and apply the multiplication operations necessary. The goal of this script was to make it as extendable as possible, with the intention of using it for increasing radix values and control variables. Below is the included code, followed by the several functions used to create the patterns. There are five such patterns, two for binary valued logic, and three for ternary valued logic. Ideally, constructing these kinds of patterns with some type of constructor would be more efficient and expandable, however increasing radices introduce increasingly complex transformations. For now, the patterns are hardcoded in the form of external functions.

```
Q = 2;  %value
M = 10; %control variables selected, can possibly be made to accept user
             %input in future

%for loop to create cell array to store, cell is {ROW, COL}
divisor = 360/(Q^M);   %create rotation needed, for uniqueness

for index = 0:1:(Q^M)-1
    %create some random, but unitary matrices here, and assign to 1st row.
    angleinRads = degtorad(index * divisor);
    if Q == 2
        Target = hadamard(Q) * [[cos(angleinRads), -sin(angleinRads)];
                                [sin(angleinRads), cos(angleinRads)]];
    elseif Q == 3
        Target = (orth(randn(3,3))) * [[cos(angleinRads),-sin(angleinRads), 0 ];
                           [sin(angleinRads), cos(angleinRads), 0];[0,0,1]];
    end
    MatrixOpsCell{1,index+1} = Target;

    %not sure if this will correctly get our matrices, but it produces
    %rotated hadamards, a quantum gate type.  This section will need more
    %experimentation.
end

%compute diagrams for polarities, the amount of polarities is Q^M
for index = 0:1:(Q^M)-1
    %take polarity as a function of q to the m, and give it M decimal
    %spaces, converts to arbitrary base Q.
    P = dec2base(index, Q ,M);

    %programmatically find the polarities provided in binary form,
    %it is critical we know which one is in each index because it
    %determines the pattern we use on the diagrams
    for k = 1:1:M
       FindStorage{k} = P(end-(M-k));   %create digits in Findstorage cell
    end

    %display polarity
    sprintf('Computing polarity %s', P)
    IMMCell = MatrixOpsCell;            %save matrix, so we can show input
           %later.
```

```
%cycle through each bit available, based on M
for j = 1:1:M
    %analyze the first bit, we need to stretch our pattern, descending
    %this is our stretch factor, should work for both binary and
    %ternary radices.  it is exponential, every index increase the stretch
    %gets bigger.
    Stretch = (Q^(j-1));

    %case switch, look at radix base.  2 or 3, determines patterns
    %needed, enclosed by foor loop, which does every 'column' of the
    %fbd for each polarity.
    switch Q
        case 2
            if FindStorage{j} == '0'
                IMMCell = ComputeBinary0(Stretch, IMMCell, Q, M, j);
            elseif FindStorage{j} == '1'
                IMMCell = ComputeBinary1(Stretch, IMMCell, Q, M, j);
            end
        case 3
            if FindStorage{j} == '0'
                IMMCell = ComputeTernary0(Stretch, IMMCell, Q, M, j);
            elseif FindStorage{j} == '1'
                IMMCell = ComputeTernary1(Stretch, IMMCell, Q, M, j);
            elseif FindStorage{j} == '2'
                IMMCell = ComputeTernary2(Stretch, IMMCell, Q, M, j);
            end
        otherwise
            sprintf('Invalid controls amount selected or radix')
    end

end
%display matrices before we reset and move to the next polarity.
sprintf('Polarity %s \n', P)
fprintf('Input = \n')
for k = 1:1:(Q^M)
    sprintf('F%d' , (k-1))
    disp(MatrixOpsCell{k}).  %input saved earlier
end

sprintf('Output = \n')
for k = 1:1:(Q^M)
    sprintf('G%d', (k-1))
```

```
        disp(IMMCell{k}) %output saved
    end
end

function Result = ComputeBinary0(Stretch, IMMCell, Q, M, j)
    %function that computes a binary valued, transformation pattern A0

    %This outer loop ensures the entire "column" ie the control
    %variable, is accounted for.  each column has the same pattern repeated
    %several times, which S accounts for.

    %inner loop is for "pattern bank" ie how many patterns in one block,
    %where S is equal how many "blocks" we run in one column. once
    %that has been complete, the control variable is considered finished.
    for s = 0:(Q^j):((Q^M)-1)
        for k = 1:1:(Q^(j-1)) %how many patterns in a bank, 1, 2, 4, ect
            %the actual pattern, made by calculating index via stretch, and
            %inverting certain matrices.
            Result{k+s} = IMMCell{(k+s)+(1*Stretch)};
            Result{((k+s)+ Stretch)} = (IMMCell{k+s} *
                                        (inv(IMMCell{((k+s) + Stretch)})));
        end
    end
end


function Result = ComputeBinary1(Stretch, IMMCell, Q, M , j)
%function that computes a binary valued, transformation pattern A1
%similar to Binary0, except the pattern indices are swapped.
    for s = 0:(Q^j):((Q^M)-1)
        for k = 1:1:(Q^(j-1))
            Result{k+s} = IMMCell{k+s};
            Result{((k+s)+ Stretch)} = (IMMCell{((k+s) + Stretch)} *
                                        (inv(IMMCell{k+s})));
        end
    end
end



function Result = ComputeTernary0(Stretch, IMMCell, Q, M, j)
%function for transformation pattern A0 for ternary operations
```

```
    for s = 0:(Q^j):((Q^M)-1)
        for k = 1:1:(Q^(j-1)) %how many patterns in a bank, 1, 2, 4, ect
            Result{k+s} = IMMCell{(k+s)+(Stretch)};
            Result{((k+s)+ Stretch)} = (IMMCell{(k+s)+(Stretch * 2)} *
                                        (inv(IMMCell{((k+s) + Stretch)})));
            Result{((k+s)+ (Stretch * 2))} = (IMMCell{k+s} *
                                        (inv(IMMCell{((k+s) + Stretch)})));
        end
    end
end

% a   =   b
% b   =   c * b^-1
% c   =   a * b^-1

%function for transformation pattern A1 for ternary operations

function Result = ComputeTernary1(Stretch, IMMCell, Q, M, j)
    for s = 0:(Q^j):((Q^M)-1)
        for k = 1:1:(Q^(j-1))
            %how many patterns in a bank, 1, 2, 4, ect
            Result{k+s} = IMMCell{(k+s)+(Stretch*2)};
            Result{((k+s)+ Stretch)} = (IMMCell{(k+s)} *
                                    (inv(IMMCell{((k+s) + Stretch*2)})));
            Result{((k+s)+ (Stretch * 2))} = (IMMCell{(k+s) + Stretch} *
                                    (inv(IMMCell{((k+s) + Stretch*2)})));
        end
    end
end

% a   =   c
% b   =   a * c^-1
% c   =  b * c^-1

%function for transformation pattern A2 for ternary operations

function Result = ComputeTernary2(Stretch, IMMCell, Q, M, j)
    for s = 0:(Q^j):((Q^M)-1)
        for k = 1:1:(Q^(j-1)) %how many patterns in a bank, 1, 2, 4, ect
            Result{k+s} = IMMCell{k+s};
            Result{((k+s)+ Stretch)} = (IMMCell{(k+s)+ Stretch} *
                                                (inv(IMMCell{k+s})));
```

```
            Result{(k+s)+ (Stretch * 2)} = (IMMCell{(k+s) + (Stretch * 2)}
                                                    * (inv(IMMCell{k+s})));
        end
    end
end

% a   =   a
% b   =   b * a^-1
% c   =  c * a^-1
```

The script is fairly short, and is accompanied by 5 functions. Function files in MATLAB must be externally located, as is explained later on. Each function can process one pattern of transformation, namely the patterns $A^0$, $A^1$, $A^n$ and so on, depending on the radix of logic. In the script's current version, the only radices supported are binary and ternary. The script functions off of two variables, Q and M. Q is expected to be 2 or 3, while M can vary in size from 1 all the way to larger amounts, like 10, maybe even 100 control variables. Then, a For loop creates the matrices needed depending on the radix selected. It is here where the greatest change can occur. For now, binary valued quantum multiplexers are computed with rotated Hadamard matrices. Ternary valued Multiplexers are rotated orthogonal matrices. As long as the matrices are unitary, there won't be a possibility of a computed polarity having the result of NaN (not a number, essentially undefined or erroneous data) in MATLAB. If the result or input is located in the complex plane, the script can easily handle the different planes. This was one of the many advantages of MATLAB over C#. Fortunately, the script allows for a lot of flexibility. A function could be created in the future to make the ideal matrices for each target function. Note, however that as the variables get larger and larger, the more matrices are needed for each polarity.

For computation of the polarities, an outer for loop is begun. It starts at 0, and continues to the largest polarity $(Q^M) - 1$. The number base is converted, and the character string is parsed for each computation. The script continues through a case - switch statement, enclosed by another for loop. This for loop iterates through each bit. There, it will select one of the patterns it needs to compute, finishing when all control variables have been computed, before moving on to the next polarity. It will finally display the input and output matrix for the polarity before resetting and moving on to the next one. Each pattern is slightly different, but generally uses several variables to ensure every index is computed correctly. Each function represents the computation of the "column" that one would see on an FBD. The higher the value of M is, the more columns one would see if it were to be drawn. Inside each of these columns one pattern is used to represent the transformation. There are pattern banks which means how many times a pattern repeats inside of the column. The demarcation of the beginning of these pattern banks is denoted by the variable $S$. The stretch factor is computed as well, and will be used to "stretch" the pattern based on which control variable it is. These variables work together to ensure the correct index is set for the pattern's computation. All patterns function very similarly in this regard, even amongst different radices. The difficulty in creating this script centered around ensuring the proper

indexes were set during each pattern. Once this algorithm was formed, the rest of the script was trivial to complete.

The script successfully computes the examples seen here in this paper, for both binary and ternary valued quantum multiplexers. However, these are only one such example. In a more realistic application, there could be many types of matrices used. These results validated the method and also ensured the patterns correctly multiplied the matrices. In addition, additional control variables were added. In the worked through examples in throughout this paper, there are typically only two control variables. Now, this script can compute many more. As such, 10 was used as an example. However, one should note that these are exponential functions that pertain to computing all polarities. Having 10 control variables vastly increases the amount of matrices that need to be calculated, resulting in a script run time of almost three minutes on a modern PC with quad-core processing and hyper-threading enabled. With subsequently higher values of $M$, the increase in available polarities to compute will increase the runtime required. Another big issue is that the output window on MATLAB can only support so many lines before truncating. With over 1,048,576 matrices to produce for a 10 control variable quantum multiplexer, as an example, one may only be able to view that last 600 matrices, give or take. To rectify this, one would need to output the matrices in a readable text file. A sample output of the computation of polarity 00 on a 2 control variable multiplexer has been provided.

```
%sample output from a Q = 2 M = 2 mux, rotated hadamards used
Polarity 10

Input =

F0

     1     1
     1    -1

F1

    1.0000   -1.0000
   -1.0000   -1.0000

F2

   -1.0000   -1.0000
   -1.0000    1.0000

F3

   -1.0000    1.0000
    1.0000    1.0000

Output =

G0

   -1.0000   -1.0000
   -1.0000    1.0000

G1

    0.0000    1.0000
   -1.0000    0.0000

G2

   -1.0000   -0.0000
    0.0000   -1.0000
```

```
G3

    1.0000   -0.0000
    0.0000    1.0000
```

Some comments on the construction of this script cover several of its shortcomings. One such shortcoming is the lack of visual representation in the form of butterfly diagrams. Ideally, for ease of use and readability, it would have been more prudent to use MATLAB's plot functionality to create a visual diagram that is a functional butterfly diagram, for each of these polarities. Given the limited selection of toolboxes available, as MATLAB needs certain software toolboxes, which come at a monetary cost on top of an existing installation of MATLAB, creating a visual of these diagrams is very time intensive. In fact, these diagrams are so uncommon there is not an official version available to plot. While graphics can be created in MATLAB, it will take far longer to implement. Using a different language like C# which can more easily draw graphics could be explored as a potential option. Lastly, future versions of MATLAB can be used beyond version 2016b, which allows the use of functions within the main script file. While originally this script was developed in a trial version of 2017a, mid-development it was switched to 2015b, which does not allow the use of function files within a script file. This means functions must be in multiple external files. For the sake of compatibility, the script runs intended for pre-2016b installations of MATLAB, with multiple function files included that must be run within the same environment path. This extension is merely a quality of life improvement in development, but does make development easier in the long term.

# 6 Conclusions and Extensions

In this paper, a method for implementing quantum multiplexers based on the Fixed Polarity Reed-Muller method for implementing classical boolean functions has been extended. This method was called the Generalized fixed polarity method. First, the rigorous definitions for GFP were established. The reasoning and mathematical applications were presented and explained, through use of binary valued quantum multiplexers. Several examples were worked through to demonstrate how a suboptimal form of a multiplexer can be reduced with GFP. Then, we developed a method to calculate any arbitrary polarity by decomposing the transformation from the basis vector to any polarity into butterfly diagrams. This Multi valued GFP method requires that we calculate $O(mq^m)$ functional butterfly diagrams for a quantum multiplexer with $m$ control variables. This method is an expanded version of the method described in [2] to MV GFP forms. The success of this method is determined by the computational complexity pertaining to the number of functional butterfly diagrams we must calculate.

Now, we describe several more forms that can be used to realize MV quantum multiplexers. Many of these forms do not have classical analogues, although they can be inspired by

various classical forms. These additional forms serve two purposes. First, they generalize several methods of logic synthesis to quantum multiplexers, allowing for cheaper realization of these quantum multiplexer circuits. Second, for many of these forms, there may alternative polarity routes, like one described in the extensions section of [3] that may require fewer functional butterfly diagrams to compute all polarity forms. These would require only $O(q^m)$ instead of $O(mq^m)$ functional butterfly diagrams.

The first extension of this method begins with an MV GFP form of arbitrary polarity. Define a function $f_k : \{0, 1, ..., q-1\} \rightarrow \{0, 1, ...q-1\}$ and permute the controls of the $k$th control variable using the function $f_k$. That is, whenever a function is controlled by the value $x$ of $C_k$ in MV GFP form, it is now controlled by $f_k(x)$. Then, there are $q!^m$ new forms for each polarity of MV GFP.

Another extension is where the control variables do not need to all have the same number of basis states; with each control variable having potentially a different radix than the other. A quantum multiplexer with mixed logic can be created.

The final extension would be to extend the definition of polarity [2]. In this method, polarity is defined by shifting each of the variables by a constant $0 \leq k \leq q-1$, thus having $q^m$ polarity vectors. Polarity could be defined by applying an arbitrary permutative gate to each of the control variables. By applying a function $f_k$, that permutes its basis states, at the input of the multiplexer. This is called a *cyclic inverter* and given a quantum multiplexer with $m$ control variables and a radix of logic $q$, there are $q!^m$ polarities using cyclic inverters.

Finally, the main addition to this method has now been codified in to a expandable tool that can compute the increasingly complex diagrams with a software script. Working through larger quantum multiplexers manually, by hand calculations, can lead to an error-prone conclusion. Due to the ever changing and evolving power of software, refinement is always needed. A straightforward extension of this script in MATLAB would simply be to calculate arbitrary radix sizes, for multiplexers with logic bases beyond ternary. To programmatically compute the patterns needed to execute the matrix operations is more challenging than recursively adding in multiple control variables. In its current iteration it can compute the matrices for the target functions and subsequently the butterfly diagrams (in a non visual form) in terms of the matrix operations.

# References


[1] Diphal Shah, Marek Perkowski, *Design of Regular Reversible Quantum Circuits*, Portland State University, 2010.

[2] Tashin Saffat, M.P *Method for Optimizing q-Valued Quantum Multiplexers*, Westview High School, 2012.

[3] Tashin Saffat, M.P *A method for optimizing quantum multiplexers* Proc. of ULSI 2012

[4] Tzvetan Metodi, A.F *Quantum Computing for Computer Architects* Madison: Morgan & Claypool, 2011.

[5] Debatosh Debnath, T.S *Exact Minimizations of Fixed Polarity Reed-Muller Expressions for Incompletely Specified Functions.* Jyushu Institute of Technology (n.d).

[6] Frederic T. Chong, T.S *Quantum Computing for Computer architects.* Morgan & Claypool, 2010.

[7] Faisal Shah Khan, Marek Perkowski *Synthesis of Ternary Quantum Logic Circuits by Decomposition* Electrical and Computer Engineering Faculty Publications and Presentations. Paper 220 2005.

[8] D.S Jankovic, R.S Stankovic, R Drechsler *Efficient calculation of Fixed polarity polynomial expressions for multiple valued logic functions* proc. ISMVL pg 76-82 2002

[9] Riley T. Perry *The Temple of Quantum Computing* Version 1.1 2006