Creating Reversible Truth Tables with Minimal Garbage and Antica Bits using RTT

Erik Gabrielsen
Bobby B. Lyle School of Engineering
Southern Methodist University
Dallas, Texas 75205
Email: egabrielsen@smu.edu

***Abstract*—As conventional computers continue to get faster and more efficient, they begin to approach their hardware limitations. The upbringing of Quantum Computing gives rise to advances in technology that will solve problems that computers today are not able to solve. One of Quantum Computing's advantages over Digital Computing is that all logic gates are reversible. This distinct characteristic has led to research in Reversibility Synthesis in both Digital Computer Design and Quantum Computer Design.**

In this paper, I will be exploring a method, RTT, of creating Reversible truth tables from conventional truth tables in a way that most efficiently uses garbage bits and antica bits. While today we are able to compute small reversible truth tables for some conventional truth tables, this method will make it possible to take larger functions and generate a reversible truth table to reduce energy dissipation in today's computers, as well as to make it easier to generate Reversible Quantum gates.

***Keywords*—Garbage bit, Antica bit, Quantum Logic, RTT, Reversible Logic**

I. Introduction

Over the past few years, extensive research has gone into reversible logic gates because of its impact in Conventional Computing and the growing field of Quantum Computing. I will outline those impacts below.

A. Conventional Computing

For conventional computers, the technology advancements in circuit design and transistor design are causing an increase in the computational utility achieved in a given quantity of time, space, material, energy and cost [1]. According to Moore's Law [2], the number of transistors on one computer chip will double every year and a half on average, meaning that the transistor will reach the atomic level in size by the year 2050. Because of this, advances in computer chip design must be made in other areas. One such area of research is reversible logic gates. Before taking a look at reversible logic gates, it is just as important to understand non-reversible functions. In many non-reversible functions, there is a loss of information from the input to output.

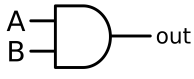

Fig. 1: A Conventional AND Gate

In Figure 1 we can see a digital logic *AND* gate. The *AND* gate takes inputs *A* and *B* and yields the output *out*. When we are given the inputs it is easy to determine the output, however when given the output, we lose a piece of information since it is only possible to determine one of the inputs *A* or *B* with absolute certainty. Because we lose one bit of information, this translates to energy dissipation in the form of heat. Landauer proved that the energy dissipation from energy loss is *kTln2* Joules, where k is a constant and T is the temperature of the chip [3]. For larger gates, this energy loss begins to add up and become very costly, especially when transistors themselves become the size of atoms. In reversible logic gates, there is no energy loss because information is preserved in the output and can easily be mapped back to its respective input value.

B. Quantum Computing

In a Quantum computer, unlike computers of today, every function, algorithm, method, or logic gate is a unitary matrix that is reversible. For instance if we were to consider a quantum *CNOT* gate depicted in Figure 2 we can see that at at state $|a\rangle$ the value of $|xy\rangle$ is $|x, x \oplus y\rangle$. When $|xy\rangle$ undergoes another *CNOT* operation, the value goes back to $|xy\rangle$.

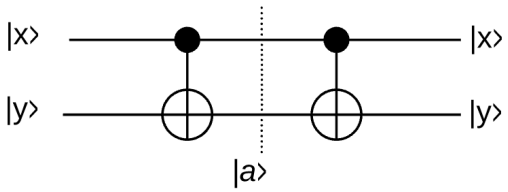

Fig. 2: Quantum CNOT Gate Reversibility

This phenomenon is one of the characteristics of Quantum computing that makes it more powerful than conventional computers because here, there is no energy loss due to a loss of information. We can correctly obtain the inputs of this *CNOT* gate given the output by passing it through the same gate a second time.

In the rest of the paper I will examine a method of transforming conventional truth tables into fully reversible truth tables that can then be used to create lossless digital logic gates and reversible quantum logic gates.

II. Definitions and Assumptions

A. Reversible Logic

Reversible logic is a logic design style in which the following two conditions are met:

1) there is a one to one mapping between the input and the output vectors [1]
2) the input and output vectors are onto (for every input *x* there exists some output *y*)

These two conditions guarantee that there is no information loss in the circuit. The goal of my research was to construct an algorithm that can transform any conventional truth table given the inputs and outputs and generate the reversible representation of the truth table using the least amount of ancila and garbage bits. An ancila bit is added to the input to sustain the reversibility of the truth table, and a garbage bit is added to the output of the truth table to sustain its reversibility.

B. Assumptions

When implementing my method to reverse truth tables, I made several assumptions about the truth tables that should hold true. These assumptions are outlined below:

1) There is at least one input and at least one output variable
2) Every input combination must have an output
3) The only possible input/output values are a 1 or 0
4) Control lines are assumed to be apart of the input
5) The number of initial outputs does not exceed the initial number of inputs

The 4^{th} assumption is important for certain truth tables such as the function for a mux, where there is *n* input wires, *log(n)* control wires and *m* output wires. I will also refer to my method of reversing truth tables as *RTT* for the remainder of the paper.

III. Methodology for *RTT*

A. Technologies Used

All code for *RTT* was done in a python notebook. I chose to use python so that I could utilize the numpy library, a fundamental scientific computing tool that is used to represents *N* dimensional arrays. I also utilized pandas, a high-performance data analysis tool that gave me access to many necessary array and vector manipulations needed to perform RTT.

Each input and output is represented as a 2D array in python. In several different instances I cast the 2D array to a panda Data Frame Object which allows me to do specific operations that will be covered later on.

B. Checking Reversibility

The first step in a *RTT* program is checking to see if the truth table is already reversible. To do this I first check to make sure that the input and output are the same shape. This checks for the 2^{nd} requirement for a reversible function. If this condition holds, then I iterate through the 2D output array and check to see if each tuple is a unique row. If this is not the case then it does not satisfy the 1^{st} condition of reversibility. The code for checking for reversibility is provided in Algorithm 1

Algorithm 1: Check Reversibility Function

```
def isReversible(input, output):
    if input.shape == output.shape:
        # check if one to one
        seen = set()
        unique_rows = []
        for item in output:
            t = tuple(item)
            if t not in seen:
                unique_rows.append(item)
                seen.add(t)
        unique_rows = np.array(unique_rows)
        return unique_rows.shape == output.shape
    else:
        return False
```

C. Finding Minimum Number of Antica and Garbage Bits

Using the fact that reversibility requires there to be a one to one relationship between the input and output, I can determine how many antica and garbage bits are needed for reversible in *O(n)* time.

Algorithm 2: Determining the Minimum Number of Garbage and Antica bits

```
def findMin(input, output, dict={}):
    max_key = max(dict, key= lambda x: len(set(dict[x])))
    num_garbage = len(dict[max_key])
    num_antica = (num_garbage + output.shape[1]) - input.shape[1]
    # Convert numbers to binary and get length for log(n) bits
    num_garbage = len("{0:b}".format(num_garbage))
    num_antica = len("{0:b}".format(num_antica))
    return num_garbage, num_antica
```

Because the antica bits are always going to be a 0 or a don't care, we only need to worry about the garbage bits and the outputs. Therefore by examining the output, we can find the output that is duplicated the most amount of times, and this will yield how many garbage bits are needed, and likewise how many antica bits are needed. The total number of garbage bits needed is equal to the number max number of duplicated outputs. The number of antica bits is equal to the number of input bits subtracted from the total number of output bits (including garbage bits).

TABLE I: A Full Adder Truth Table

c_{in}	x	y	sum	carry
0	0	0	0	0
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	1

Lets consider the case of a Full Adder Truth Table. The Full adder truth table is depicted in Table I.

The outputs marked in bold are the most often duplicated output in this truth table. So from this we can immediately tell that the number of garbage bits needed will be 2 (the number of duplications). This is because we need a unique ouput to map to each input, so with 3 outputs that are the same, we need a minimum of 2 bits to represent those outputs. Therefore the worst possible case for RTT would be when all the outputs are the same, meaning that we would need *log(n)* number of garbage bits, where *n* is the number of output combinations in the truth table.

Finding the minimum number of garbage bits needed for a truth table is important for both digital and quantum circuit design. Specifically for Quantum circuits, just knowing the number of garbage and antica bits needed for a circuit, we can immediately draw up the circuit diagram required for that specific function. Concider the full adder once more. Knowing that we need 2 garbage bits and 1 antica bit in addition to our 3 inputs and 2 outputs, we can then create a quantum circuit that takes in 4 qubits, and then perform some function on those qubits to obtain the full adder. A depiction of the full adder unitary matrix design for a quantum circuit is depicted in Figure 3.

Fig. 3: Quantum Full Adder Gate

IV. *RTT*

The Reverse Truth Table function is broken up into 3 different sub routines that make it possible to generate the optimal reversible truth table for any given input output vector pair. Those routines are listed in order below:

1) Marking duplicate output tuples
2) Table Expansion
3) One to One Mapping

A. Marking Duplicate Output Tuples

Algorithm 3: mark_duplicate_tuples()

```
def mark_duplicate_tuples(output):
    seen = set()
    duplicates = {}
    index = 0
    for row in output:
        t = tuple(row)
        if t in seen:
            if t in duplicates:
                duplicates[t].append(index)
            else:
                duplicates[t] = [index]
        seen.add(t)
        index += 1
```

Figure 3 outlines the first method in *RTT*. Here, I will only consider the output 2D array as a parameter. This routine then iterates through each tuple of the 2D array and looks for duplicates. If one is found, the tuple values of the 2D array is added to the dictionary as the key and its value is an array containing the row number that the tuple is found on. For the Full Adder from Figure I The data structure once completed would look something like Table II.

TABLE II: Duplicates Dictionary

Key	Value
(1,0)	[1, 2, 4]
(0,1)	[3, 5, 6]

This method runs in *O(n)* where *n* is the number of tuples in the output 2D array. It only has to pass through the array once in order to map corresponding duplicate outputs (finding duplicates time complexity is of *O(1)* in this case) to row indices. Once these values are in the dictionary, we can then use them for Step 2 in the *RTT* process.

B. Table Expansion

After marking duplicate output tuples, *RTT* will begin the table expansion. This method will ensure that the second condition of a reversible truth table holds true: for every input *x* there must exists some output *y*. This method is also responsible for determining how many antica and garbage bits are needed for the reversibility of the truth table. Once these two numbers are determined, the truth table is expanded with the appropriate number of antica and garbage bits which are all set to an initial value of *0*. Figure 4 provides the code for this routine.

Algorithm 4: expand_table()

```
def expand_table(input, output, dict={}):
    max_key = max(dict, key= lambda x: len(set
    (dict[x])))

    # Gets # of garbage and antica bits
    num_garbage = len("{0:b}".format(len(dict[
    max_key])))
    num_antica = (num_garbage + output.shape
    [1]) - input.shape[1]

    # Add on column of 0s for every antica bit
    new_input = np.hstack((input, np.zeros((
    input.shape[0], num_antica), dtype=input.
    dtype)))

    # Add on column of 0s for every garbage
    bit
    new_output = np.hstack((output, np.zeros((
    output.shape[0], num_garbage)), dtype=input
    .dtype)))
```

The *expand_table* method take in the input and output vectors, as well as the dictionary of duplicate tuples and row indices that was calculated from step one of *RTT*. Using the dictionary, *expand_table* finds the maximum number of

duplicates and converts it to a binary number. Then by finding the length of this binary number, we know how many garbage bits are needed and consequently how many antica bits are needed - *log(n)* bits where n is the max number of duplication.

For this method, the time complexity is *O(nlog(n))* as it loops through each 2D array *log(n)* times at worst case and inserts a 0 *n* times. At the conclusion of this step, we will have a truth table that looks like Table III.

TABLE III: A Full Adder Truth Table Expanded

a_0	c_{in}	x	y	sum	carry	**g_0**	**g_1**
0	0	0	0	0	0	**0**	**0**
0	0	0	1	1	0	**0**	**0**
0	0	1	0	1	0	**0**	**0**
0	0	1	1	0	1	**0**	**0**
0	1	0	0	1	0	**0**	**0**
0	1	0	1	0	1	**0**	**0**
0	1	1	0	0	1	**0**	**0**
0	1	1	1	1	1	**0**	**0**

The bold face columns are the garbage and antica bits added on during this step.

C. One to One Mapping

The final step, following table expansion, is one to one mapping. In this stage the *RTT* will map each input to a unique output, ensuring that the 1st condition of a reversible truth table holds true. The outline of this step is outlined in Algorithm 5

Algorithm 5: one_to_one_mapping()

```
def one_to_one_mapping(input, output,
    num_antica, num_garbage, duplicates={}):
    buffer = output.shape[1] - num_garbage
    get_bin = lambda x, n: format(x, 'b').
    zfill(n)

    for key in duplicates:
        number = 1
        for item in duplicates[key]:
            garbage = get_bin(number,
    num_garbage)
            for col in range(num_garbage):
                output[item][col+buffer] = int
    (garbage[col])
            number = number + 1

    return (input, output)
```

The parameters for *one_to_one_mapping* are the input and output 2D arrays and the number of antica and garbage bits that were generated from the table expansion in step 2, and the duplicates dictionary found from step 1. Then each tuple in the duplicates dictionary is looped over. For each tuple we then loop through each row index that corresponds with that tuple value. Each iteration increases the number representation for those garbage bits by 1. The variable *number* will never exceed the binary representation of 2^g garbage bits as we calculated how many garbage bits were needed already in step 2. In Table IV, we can view the finished product of RTT and check that it is indeed reversible.

The time complexity for *one_to_one_mapping* is *O(n*g)* where n is the number of duplicates and g is the number of garbage bits required by *RTT*.

TABLE IV: A Reversible Full Adder Truth Table

a_0	c_{in}	x	y	sum	carry	g_0	g_1
0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0
0	0	1	0	1	0	0	1
0	0	1	1	0	1	0	0
0	1	0	0	1	0	1	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	1	0
0	1	1	1	1	1	0	0

V. Performance

To test the performance of *RTT*, I examined the time of execution when running different values of n where n is the number of input bits and output bits. Because I wanted to test the worst case time for each scenario, I made the output lines all 0 so that RTT had to iterate through the max number of times required to make the truth table reversible.

TABLE V: RTT Time Complexity

Routine	Time Complexity
Marking Duplicates	$O(n)$
Table Expansion	$O(nlog(n))$
One to One Mapping	$O(n * g)$

Table V gives the time complexity for each step in the process and I will be examining the overall performance of RTT. I will test with the values of n given in Table VI.

TABLE VI: RTT test values of n

n	$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12$

A. Results

Based on the results outlined in Figure 4, we can see that the relationship between execution time and the number of bits, n, is exponential. This is to be expected as the truth table itself will increase in size by a value of 2^n. Because RTT's execution time grows at the same rate as the truth table itself, I can confirm that the table expansion step in RTT is the most expensive step in the method where Time Complexity is equal to $O(nlog(n))$. With such small values of n we can expect

RTT to run without any problems, but as soon as n reaches a certain threshold, the performance begins to struggle as the table is growing at a exponential rate.

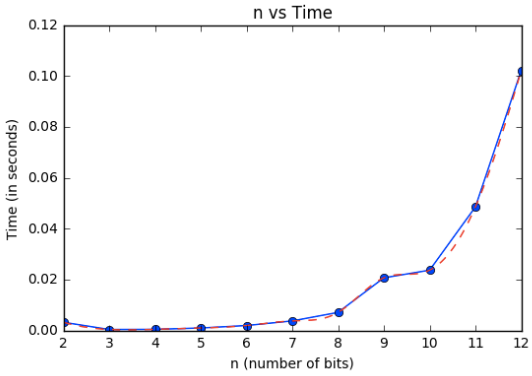

Fig. 4: RTT n vs $time$

VI. Further Research

In my research I only considered reversible truth tables for logic functions where the number of inputs were always greater than or equal to the number of outputs. The reason for this is because all digital logic functions and gates can be made up of any combination of the following digital logic gates: AND, NOT, and OR [5]. To reverse truth tables where the inputs are outnumbered by the output, we would just at a pre-processing step to the truth table where we fill in the necessary number of antica bits with all 0 to match the number of outputs. After the step is completed RTT can continue and reverse that table successfully. In addition, creating a reversible truth table is just one step in having digital circuits that do not have any heat dissipation from loss of information. More research is needed in ways of creating a digital circuit design from a reversible truth table that limits the amount of fanout and gates to reduce the cost of the digital circuit. And finally, research for quantum computing reversible gate synthesis is still needed to further explore how to transform today's digital circuits into tomorrow's quantum circuits.

VII. Conclusion

Reversible logic synthesis brings a new solution to the digital computer logic problem as transistors approach atomic size. We can start to design and create new simple logic gates for conventional computers that run on low power because of little to no energy dissipation from information loss. We can also use reversible logic synthesis to generate new quantum circuits that can inevitably replace those of a conventional computer. As research into Quantum Computing and reversible logic synthesis intensifies, we will become closer and closer to solving world problems that we cannot solve on a conventional computer, leading to advances in technology beyond our comprehension.

Acknowledgment

I would like to thank Dr. Mitch Thornton of Southern Methodist University for my introduction into Quantum Computing and for supporting my continued interest in the field.

References

[1] Vandana Maheshwari, *Development of SyReC Based Expandable Reversible Logic Circuits* Rajasthan, India: Rajasthan Technical University, 2014.

[2] Gorden E. Moore, *The Future of Integrated Electronics* Fairchild Semiconductor internal publication, 1964.

[3] R. Landauer *Irreversibility and heat generation in the computing process*, IBM J. Res. Dev., vol. 5, p. 183, 1961.

[4] R. Drechsler and R. Wille, *From Truth Tables to Programming Languages: Progress in the Design of Reversible Circuits*, Bremen, Germany: University of Bremen, 2011.

[5] David Y. Feinstein and Mitchell A. Thornton, *On the Guidance of Reversible Logic Synthesis by Dynamic Variable Reordering*, Dallas, Tx: Southern Methodist University.