# EE677 Project

Jay Mehta (22B1281)

Kshitij Vaidya (22B1829)

Jainam Ravani (22B1242)

Kushal Jain (22B4514)

November 2024

# Contents

# Chapter 1

# Introduction

## 1.1 Quantum Circuit Synthesis

Quantum Circuit Synthesis is the process of developing an optimisied quantum circuit for a given set of truth tables. The quantum circuit is a sequence of quantum gates that are applied to the qubits. The quantum gates are the basic building blocks of the quantum circuit. The quantum circuit synthesis is an important step in the quantum computing as it helps in reducing the number of quantum gates and the number of qubits required for the quantum algorithm. The quantum circuit synthesis is a challenging task as the quantum gates are non-commutative and the quantum circuits are probabilistic in nature. The quantum circuit synthesis is an NP-hard problem and the quantum circuit synthesis algorithms are based on the heuristics and the optimisation techniques.

# Chapter 2

# The Algorithm

The process of conversion of the set of truth tables to an optimisied quantum circuit involves the following steps:

1. The input function is first recorded in the form of a truth table.
2. The input can consist of a set of truth tables. For example, the function can consist of 3 inputs and 2 outputs like:

| A | B | C | F1 | F2 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

3. The next step is to convert the set of truth tables into a bijective function, that is each input combination maps to a unique output combination. This is done by extending output by some bits.
4. The number of bits required to make function bijective is given by, $n = \lceil \log_2(m) \rceil$, where $m$ is the maximum repetition of an output value in the original truth tables.

   For the above example

| A | B | C | F1 | F2 | Output value |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | $(01)_2 = 1$ |
| 0 | 0 | 1 | 1 | 0 | $(10)_2 = 2$ |
| 0 | 1 | 0 | 1 | 0 | $(10)_2 = 2$ |
| 0 | 1 | 1 | 0 | 1 | $(01)_2 = 1$ |
| 1 | 0 | 0 | 1 | 0 | $(10)_2 = 2$ |
| 1 | 0 | 1 | 0 | 1 | $(01)_2 = 1$ |
| 1 | 1 | 0 | 0 | 1 | $(01)_2 = 1$ |
| 1 | 1 | 1 | 1 | 0 | $(10)_2 = 2$ |

Here, $m = 4$, so $n = \lceil \log_2(4) \rceil = 2$. So, we need to extend the output by 2 bits.

5. Now we extend the output by $n$ bits. The output is now represented as $F1F2F3F4$. This is done in order to make the function bijective. For the above example

| A | B | C | F1 | F2 | F3 | F4 | Output value |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | $(0100)_2 = 4$ |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 | $(1000)_2 = 8$ |
| 0 | 1 | 0 | 1 | 0 | 0 | 1 | $(1001)_2 = 9$ |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | $(0101)_2 = 5$ |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | $(1010)_2 = 10$ |
| 1 | 0 | 1 | 0 | 1 | 1 | 0 | $(0110)_2 = 6$ |
| 1 | 1 | 0 | 0 | 1 | 1 | 1 | $(0111)_2 = 7$ |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | $(1011)_2 = 11$ |

6. Now we also extend the function to accomodate the remaining values between 0 and $2^n - 1$. Here $n = m + n_0$, where m = number of bits extended and $n_0$ = initial output bits. This is done by adding a new rows to the table, these rows are basically garbage inputs.

7. Now we can find the reed muller expansion of each of the output bits using positive davio expansion.

8. The reed muller expansion is then converted to a quantum circuit using the reed muller expansion to quantum circuit conversion algorithm. This algorithm involves converting the output bits to input by applying all possible Toffoli gates and prioritising the expansions which lead to a state closest to the input. This is done by using a priority queue by following a BFS traversal of the tree.

9. This algorithm relies on the invertibility of quantum gates.

# Chapter 3

# The Code

## 3.1 The Code Structure

The code is divided into the following files:

- **main.py** : This is the main file which contains the main function. This file reads the input from the user and calls the required functions.
- **truth_table.py** : This file contains the processing of the truth table.
- **reed_muller.py** : This file converts a truth table to reed muller expansion and also contains some necessary functions.
- **bijective.py** : This file contains the functions to convert the truth table to a bijective function.
- **quantum_circuit.py** : This file contains the functions to convert the bijective functions to a quantum circuit.

## 3.2 The Code Flow

The code flow is as follows:

1. The main function reads the input from the user.
2. The main function calls the truth table processing function from the truth table file.
3. The truth table processing function calls the bijective function from the bijective file.
4. The bijective function calls the reed muller expansion function from the reed muller file.

5. The reed muller expansion function calls the quantum circuit function from the quantum circuit file.
6. The quantum circuit function converts the reed muller expansion to a quantum circuit.
7. The output is written to a file.

## 3.3 truthTable.py

The Python class, `TruthTable`, represents a truth table. The class allows you to:

- Retrieve outputs for a given set of binary inputs.
- Generate a sub-table by fixing one variable to a specific value.
- Print the truth table in a readable format.

### 3.3.1 Explanation of Code

**Class Initialization (`__init__`)**

The constructor initializes the truth table with the following parameters:

- `n`: Number of variables.
- `variables_list`: List of variable names (e.g., `['x1', 'x2']`).
- `output_list`: Output values for all combinations of inputs, ordered by binary sequence.

`Function getOutputs`

This method fetches the output for a given binary input.

- Converts the binary input list to an integer using base-2 conversion.
- Uses this integer as an index to access the output list.

**Example:** For inputs `[1, 0, 1]` (binary `101`), the integer index is `5`, and the method returns `output_list[5]`.

`Function getSubTable`

This method generates a sub-table by fixing one variable, `x_i`, to a specific value (`0` or `1`).

- Removes `x_i` from the variable list.
- Iterates through all possible combinations of the remaining variables.

- Inserts the fixed value of `x_i` into each input and fetches the corresponding output.
- Returns a new `truth_table` object for the sub-table.

`Function printTable`

This method prints the truth table in a readable format:

- Generates all possible combinations of variables as binary inputs.
- Fetches the output for each input and displays the results.

# Usage Examples

## Full Truth Table

The following code creates a truth table for 3 variables and prints it:

```
variables = ['x1', 'x2', 'x3']
outputs = [0, 1, 1, 0, 1, 0, 0, 1]
tt = TruthTable(3, variables, outputs)
tt.printTable()
```

**Output:**

```
Truth table:
Variables: ['x1', 'x2', 'x3']
000 0
001 1
010 1
011 0
100 1
101 0
110 0
111 1
------------------------
```

## Fetching an Output

You can fetch the output for a specific input (e.g., `[1, 0, 1]`):

```
print(tt.getOutputs([1, 0, 1]))  # Output: 0
```

### Sub-Table Generation

You can fix a variable (e.g., `x2 = 1`) and generate the corresponding sub-table:

```
sub_tt = tt.getSubTable('x2', 1)
sub_tt.printTable()
```

**Output:**

```
Truth table:
Variables: ['x1', 'x3']
00 1
01 0
10 0
11 1
------------------------
```

## 3.4 reedMuller.py

The given code defines a class `RM` that computes and manipulates the Reed-Muller (RM) form for Boolean functions. The class accepts a truth table as input and provides methods for:

- Computing the Reed-Muller form recursively.
- Converting the form to a list of lists.
- Evaluating the Reed-Muller form for given inputs.
- Updating the function with new literals and expressions.

### 3.4.1 Explanation of Code

The class `RM` contains the following methods:

#### Class Initialization (`__init__`)

The constructor method initializes the Reed-Muller class instance. It takes a `truth_table` object as an argument, which contains:

- `truth_table.varLisr`: A list of variables.
- `truth_table.n`: The number of variables in the Boolean function.
- `truth_table.output_list`: The output list of the truth table.

It also computes the Reed-Muller form using the `getReedMuller` method and converts the form into a list of lists using the `listOfLists` method.

### 3.4.2 The `getReedMuller` Method

The `compute_RM` method recursively computes the Reed-Muller form for a given truth table. The method works as follows:

- **Base case**: If there is only one variable, the function checks the output values for all possible inputs and returns the corresponding RM form. If the output is always 0, then it returns 0. If the output is always 1, then it returns 1. If the output is the same as the input variable $x_i$ then it returns $x_i$ otherwise it returns $1 \oplus x_i$.

```
        if self.numVar == 1:
        if self.truthTable.outputList == [0, 1]:
            return [self.truthTable.varList[0]]
        elif self.truthTable.outputList == [1, 0]:
            return ['1', self.truthTable.varList[0]]
        elif self.truthTable.outputList == [0, 0]:
            return ['0']
        elif self.truthTable.outputList == [1, 1]:
            return ['1']
```

- **Recursive case**: For multiple variables, the method divides the truth table into two parts, `subTableVar` and `subTableBarVar`, based on the first variable. It recursively computes the RM forms for these parts:

```
        subTableVar =
            self.truthTable.getSubTable(self.truthTable.varList[0],
            0)
        subTableBarVar =
            self.truthTable.getSubTable(self.truthTable.varList[0],
            1)
```

- **Combining results**: The method then combines the RM forms for `subTableVar` and `subTableBarVar` by multiplying the terms with the variable $x$ and $\overline{x}$, respectively. It handles the removal of zero terms and duplicates because duplicate terms result in 0 in RM form.

```
        xFx = []
        for term in reedMullerVar:
            if term == '0':
                continue
            elif term == '1':
                xFx.append(self.truthTable.varList[0])
            else:
                xFx.append(self.truthTable.varList[0] +
                    term)

        xFxBar = []
        for term in reedMullerBarVar:
```

```
            if term == '0':
                continue
            elif term == '1':
                xFxBar.append(self.truthTable.varList[0])
            else:
                xFxBar.append(self.truthTable.varList[0]
                    + term)
        combinedReedMuller = xFx + xFxBar +
            reedMullerVar
```

The resulting RM form is returned.

**Function `listOfLists`**

This method converts the Reed-Muller form, represented as a list of strings, into a list of lists, where each list contains the variables involved in a particular term. If a term involves a constant `'1'`, it is replaced by a list containing `1`.

**Function `evaluateReedMuller`**

This method evaluates the Reed-Muller form for a given set of inputs. It iterates through each term in the RM list and multiplies the corresponding values of the variables. The result is summed and taken modulo 2, as required for Boolean functions.

**Function `updateFunction`**

The `updateFunction` method updates the Reed-Muller form by adding new literals to existing terms and removing duplicates. It performs the following steps:

- It checks each term in the RM form to see if it contains the literal `alterLiteral`.
- If `alterLiteral` is present, it is replaced with the new expression, which can include other literals.
- The resulting terms are checked for duplicates, which are removed using the `Counter` function.

### 3.4.3 The `updateClassFunction` Method

This method calls the `updateFunction` method to update the Reed-Muller form. After updating, it updates the variable list and prints the updated RM form.

## 3.5 Usage Examples

### 3.5.1 Example 1: Creating an `RM` Object and Printing the RM Form

Consider a truth table object `tt` with variables {x1, x2} and an output list {0, 1, 1, 0}. To create an `RM` object and print the Reed-Muller form, use the following code:

```
rm = RM(tt)
rm.print_RM()
```

### 3.5.2 Example 2: Evaluating the RM Form for Given Inputs

To evaluate the Reed-Muller form for the inputs {1, 0}, use the following code:

```
inputs = [1, 0]
output = rm.evaluateReedMuller(inputs)
print(output)
```

### 3.5.3 Example 3: Updating the Function with New Literals

To update the function by replacing `x1` with a new literal `a` and adding the expression {a, b} to the terms containing `x1`, use:

```
new_literals = ['a', 'b']
expression = [['a', 'b']]
alter_literal = 'x1'
rm.updateClassFunction(new_literals, expression, alter_literal)
```

## 3.6 solver.py

The primary purpose of this class is to convert a boolean function (represented by its truth table) into a reversible function by applying bit extensions. The class detects and handles repeated terms, extends the bit-length, and then converting the function into its Reed-Muller (RM) form.

## 3.7 Explanation of Code

**Class Initialization (`__init__`)**

The constructor method initializes the `Solver` class with three parameters:

- Size of the boolean function

- A list of the literals
- A list of the truth tables for the output variables

It performs the following operations:

- Initializes the class variables and converts the list of truth tables into instances of the `truth_table` class.
- Calls `print_vars` to display the initial information such as the size, literals, and the truth tables.
- Calls `update_truth_tables` to update the truth tables by removing repeated terms and extending their bit-length to make the function reversible.
- The size of the function and the literals are updated based on the bit extension, ensuring that all truth table terms are distinct.
- Converts each truth table to its Reed-Muller form using the `RM` class and prints the results.

```
    class Solver:
    def __init__(self, numVar : int, literals : List[str],
        listTables : List[List[int]]):
        self.numVar = numVar
        self.literals = literals
        self.listTables = listTables
        self.truthTables = []
        for table in self.listTables:
            self.truthTables.append(TruthTable(self.numVar,
                self.literals, table))

        self.newListTables, self.numVar =
            self.updateTruthTables()
        # print(type(self.newListTables[0]))
        self.reedMullerList = []
        for table in self.newListTables:
            tempTable = TruthTable(self.numVar,
                self.literals, table)
            self.reedMullerList.append(ReedMuller(tempTable))
            print(self.reedMullerList[-1].getReedMuller(),
                file=outputFile)
            # print(self.reedMullerList[-1].reedMuller())
```

**Function** `printVariables`

This method is responsible for printing the following details about the boolean function:

- Size of the function.
- List of literals (variables like `x1`, `x2`, etc.).
- Truth tables.

It is invoked twice: once during the initialization of the `Solver` class and once after the truth tables have been updated and bit extensions applied.

```
def printVariables(self):
    print(file=outputFile)
    print(f'Size of Function : {self.numVar}',
        file=outputFile)
    print(f'Variables : {self.literals}', file=outputFile)
    print(file=outputFile)
    print('Truth Tables :', file=outputFile)
    for table in self.truthTables:
        table.printTable()
    print(file=outputFile)
```

**Function** `updateTruthTables`

This method plays a key role in making the boolean function reversible. It performs the following steps:

- Combines the original terms from the truth tables into a single list, where each term is weighted based on its position.
- Detects repeated terms in the truth tables and applies bit extension to remove redundancy. The idea behind bit extension is to enlarge the bit representation of repeated terms, making each term unique and thus ensuring that the function is reversible.
- If no repeated terms are found, the function remains unchanged.
- If bit extension is necessary, the bit-length is increased to accommodate the maximum repetition of terms. This is done by scaling the terms.
- After applying the bit extension, the truth tables are reconstructed with the new terms, ensuring that they represent a unique set of values for the function, which is essential for reversibility.

```
    def updateTruthTables(self):
        terms = []
        for i in range(len(self.listTables[0])):
            num = 0
            for j in range(len(self.listTables)):
                num = num + self.listTables[j][i] * (2 ** j)
            terms.append(num)
        print(f'Original Terms : {terms}', file=outputFile)
```

```
9
10          # Finding the repeated terms
11          counts = Counter(terms)
12          print('Counts :', file=outputFile)
13          print(counts, file=outputFile)
14          # Print the repeated terms
15          repeatedTerms = []
16          for key, value in counts.items():
17              if value > 1:
18                  repeatedTerms.append(tuple((key, value)))
19          # Bit extension = log2(maxRepeatedTerms)
20          if repeatedTerms == []:
21              bitExtension = 0
22              print('No Bit Extension', file=outputFile)
23              return self.listTables, self.numVar
24
25          maxRepeats = max(repeatedTerms, key = lambda x :
                x[1])
26          bitExtension = ceil(log2(maxRepeats[1]))
27          print(f'Bit Extension Needed : {bitExtension}',
                file=outputFile)
28
29          newTerms = []
30          # Update the terms to accomodate for the
                bitExtension
31          for i in range(len(terms)):
32              newTerms.append(terms[i] * (2 ** bitExtension)
                    + counts[terms[i]] - 1)
33              counts[terms[i]] = counts[terms[i]] - 1
34
35          # Add values to the literals to accomodate for the
                extra variables
36          for i in range(bitExtension):
37              self.literals.append(f'x{self.numVar+i+1}')
38
39          self.numVar = self.numVar + bitExtension
40          for i in range(2 ** self.numVar):
41              if i not in newTerms:
42                  newTerms.append(i)
43          print(f'New Terms \n {newTerms}', file=outputFile)
44
45          # Convert the new terms into appropriate Truth
                Tables
46          newListTruthTables = []
47          for i in range(self.numVar):
48              newTable = []
49              for j in range(len(newTerms)):
50                  newTable.append((newTerms[j] >> i) & 1)
51              newListTruthTables.append(TruthTable(self.numVar,
                    self.literals, newTable))
```

```
52          # print(newListTruthTables)
53          return newListTruthTables, self.numVar
```

## 3.8 Usage Example

Here is an example demonstrating how to use the `Solver` class:

- Assume we have a boolean function with three variables: `x1`, `x2`, and `x3`, and their corresponding truth tables.
- We will use the `Solver` class to make the function reversible by applying bit extensions and updating the truth tables accordingly.

Example code:

```
size = 3
literals = ['x1', 'x2', 'x3']
list_truth_tables = [
    [1, 0, 1, 1, 0, 1, 1, 0],
    [0, 1, 1, 0, 1, 0, 1, 1],
    [1, 1, 0, 0, 1, 1, 0, 1]
]

solver = Solver(size, literals, list_truth_tables)
```

This will create an instance of the `Solver` class, which performs the following:

- Prints the initial size, literals, and truth tables.
- Updates the truth tables by applying bit extensions to make the function reversible.
- Prints the updated truth tables and their Reed-Muller form.

## 3.9 IterativeSolver.py

The `IterativeSolver` class implements an iterative algorithm to optimize a list of Reed-Muller (RM) representations of boolean functions. Its goal is to iteratively reduce the complexity of the RM list by minimizing the number of terms while keeping track of optimization metrics. The following are the main highlights of the class:

- The algorithm uses a priority queue (heap) to explore possible substitutions iteratively.
- A scoring function based on weighted parameters (`alpha`, `beta`, and `gamma`) drives the optimization.
- The recursive approach explores various combinations of literals to simplify the RM representations.

## 3.10 Code Explanation

### Class Initialization (`__init__`)

The constructor initializes the class, sets up the priority queue, and starts the recursive algorithm.

- Inputs:
  - `rm_list`: A list of RM forms of boolean functions.
  - `literals`: A list of literals (variables).
  - `params`: A tuple containing the weighting parameters `(alpha, beta, gamma)`.
- Initializes metrics `depth`, `elim`, and `term_count`.
- Pushes the initial state of the RM list into the priority queue with a computed score.
- Generates the list of substitution options using the `generateOptionsReedMuller` method.
- Calls `recursiveAlgorithm` to build the solution tree.

```
    def __init__(self, reedMullerList, literals,
        parameters):
        self.reedMullerList = reedMullerList
        self.literals = literals
        self.numEqns = len(reedMullerList)

        self.iterator = 0
        self.priorityQueue = []
        self.finalAnswer = []
        self.bestDepth = 0

        self.alpha = parameters[0]
        self.beta = parameters[1]
        self.gamma = parameters[2]

        depth = 1
        eliminations = 0
        termCount = 0
        answer = []
        for RM in reedMullerList:
            termCount += len(RM.reedMuller)
        self.bestTermCount = termCount

        pushElement = [reedMullerList, depth, eliminations,
            termCount, answer]
```

```
        heapq.heappush(self.priorityQueue,
            (self.alpha*depth + self.beta*(eliminations /
            depth) - self.gamma*termCount, pushElement))

        self.generateOptionsReedMuller()
        self.recursiveAlgorithm()
```

**Function** `generateOptionsReedMuller`

This method generates all possible substitution options for simplifying the RM list:

- For each literal, combinations of other literals are generated as substitution options.
- Stores each substitution option as a pair of target literal and its substitution.

```
    def generateOptionsReedMuller(self):
        self.options = []
        for i in range(self.numEqns):
            literal = self.literals[i]
            otherLiterals = [x for x in self.literals if x
                != literal]
            self.options.append([[[literal], ['1']],
                literal])
            result = []
            remLen = len(otherLiterals)
            for combSize in range(1, remLen + 1):
                result.extend(combinations(otherLiterals,
                    combSize))
            for res in result:
                self.options.append([[[literal], res],
                    literal])
        return self.options
```

**Function** `generateOptionsForRM`

This helper method returns the substitution options generated earlier.

```
def optionsForRM(self):
    return self.options
```

**Function** `reedMullerAfterChecking`

Applies a given substitution option to each RM in the RM list and returns the updated RM list.

```
    def reedMullerListAfterChecking(self, reedMullerList,
        option):
        finalReedMullerList = []
        for i in range(self.numEqns):
            if type(reedMullerList[i]) == list:
                finalReedMullerList.append(reedMullerList[i].updateFunction(reedMulle
                    option[0], option[1]))
            else:
                finalReedMullerList.append(reedMullerList[i].updateFunction(reedMulle
                    option[0], option[1]))
        return finalReedMullerList
```

**Function** `recursiveAlgorithm`

The core recursive optimization algorithm:

- Pops the top element from the priority queue (state with the best score).
- If the term count is minimal, the solution is found, and the recursion terminates.
- Otherwise, generates new states using substitution options and pushes them into the priority queue if they improve the score.
- Calls itself recursively to explore the next state.

```
    def recursiveAlgorithm(self):
        popElement = heapq.heappop(self.priorityQueue)[1]
        reedMullerList, originalDepth, _, originalTerms,
            originalAns = popElement

        space = '--' * originalDepth
        if setPrint:
            print(f'{space} Current Depth is
                {originalDepth}', file = outputFile)
            print(f'{space} Current Functions :',
                file=outputFile)
            for i in range(self.numEqns):
                if type(reedMullerList[i]) == list:
                    print(f'{space}  {reedMullerList[i]}',
                        file=outputFile)
                else:
                    print(f'{space}
                        {reedMullerList[i].reedMuller}\n',
                        file=outputFile)
        space += '--'

        if (originalTerms == self.numEqns):
            print('SOLVED!!!', file = outputFile)
```

```
18              self.finalAnswer = originalAns
19          else:
20              options = self.optionsForRM()
21              print(f'Options : {options}', file=outputFile)
22
23              for option in options:
24                  newReedMullerList =
                        self.reedMullerListAfterChecking(reedMullerList,
                        option)
25                  if setPrint:
26                      print(f'{space} Substituting {option}',
                            file=outputFile)
27                      print(f'{space} New Functions :',
                            file=outputFile)
28                      for i in range(self.numEqns):
29                          print(f'{space}
                                {newReedMullerList[i]}',
                                file=outputFile)
30                  newDepth = originalDepth + 1
31                  newEliminations = 0
32                  newTerms = 0
33                  newAnswer = originalAns + [option]
34
35                  for RM in newReedMullerList:
36                      newTerms += len(RM)
37
38                  if (newTerms < self.bestTermCount):
39                      self.bestTermCount = newTerms
40                  self.iterator += 1
41                  print(f'{self.iterator} \t
                        {self.bestTermCount} \t {newTerms}',
                        file=outputFile)
42
43                  newEliminations = originalTerms - newTerms
44                  pushElement = [newReedMullerList, newDepth,
                        newEliminations, newTerms, newAnswer]
45
46                  if (newEliminations >= 0 and newTerms <=
                        self.bestTermCount):
47                      print(f'Pushing element with depth
                            {newDepth} best count
                            {self.bestTermCount}, term count
                            {newTerms} Reed Muller List
                            {newReedMullerList}',
                            file=outputFile)
48                      heapq.heappush(self.priorityQueue,
                            (self.alpha*newDepth +
                            self.beta*(newEliminations /
                            newDepth - self.gamma*newTerms),
                            pushElement))
```

```
            self.recursiveAlgorithm()
```

## 3.11 Usage Example

Here is an example of how to use the `IterativeSolver` class:

```
    def main():
    # listTables = [[1, 1, 1, 1, 0, 0, 0, 0],
        #                       [0, 1, 1, 0, 0, 0, 1, 1],
    #                 [0, 0, 1, 1, 1, 1, 0, 1]]
    # numVar = 3
    # literals = ['x1', 'x2', 'x3']
    outputFile = open('output.txt', 'w')
    finalAns = open('finalAnswer.txt', 'w')
    numVar = 2
    literals = ['x1', 'x2']
    listTables = [[0, 1, 1, 0], [0, 0, 1, 1]]

    # Making the function reversible by running the Solver
    solverInstance = Solver(numVar, literals, listTables)
    # tables, size = solverInstance.updateTruthTables()
    tables, size = solverInstance.newListTables,
        solverInstance.numVar
    # print('Table :')
    # print(type(table[0]))
    tempLiterals = [chr(ord('a') + i) for i in range(size)]
    for table in tables:
        table = TruthTable(size, tempLiterals, table)
        table.printTable()
    print(f'Size : {size}', file = outputFile)

    literals = []
    for i in range(size):
        literals.append(chr(ord('a') + i))

    # Getting all the Reed Muller Representations
    reedMullerList = []
    # print(type(tables[i]))
    for i in range(size):
        # truthTable = TruthTable(size, literals,
            tables[i].outputList)
        truthTable = TruthTable(size, tempLiterals,
            tables[i])
        reedMuller = ReedMuller(truthTable)
        reedMullerList.append(reedMuller)
        # print(f'Function {i+1} :
            {reedMuller.listOfLists()}')

```

```
    parameters = [0, 0, -1]
    solver = IterativeSolver(reedMullerList, literals,
        parameters)
    for step in solver.finalAnswer:
        print(step, file=finalAns)
```

This will:

- Generate substitution options.
- Optimize the RM list recursively.
- Print the final solution once the minimal term count is achieved.

# Chapter 4

# Results

We begin with three truth tables representing Boolean functions $f_1(x_1, x_2, x_3)$, $f_2(x_1, x_2, x_3)$, and $f_3(x_1, x_2, x_3)$. The input-output mappings for these functions are as follows:

| $x_1$ | $x_2$ | $x_3$ | $f_1(x_1, x_2, x_3)$ | $f_2(x_1, x_2, x_3)$ | $f_3(x_1, x_2, x_3)$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 |

Each column corresponds to the truth table output for a specific function given all possible combinations of inputs $x_1, x_2, x_3$.

### Reed-Muller (RM) Representation

For each function, we compute the Reed-Muller decomposition, which expresses the Boolean function in a canonical form. The RM decomposition for each function is represented as a list of terms after making the function bijective. The output obtained is:

Function 1: $\{\{a\}, \{b\}, \{1\}\}$
Function 2: $\{\{a, b, d\}, \{a, d\}, \{b, d\}, \{c\}, \{d\}\}$
Function 3: $\{\{a, b, d\}, \{a, c, d\}, \{a, b\}, \{a, c\}, \{b, c, d\}, \{b\}, \{c\}\}$
Function 4: $\{\{a, c\}, \{a, c, d\}, \{a, d\}, \{b, c\}, \{b, c, d\}, \{b\}, \{b, d\}\}$

Here {a,b,d} will denote the AND of a,b, and d. {a,b,d},{a,d} denotes the XOR of the terms abd and ad. Therefore it represents : abd $\oplus$ad.

## Iterative Solver

The Reed-Muller representations are passed to an iterative solver.

$$\text{Parameters: } \alpha = 0, \beta = 0, \gamma = -1.$$

The solver outputs a series of steps, which are stored in `output.txt`. The following are the results of the iterative solver based on the Reed-Muller decompositions of the Boolean functions:

$$d => d \oplus ab$$

$$d => d \oplus b$$

$$c => c \oplus abd$$

$$b => b \oplus ac$$

$$d => d \oplus bc$$

$$b => b \oplus c$$

$$a => a \oplus 1$$

$$c => c \oplus ad$$

$$b => b \oplus acd$$

$$c => c \oplus d$$

$$a => a \oplus b$$

Each step represents a choice made by the algorithm in the solution tree to modify the Reed muller functions. By following each of these above steps in the solution tree(by applying the above transformations), we will reach the final quantum circuit.