

3-3-2023

Design and Optimization with Quantum and Memristor Platforms

Yiwei Li
Portland State University

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



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Li, Yiwei, "Design and Optimization with Quantum and Memristor Platforms" (2023). *Dissertations and Theses*. Paper 6339.

<https://doi.org/10.15760/etd.8193>

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Design and Optimization with Quantum and Memristor Platforms

by

Yiwei Li

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

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
Xiaoyu Song, Chair
Marek A. Perkowski
Fu Li
Jingke Li

Portland State University
2023

Abstract

The complexity and diversity of modern computing challenges have made it difficult for traditional computers to efficiently handle tasks such as computer-aided design, design optimization, and combinatorial problems within a reasonable time frame. In applications such as bitcoin mining and robotic control, power consumption and circuit size are critical factors. To overcome these limitations, this dissertation examines alternative computing technologies such as quantum computers and hybrid memristive circuits. The research demonstrates that these technologies hold the potential to offer more efficient and effective solutions for particular problems, whether in terms of faster computation or reduced power consumption.

Table of Contents

Abstract	i
List of Tables	vi
List of Figures	viii
Chapter 1 Introduction	1
1.1 Overview of quantum computing.....	1
1.2 Overview of memristor circuit	2
Chapter 2 Quantum computing.....	3
2.1 Quantum computing.....	3
2.2 Quantum circuits and gates	3
2.3 Grover’s algorithm	4
Chapter 3 Grover’s algorithm for generalized Ashenhurst-Curtis decomposition	9
3.1 Generalized Ashenhurst-Curtis decomposition.....	10
3.2 Functional decomposition problem formulation	13
3.3 Grover algorithm for Ashenhurst-Curtis decomposition	21
3.3.1 Finding optimal ΠG by search with an oracle.....	21
3.3.2. Proposed methodology to construct quantum oracle for decomposition	23
3.3.3 Encoding scheme for the partitions	24

3.3.4 Representation of partition multiplication using boolean encodings	25
3.3.5 Testing of refinement relation with boolean function	26
3.3.6. Oracle synthesis for decomposition problem with known free and bound sets	29
3.3.7 Oracle synthesis for the decomposition problem with minimum number of blocks.....	35
3.4 Quipper and circuit modeling.....	38
3.5. Resource analysis of oracle and algorithm simulation via Quipper.....	43
3.5.1 Resource analysis by Quipper	43
3.5.2 Simulation with Quipper	47
3.6 Conclusion.....	51
Chapter 4 Quantum hybrid graph coloring algorithm for finding column multiplicity.....	53
4.1 Graph coloring and related work.....	54
4.2 Graph coloring based on domination covering	57
4.3 Quantum oracle for finding the domination pairs	61
4.4 Quantum counting.....	63
4.5 Quantum circuit block for building quantum oracle	65
4.6 Hybrid quantum algorithm for graph coloring	67
4.7 Quipper modeling and simulation.....	68

4.8 Experiment Analysis	73
Chapter 5 Design and optimization of memristive FSM	75
5.1 Memristor and related work	75
5.1.1 History of memristor	75
5.1.2 Memristor circuit and related work	76
5.1.3 Memristor characteristic and memristor circuit.....	77
5.2 Low-power memristive state machine	80
5.2.1 Execution sequence of the memristive state machine	83
5.3 Background for the state assignment problem and related work	85
5.4. State assignment of Memristive FSM	89
5.4.1 Operators and relations on cubes.....	91
5.4.2 generation of prime implicants of multi-valued function.....	93
5.5 Experimental Results.....	109
Chapter 6 Memristor-based pulse rate system	112
6.1 Pulse rate measurement systems and related research	112
6.2 Development of memristor-based components for a pulse rate system.....	114
6.2.1 Circuits to perform arithmetic operations on two pulse rate signal.....	115
6.2.2 Addition	116
6.2.3 Pulse rate integrator.....	117

6.2.4 System of nonlinear equations.....	119
6.3 Implementation using memristor imply gates.....	119
6.3.1 Implement RM with imply gates.....	119
6.3.2 Implement RC with imply gates.....	127
6.4 Design of pulse rate system with memristors FPGA	134
Chapter 7 Conclusion.....	135
7.1 Related Work and Contributions.....	135
7.1.1 Quantum-based algorithm for generalized Ashenhurst-Curtis decomposition.	135
7.1.2 A new quantum algorithm for vertex graph coloring based on domination to solve the column multiplicity problem.....	136
7.1.3 Implementation of memristive finite state machines.....	137
7.1.4 Development of Memristor-based Components for a Pulse Rate System.....	138
7.2 Conclusion.....	139
References.....	141

List of Tables

Table 1 The truth table of incompletely specified function F1.....	16
Table 2 The truth table of F2	20
Table 3 a. Block {2,4,7} is encoded as 1, others as 0. b. Function G.....	20
Table 4 .a. X1, X2 and G are the inputs to function H b. Function H	21
Table 5 The truth table of an incompletely specified function F3	25
Table 6 Numbers of qubits used to implement the Grover's algorithm using Oracle 1 ...	45
Table 7 Numbers and types of quantum gates used to implement the Grover's algorithm using Oracle 1. We convert all the n-bits Toffoli gates to 3 by 3 CCNOT gate and 1 X gate with ancilas for gates count in the table	46
Table 8 Numbers of qubits used to implement the Grover's algorithm using Oracle 2 ...	46
Table 9 Number and types of quantum gates used to implement the Grover's algorithm using Oracle 2	46
Table 10 Simulation results for Algorithm 2 with single threshold value.....	48
Table 11 Estimation of Qubits required to construct Oracle for benchmark functions	51
Table 12 Table. 4.1. Table for encoding of a graph in Fig. 24.	62
Table 13 gate count for some case in graph coloring benchmark.....	72
Table 14 The operator of row compatibility	91
Table 15 The operator of columns compatibility.....	92
Table 16 The relation of rows absorption	92
Table 17 The relation of columns absorption	92
Table 18 The pulse count of permissive state machine in benchmark circuits from MCNC and ISCAS.	110

Table 19 The Excitation table of the JK Flip Flop.....	122
Table 20 the truth table of Rate Counter.....	128

List of Figures

Fig. 1 Diagram for Grover’s algorithm.....	5
Fig. 2 Grover iteration G.....	6
Fig. 3 The classical oracle on the left and the quantum oracle on the right.....	7
Fig. 4 Diagram of the generalized Ashenhurst-Curtis decomposition of function $F(x)$. (a) is $F(x)$ before decomposition and (b) are the sub-functions after decomposition.	12
Fig. 5 Flowchart for finding the optimal ΠG by the exhaustive search on a classical oracle.	22
Fig. 6 Circuit schematic for function $EQ(a_1, a_2)$. Inputs a_1 and a_2 are n -bit bus each....	28
Fig. 7 The boolean circuit implementation of refinement test function $R(P(A), P(B))$. Inputs a_1, a_2, a_3, a_4 and a_5 and b_1, b_2, b_3, b_4, b_5 are 5-bit bus each.....	28
Fig. 8 Grover Oracle for finding ΠG	30
Fig. 9 Grover iteration G including diffusion operator.....	31
Fig. 10 Quantum circuit for $EQ(a_1, a_2) \rightarrow EQ(b_1, b_2)$. Operations (1), (2) and (3) are drawn in parallel to save space. But, in fact, they operate in sequence not in parallel.....	32
Fig. 11 Quantum refinement test for partition of projective minterms with 3 elements. $R = (EQ(a_1, a_2) \rightarrow EQ(b_1, b_2)) \cdot (EQ(a_2, a_3) \rightarrow EQ(b_2, b_3)) \cdot (EQ(a_1, a_3) \rightarrow EQ(b_1, b_3))$	33
Fig. 12 bitwise XNOR on the concatenation of $P(A)$ and $P(C)$. The mirror circuit is not shown.	33
Fig. 13 Classical and quantum hybrid system for decomposition.	34

Fig. 14 Partition Counter and Threshold Checker are added, such that Oracle 2 tests both validities of the derived partition and the number of blocks.	35
Fig. 15 Quantum Partition Counter and Threshold Checker.	36
Fig. 16 Quipper code for a block structure and an example shows calling the subroutine to construct a block of circuits.	40
Fig. 17 Equivalence test circuit is constructed by calling the subroutine EQ.....	41
Fig. 18 Use for loop to describe Grover iterations.....	42
Fig. 19 An example of using recursion to describe a parameterized control counter.....	43
Fig. 20 A schematic of quantum circuit implementing one Grover's iteration for decomposition of function with three minterms generated by Quipper.	44
Fig. 21 Demonstration of node A cover node E.	57
Fig. 22 Demonstration of E node E is removed from the graph of Figure 21, together with all adjacent edges.	58
Fig. 23 Example demonstration of Graph coloring algorithm based on domination.....	61
Fig. 24 Example graph.....	62
Fig. 25 Block diagram for the oracle.	63
Fig. 26 block diagram for quantum counting. (a) is Hadamard, (b) is Grover iterator (c) quantum Fourier transform.	64
Fig. 27 quantum multiplexer 1.....	65
Fig. 28 4:1 Mux with half number of 1 and 0 as input.	66
Fig. 29 The left Mux is based on Shannon, and the right is based on Davio expansion. .	66
Fig. 30 quantum 2bit comparator and the encoding of output.....	67
Fig. 31 Hybrid algorithm for graph coloring.	68

Fig. 32 Quipper code example demonstrates abstraction.	69
Fig. 33 Quipper code example demonstrates map function.....	71
Fig. 34 gate count vs node count of a random generated graph.	72
Fig. 35 Memristor and its relation between the magnetic flux and the electric charge.	76
Fig. 36 I-V Characteristic of Memristor.	77
Fig. 37 Circuit implementation of IMPLY logic gate.....	78
Fig. 38 A crossbar with memristor.	78
Fig. 39 MsFPGA architecture.	79
Fig. 40 one pulse corresponds to one imply gate in that given pulse period. Before t_1 , wm1 and wm2 are reset to 0. At t_1 , the wm1 becomes a' , because $wm1 = a$, $wm1 = 0+a'$	80
Fig. 41 rectangles represents value transition of a memristor cell.....	82
Fig. 42 I is the memristor cell I store input, and cell S store current state, cell O store the output.	83
Fig. 43 Cell in green demonstrates the input cells of the output function λ , and the blue cell shows the output cell.....	84
Fig. 44 (a) state transition diagram for a given FSM, (b) the memristor realization represented with ISD.....	85
Fig. 45 encode the next state in the transition table with partition.	97
Fig. 46 encode the current state with one-hot code.	97
Fig. 47 encodes the current state with one-hot code.....	98
Fig. 48 demonstration of applying row compatibility.	98
Fig. 49 Union P1, P2 and P3.....	99

Fig. 50 Demonstration of deleting the G-implicants.	100
Fig. 51 ISD for excitation function.	101
Fig. 52 cost computation with synthesis for an excitation function.	103
Fig. 53 Step 1.	106
Fig. 54 Step 2.	106
Fig. 55 Step 3.	107
Fig. 56 Step 4.	107
Fig. 57 Step 5.	108
Fig. 58 Step 6.	108
Fig. 59 step 7.	109
Fig. 60 Curve plot for table 18.	111
Fig. 61 Pulse rate signal with clock.	113
Fig. 62 Rate multiplier.	114
Fig. 63 Reversible counter.	115
Fig. 64 Blocks connected for multiplication $Z= A*B$	116
Fig. 65 Blocks connected for square root $Z=\sqrt{A}$	116
Fig. 66 addition of two pulses A and B.	117
Fig. 67 Blocks connected for addition $A+B = Z$	117
Fig. 68 Blocks connected for Linear Differential Equations.	118
Fig. 69 the circuit to solve nonlinear system of two algebraic equations $x + y= B$, $x*y= A$	119
Fig. 70 the basic behavior of 3-bit Rate Multiplier.	120
Fig. 71 The rate generator and one bit of submodule from the counter.	120

Fig. 72 The transform diagram of JK Flip Flop.....	122
Fig. 73 the transform unit behavior circuit without clock.	124
Fig. 74 An example for the imply gate representation.....	124
Fig. 75 clear gate symbol.....	125
Fig. 76 The unit circuit of 3-bit Rate Multiplier for bit 0 position.	125
Fig. 77 The unit circuit of 3-bit Rate Multiplier for bit 1 position.	126
Fig. 78 The unit circuit of 3-bit Rate Multiplier for bit 2 position.	126
Fig. 79 the black block of the Rate Counter.	127
Fig. 80 the 3-bit binary down and up counter.....	128
Fig. 81 the basic behavior circuit of the 3-bit Rate Counter.....	129
Fig. 82 the behavior circuit of the up-down selector.	129
Fig. 83 Calculating for all selected condition C0, C1 and C2 part of the memristor-level circuit of 3-bit Rate Counter.	131
Fig. 84 Calculating the JK for each bit part of the memristor-level circuit of the 3-bit Rate Counter.....	132
Fig. 85 Calculating the JK _n for each bit part of the memristor-level circuit of the 3-bit Rate Counter.	132
Fig. 86 Calculating the new X for each bit part of the memristor-level circuit of the 3-bit Rate Counter.	133
Fig. 87 Block diagram for hybrid memristor circuit for pulse rate adder.....	133

Chapter 1 Introduction

1.1 Overview of quantum computing

Quantum computing leverages the principles of quantum mechanics, such as superposition and entanglement, to process information. Unlike traditional computers that use bits to represent and manipulate data, quantum computers employ qubits. These qubits allow for the simultaneous representation and manipulation of multiple values, enabling quantum computers to perform certain tasks much faster than traditional computers. There is potential for quantum computers to solve problems that classical computers cannot tackle, such as cracking specific encryption methods or simulating intricate quantum systems. Despite these promising capabilities, quantum computers are still in the early stages of development and face numerous technical hurdles before they can be widely adopted.

Quantum computers have the potential to perform algorithmic and computational tasks that cannot be achieved by classical computers due to their unique properties such as superposition, interference, and entanglement. One example of this is Grover's Algorithm. This dissertation presents methods for mapping Functional Decomposition based on partition calculus into Grover-based quantum algorithms that provide quadratic speedup. A library of quantum circuit blocks with basic functionalities such as counters, adders, and multiplexers was developed in this dissertation, and a bottom-up approach was proposed for building quantum oracles using these functional blocks. These techniques were demonstrated using problems such as Ashenurst-Curtis decomposition and graph coloring. These approaches have the potential to enable massively parallel computing

schemes, potentially surpassing any transistor-based supercomputers in terms of computing power.

1.2 Overview of memristor circuit

A memristor is a non-volatile, nanoscale memory device capable of performing stateful logic operations, allowing for in-memory computing and reducing data transfer delays between the processor and memory. Its low power consumption and compact circuit size make it ideal for use in embedded systems such as edge computing and robotics. Our team proposed a memristor-based FPGA circuit architecture. In this dissertation, I introduced memristor-based pulse rate computing components and a pulse rate computing system based on a memristive FPGA. The core concept behind pulse rate computing is to represent a number with a pulse signal in a single bit, simplifying the circuits for algebraic operations like multiplication and solving differential equations. By developing a pulse rate computing platform with memristors, I was able to further reduce power consumption and circuit size.

Chapter 2 Quantum computing

2.1 Quantum computing

Quantum computing is an emerging technology with a high rate of growth in the digital world today. It combines principles of quantum physics and computing to solve problems that are too complex for classical computers. A quantum computer leverages the unique properties and phenomena of quantum mechanics, such as entanglement and superposition, to achieve significant advances in processing power. However, the current state of quantum computing is similar to the early stages of classical computing in the 1950s, with a focus on hardware development and the use of low-level languages for programming. Many companies and institutions, including Google, IBM, Microsoft, Intel, DWAVE, and startups, are investing in quantum computing research.

2.2 Quantum circuits and gates

In quantum computing, a quantum gate is a basic quantum circuit that performs a specific operation on one or more qubits. Quantum gates are the building blocks of quantum circuits, which are collections of quantum gates that are used to perform specific tasks. Quantum circuits are the quantum equivalent of classical digital circuits, which are used to perform logical operations in classical computers.

Quantum gates can be used to perform various operations on qubits, such as not (negation), (conjunction), or (disjunction), and controlled-not (CNOT). These operations can be combined in various ways to perform more complex tasks, such as quantum teleportation and quantum error correction.

Quantum circuits can be represented using a diagram called a quantum circuit diagram, which shows the flow of qubits through the circuit and the operations performed on them at each step. Quantum circuit diagrams are used to design and analyze quantum algorithms and to implement them on quantum computers.

The model of quantum circuits is capable of representing a large number of quantum algorithms and is similar to classical digital circuits, with wires that transmit data through logic gates. However, because quantum mechanics describes a reversible process, quantum circuits must be constructed exclusively from reversible gates. As a result, the model of quantum circuits is more closely related to classical reversible circuits.

2.3 Grover's algorithm

Grover's algorithm is one of the most famous quantum algorithms [22-24]. Every particular problem for it is represented as a logic oracle. Grover's algorithm performs searching on a "black box," an unsorted dataset, to find an element (a minterm of a Boolean function describing the problem) that satisfies the oracle. The oracle is explicitly built for the given problem. The idea of Grover's algorithm is to place the quantum bits (qubits) representing the entire search space of size N in a superposition state. Then the phase of the states marked by the oracle is inverted, followed by an inversion of the mean operation, also known as the diffusion operation. Diffusion operation amplifies the amplitude of the marked states to increase the probability that this state will result from measurement performed on a vector of input qubits. (Diffusion operator is described in

detail by various papers [22-24]). After $O(\sqrt{N})$ operations, the probability of measuring the target solution approaches 1 [22].

Grover's algorithm operates as shown in Fig.1. The algorithm begins with putting the input qubits S , representing the search space, in the equal superposition state of all possible minterms for the size of the problem by applying a vector of Hadamard gates. Then a quantum operator called Grover's iteration, denoted by G , is repeated. After \sqrt{N} iterations of G , the solution is measured on qubits S . The measured result is a midterm binary vector given to the verification algorithm. If the result is correct, then it is outputted as a solution. Otherwise, the algorithm is repeated. The probability of correct measurement is very high.

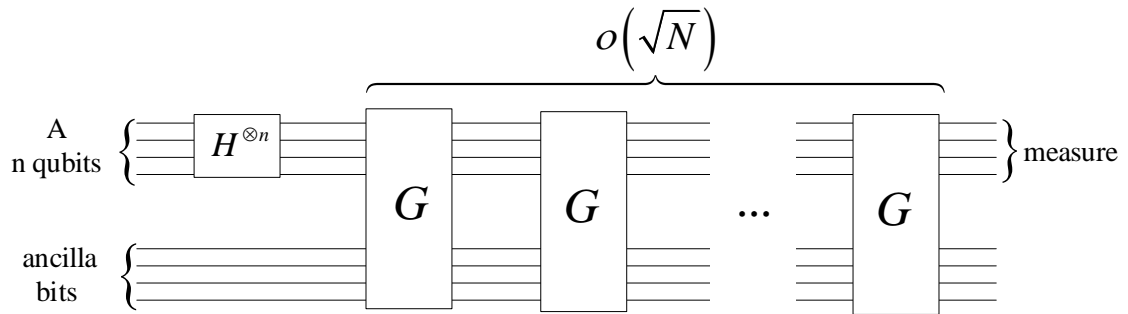


Fig. 1 Diagram for Grover's algorithm.

The Grover iteration, whose quantum circuit is illustrated in Fig.2. G consists of four steps:

1. Execution of Oracle f .
2. Apply the first set of Hadamard gates to perform the Hadamard transform.
3. Performs a conditional phase shift in the search space.

4. Apply the second set of Hadamard gates to perform the Hadamard transform.

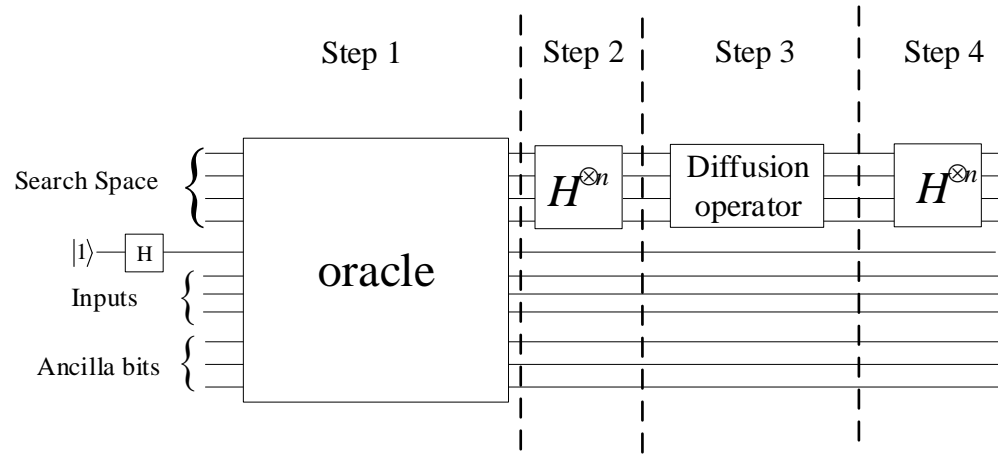


Fig. 2 Grover iteration G.

In step one, the oracle is applied to all the qubits of the system, including the search bit qubit representing the search space, the qubit stored input data, and the extra auxiliary qubits called ancilla qubits. After steps 2 to 4, the two Hadamard gates and the phase shift gates apply the diffusion operator to the qubits representing the search space.

Like the classical oracle, a quantum oracle is a quantum circuit that implements a decision function $f(x)$. However, there are the following differences. First, the inputs to the quantum oracle are in a superposition state of all possible binary vectors, while the inputs to the classical oracle are sequentially provided all binary vectors from the solution space. Second, in contrast to a classical oracle, an arbitrary boolean circuit, a quantum oracle for Grover's algorithm must be entirely built from reversible quantum gates (usually using ancilla bits). Any data input and ancilla qubits that are modified during the computation will need to be restored to their initial constant values by a mirror inverse block of $f(x)$. Since the oracle is repeated for each iteration, the data input and ancilla

qubits must be restored for the next iteration. A Quantum oracle designed using in a mirror structure is shown in Fig.3.

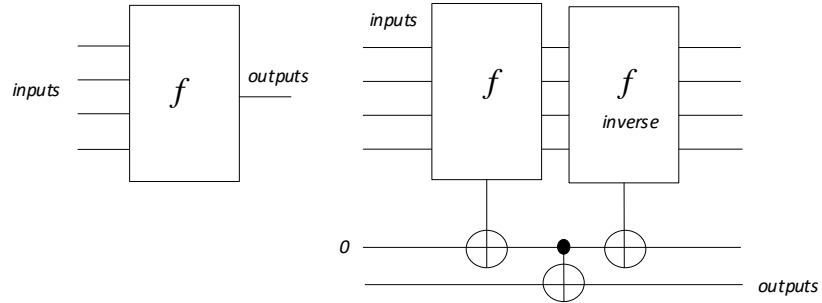


Fig. 3 The classical oracle on the left and the quantum oracle on the right.

The original algorithm proposed by Grover is to assume a unique solution that satisfies the desired condition. Grover briefly considered the situation of multiple solutions and stated that it could be achieved by modifying the number of Grover iterations, but he did not provide any detailed implementation and analysis. Boyer et al. [43] expanded and generalized Grover's algorithm to k-solutions [43,44]. They provided a tight analysis of Grover's algorithm and proposed a new algorithm to find a solution when the number of solutions is more than one and unknown ahead of time. Younes et al. [44] verified the algorithm and observed that the algorithm only works for $1 \leq k \leq 3N/4$, and k is the number of solutions. Their algorithm is shown as follows:

1. Initialize $m = 1$ and set $\lambda = 6/5$. (λ is between 1 and $4/3$)
2. Pick an integer $0 < j < m-1$ uniformly and randomly among the non-negative integers smaller than m.
3. Apply j times of Grover iterations G.

4. Measure the register: let i be the outcome.
5. If $f[i] = 1$, then the solution is found and exits.
6. Otherwise, set m to $\min(\lambda m, \sqrt{N})$ and go back to step 2.

The expected time complexity of this algorithm is $O(\sqrt{N/k})$ [43,44]. This algorithm is commonly used for solving problems with multiple solutions. It is used in my Grover's Algorithm for Generalized Ashenhurst-Curtis decomposition which will be introduced in Chapter 3.

Chapter 3 Grover's algorithm for generalized Ashenhurst-Curtis decomposition

"Grover-based Ashenhurst-Curtis Decomposition Using Quantum Language Quipper",
Quantum Information & Computation vol. 19, pp. 35-66. February 2019.

Yiwei. Li, Edison. Tsai, Marek Perkowski, Xiaoyu Song,

Authors' Contributions:

Yiwei, Li: Conceptualization, methodology, experiment implementation, data collection, data analysis, and writing of the manuscript.

Edison Tsai: Provide the idea of quantum gate implementation of partition operations.

Marek Perkowski: Review of the manuscript.

Xiaoyu Song: Supervision and reviewed the manuscript.

3.1 Generalized Ashenhurst-Curtis decomposition

Functional decomposition transforms a description of a large logic function (possibly incompletely specified) into a network of smaller sub-functions that are realized by specific gates. The decomposition converts, therefore incompletely specified functions to completely realized functions when realized in hardware. The function can be Boolean, Multi-valued, or continuous, but in this dissertation, I will be restricted to binary Boolean functions. The decomposition of a large function not only reduces the complexity but also increases the scalability and improves the reliability of the circuit implementing the function [1]. In some instances, it also improves the “understandability” of the function, because it is easier for a human to understand a network of known simple functions than a large function being a mapping. Therefore, functional decomposition is widely adopted in the implementation of cost-effective ASIC and FPGA design. The best-known variants of functional decomposition were created by Ashenhurst and Curtis. In addition to the digital circuit design, the decomposition methods find many applications in other areas, such as compression in databases and data privacy protection for smart grids [2-5]. The most important possible future application of this method is in the new area of Quantum Machine Learning [48-50]. In Machine Learning, the decomposition method, besides being used as a classifier [47], is applied to discover some hidden properties of the data and decompose decision rules [2], [35]. Ashenhurst-Curtis decomposition is also used to create decomposed structures for neural networks [45,46]. Unfortunately, the high computational complexity of this decomposition does not allow it to be used for large data that are typical for deep learning. Many attempts have been undertaken to create efficient data structures and algorithms to speed up the Ashenhurst-Curtis Decomposition

[9,10]. This dissertation presents a new problem formulation for functional decomposition based on partition calculus (partition algebra) [18], which enables the application of the quantum Grover's search algorithm to the decomposition problem, thus providing a quadratic speedup with respect to the standard software algorithm. Due to the exclusive properties of the quantum system, like superposition, interference, and entanglement, Grover's search algorithm offers quadratic speedup compared to its classical counterparts in terms of the number of evaluations of oracle. However, the efficiency of the oracle itself is also important, and this will be dealt with in this dissertation.

Moreover, some extensions to Grover give better than quadratic speedup, taking into account certain specific properties of the problem [41]. Grover's algorithm is also used as a subroutine in other new quantum algorithms that give even higher speedups. Studying the fundamental Grover's algorithm for this problem can thus open the door to several improved approaches. Of course, these methods will become practical only in the future when quantum computers with many qubits become available.

Ashenurst-Curtis decomposition is one of the well-known decomposition methods in logic synthesis [9]. The Ashenurst-Curtis decomposition of function $F(X)$ can be written as

$$F(X) = F(A, B, C) = H(A, G(B, C), C), \quad (1)$$

where A , B , and C are disjoint subsets of the set of input variables X . Functions G and H are called the predecessor function and the successor function, respectively. The subset B

UC of input variables, on which only the predecessor function depends, is called the bound set, the subset AUC is called the free set. Subset C is the intersection of free and bound sets. If the set C of shared variables in decomposition is empty, then this decomposition is called a disjoint decomposition. Otherwise, it is a non-disjoint decomposition. In Ashenhurst's formulation [6], function G has one output, while in Curtis' formulation[7], function G can have more than one output but less than the number of variables in BUC. In both Ashenhurst's and Curtis's formulations [7], function H has a one-bit output. This dissertation uses a generalized form of Ashenhurst-Curtis decomposition illustrated in Fig.4, in which Function H can have an arbitrary number of outputs. Moreover, the inputs and outputs can be multi-valued. This form of decomposition is referred to as a generalized Ashenhurst-Curtis decomposition in this dissertation.

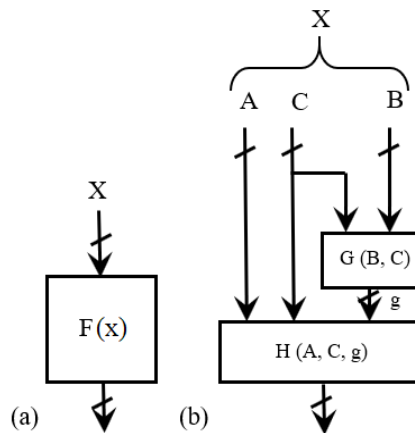


Fig. 4 Diagram of the generalized Ashenhurst-Curtis decomposition of function $F(x)$. (a) is $F(x)$ before decomposition and (b) are the sub-functions after decomposition.

The goal of Ashenhurst-Curtis decomposition is to find an optimal partition of the inputs set X and a predecessor function G such that the decomposition has the smallest cost, in

terms of the minimum total complexity of subfunctions G and H. The two primary steps required in the decomposition process are: partitioning input variables and computing the predecessor function $G(B, C)$. Once the two significant steps are executed, the derivation of the successor function is straightforward.

3.2 Functional decomposition problem formulation

The functional decomposition problem has been studied over the years in the field of logic synthesis. Many solutions have been proposed, which involve several combinatorial tasks [13-16]. One of the popular solutions was to find the minimum multiplicity index for the selected bound set. This problem has been reduced to one of the following sub-problems: minimum coloring of the incompatibility graph, compatibility graph maximum clique covering, compatibility graph maximum clique partitioning, and others [9]. All these problems can be solved with heuristic approaches or using theoretically exact minimum algorithms. The decomposition process is usually recursively applied to both the functions G and H until a network is constructed where each block can be directly implemented in a single logic cell, is non-decomposable, or satisfies some machine-learning conditions. In this dissertation, I will discuss only a single level of decomposition. Multi-level decomposition can be executed on a hybrid computer that stores intermediate results in a standard memory and uses a quantum computer for every particular decomposition.

Early decomposition algorithms are to compute column multiplicity on the decomposition charts [7]. Later, decomposition charts were replaced by cube representation [8]. Column multiplicity algorithms based on operations on arrays of cubes

were formulated [8]. Due to the size of decomposition charts and cube representations increasing exponentially, the previous algorithms were not feasible to decompose large practical data. In 1993, Lai, Pedram, and Vrudhula proposed one of the most successful decomposition algorithms [51]. Their general theory uses a new algorithm based on both binary decision diagrams, BDD and EVBDD representations, for generating the set of all bound variables that make the function decomposable. By constructing a BDD of the decomposed function, in which the bound set is on the top of the BDD, and checking the number of children below the cut line [51], the column multiplicity can be derived easily. The problem of finding an optimal free and bound set was converted into finding the best cut with perfect ordering [9]. The application of BDDs made it possible to develop decomposition algorithms for larger functions than the respective previous methods that used another representations of boolean functions. After that, many BDD-based decomposition algorithms have been proposed [56-59]. The BDD approaches perform well with disjoint decomposition and can be extended to non-disjoint decomposition. However, they have several limitations [52]: Firstly, BDD suffers from the memory explosion problem. In representing a Boolean function, a BDD can be large. It takes even more memory for multiple values and incompletely specified functions. Secondly, the partition of the variable needs to be specified ahead and cannot be automated as an integrated part of the decomposition algorithm. Thirdly, the non-disjoint decomposition cannot be handled easily. The decomposability needs to be analyzed by cases with an exponentially increasing number of shared variables. More recently, an SAT-based approach has been proposed [52]. The decomposability of a function is formulated as an SAT-solving problem using Craig interpolation [54] and functional dependency [55]. The

SAT-based approach naturally supports non-disjoint decomposition, and it works for large functions with 300 input variables, according to experimental results [52]. The downside of the SAT approach is the decomposed function has to be completely specified, and SAT expression does not support multi-valued functions. SAT approach will only be useful for logic synthesis applications since functions in Machine Learning are usually incompletely specified and with multi-valued inputs. In contrast, the decomposition method that I propose in this dissertation is universal, which means that it can be applied to entirely or incompletely specified single- or multiple-output functions.

Łuba [16] proposed an original function representation using partitions of sets. His partition-based representation provides a simple way to verify if a given predecessor function guarantees the existence of a decomposition. The innovative idea of Łuba allows the invention of simple and elegant solutions to many decomposition problems based on partition calculus. The definitions of incompletely specified Boolean functions and set partitions are introduced as follows. My approach presented here can be easily extended to other problems formulated in partition calculus.

Definition 1. An incompletely specified Boolean function F of n input variables and m output variables is defined as a mapping from a two-valued domain (0 and 1) to a three-valued domain (0, 1 and don't care) $F: \{0, 1\}^n \rightarrow \{0, 1, -\}^m$ [16].

Definition 2. The elements of the domain $\{0, 1\}^n$ of the function F are called minterms. A truth table is a list of minterms with the corresponding output values of the function. In the case of the incompletely specified function [16], the truth table does not include minterms for which all outputs are "don't care values."

Definition 3. Let M be the vector of minterms and Y be the vector of outputs in the truth table of function F . Let X be the set of input variables of F . Let $A \subseteq X$ and $m \in M$. The projection of minterm m with respect to set A is denoted by m_A , which is called a projective minterm induced by set A . M_A denotes the vector of all projective minterms m_A .

For example, a function F_1 with inputs variables x_1, x_2, x_3 , and x_4 is shown in Table 1. The vector of minterms M is $\langle 0011, 0100, 1100, 0010, 0011 \rangle$, and the vector of outputs Y is $\langle 01, 10, 01, 00, 10 \rangle$. Assume a subset of the set of input variables, A , is $\{x_1, x_2\}$, then the vector M_A of projective minterms induced by A is $\langle 00, 01, 11, 00, 00 \rangle$. In this paper, the vector of projective minterms M_A is used to represent the input variables that belong to either the free set or the bound set.

F	Inputs				outputs	
label	x_1	x_2	x_3	x_4	y_1	y_2
1	0	0	1	1	0	1
2	0	1	0	0	1	0
3	1	1	0	0	0	1
4	0	0	1	0	0	0
5	0	0	1	1	1	0

Table 1 The truth table of incompletely specified function F_1 . Note that unspecified minterms are not present. This property is useful, especially in Machine Learning, where we deal with functions of very many variables (attributes) but not that many minterms

Some projective minterms may have identical values. For example, in Table 1, assume that a subset of input variables A is $\{x_1, x_2\}$, then M_A is $\langle 00, 01, 11, 00, 00 \rangle$. The

projected minterms 00 in positions 1, 4, and 5 are identical. A set partition representation is proposed and used here to reflect the identity among projective minterms.

Definition 4. A partition π on set S is a collection of the disjoint non-empty subsets of S whose set union is S , i.e., $\forall(A_i, A_j) \in \pi, i \neq j$, such that $A_i \cap A_j = \emptyset$ for $\cup_{A_i \in \pi} (A_i) = S$ [17]. The elements of set π are referred to as blocks of π . The blocks are distinguished with bar and semi-colons when a partition is written out. For example, if $S = \{1, 2, 3, 4, 5, 6\}$, and partition π on S has blocks $\{1, 2\}$, $\{3, 4\}$ and $\{5\}$, then the partition is written as $\pi = \{\overline{1,2}; \overline{3,4}; \overline{5}\}$.

Theorem 1. If R is an equivalence relation on the set S , then the equivalence classes form a partition of S . Conversely, if π is a partition of set S , then there is an equivalence relation on S whose equivalence classes are elements of π (blocks) [15].

Vector of projective minterms or outputs can be partitioned into equivalence classes by an equivalence relation. According to Theorem 1, equivalence classes of a relation and a partition are equivalent. For a vector of projective minterms induced by A , $M_A = \langle m_{a1}, m_{a1}, \dots, m_{an} \rangle$, if $m_{ai} = m_{aj}, i \neq j$, then these two projective minterms are considered to belong to a single block in the partition of M_A . Otherwise, they belong to different blocks. The partition of M_A is called an input partition induced by set A and is denoted by $P(M_A)$. For example, consider M_A is $\langle 00, 01, 11, 00, 00 \rangle$, the partition of M_A , $P(M_A)$ is $\{\overline{1,4,5}; \overline{2}; \overline{3}\}$. The same rules are applied to derive the partition of output Y of function F . Partition of output Y is denoted by P_F . By using the partition representation of the function, Łuba [16] derived Theorem 2 for the “serial decomposition” which is called the

generalized Ashenhurst-Curtis decomposition in my terminology. Theorem 2 states a condition that guarantees the existence of decomposition with any given output partition of the predecessor function. Theorem 2 and the partition operators mentioned in Theorem 2 are introduced as follows.

Definition 5. Partition multiplication [17].

If π_1 and π_2 are partitions on S , then, $\pi_1 \bullet \pi_2$ is the partition on S such that, any block of $\pi_1 \bullet \pi_2$ is an intersection of some block from π_1 and with some block from π_2 .

$$B_{\pi_1 \bullet \pi_2}(s) = B_{\pi_1} \cap B_{\pi_2}$$

Example:

$$\pi_1 = \{\overline{1,2,3,4}; \overline{5,6}; \overline{7,8,9}\}, \pi_2 = \{\overline{1,6}; \overline{2,3,4}; \overline{5,9}; \overline{7,8}\}$$

$$\pi_1 \bullet \pi_2 = \{\overline{1}; \overline{2,3,4}; \overline{5}; \overline{6}; \overline{7,8}; \overline{9}\}$$

Definition 6. Refinement relation of partition [17].

Given a partition π_1 and π_2 on set S , if for all $U \in \pi_1$, there exists $V \in \pi_2$, such that $U \subseteq V$, then we say that π_1 is a refinement of π_2 , and π_1 is finer than π_2 [25]. This relation of partitions is a partial order, so it is denoted by “ \leq ”. In that case, π_1 is finer than π_2 , which is written as $\pi_1 \leq \pi_2$.

Example:

$$\pi_1 = \{\overline{1,2}; \overline{3,4}; \overline{5,6}; \overline{7}\}, \pi_2 = \{\overline{1,2,5,6}; \overline{3,4,7}\}$$

$$\pi_1 \leq \pi_2$$

Notation

Π_G is the partition of the output of the predecessor function G . Partition $P(\text{BUC})$ is the partition of the bound set, and $P(\text{AUC})$ is the partition of the free set. P_F is the partition representation of the output of function F .

Theorem 2. [16]. There exists an Ashenurst-Curtis decomposition of F if and only if there exists a partition Π_G , such that

$$(a) P(\text{BUC}) \leq \Pi_G \quad \text{and}$$

$$(b) P(\text{AUC}) \cdot \Pi_G \leq P_F.$$

Point (a) of Theorem 2 states the input partition of predecessor function G must be finer than output partition Π_G of predecessor function G . Point (b) of Theorem 2 states that the partition product of free set partition $P(\text{AUC})$ and predecessor output partition Π_G must be finer than output partition P_F of the function F . In such a case, function F can be decomposed into sub-functions G and H . During the decomposition process, after a partition Π_G that satisfies Theorem 2 has been found, the component functions G and H can be easily derived. For example, a truth table of function F_2 is given in Table 2. Variables $X_3, X_4,$ and X_5 are chosen as a bound set, while X_1 and X_2 are chosen as a free set. Assuming that an output partition of the predecessor function $\Pi_G = \{\overline{1,3,5,6,8,9,10,11}; \overline{2,4,7}\}$ is found, each block of the partition is encoded with a binary string (a code of this block). For example, block $\{1,3,5,6,8,9,10,11\}$ is encoded as 0 and block $\{2,4,7\}$ as 1 to generate a column G in Table 3a. By merging the duplicate rows in Table 3a Table 3b is derived, which represents the predecessor function G . By selecting free set inputs X_1, X_2 and predecessor output G as inputs, the truth table for the successor function H is derived and shown in Table 3.4.a. Further removing the duplicate rows, the

final truth table for H is created as shown in Table 3.4.b. Technically, by obtaining the functions G and H, the decomposition is completed. However, in some applications, these functions have to be converted to completely specified functions and further minimized. For instance, from Table 3b, a Boolean expression, $G = x_3 \cdot \overline{x_4} \cdot \overline{x_5} + x_4 \cdot x_5$ can be derived using a logic minimizer such as ESPRESSO [40]. Because this dissertation is related only to the decomposition problem, the preprocessing and postprocessing issues are not discussed.

	X ₁	X ₂	X ₃	X ₄	X ₅	F ₂
1	0	0	0	0	0	0
2	0	1	0	1	1	0
3	0	0	1	0	1	0
4	0	1	1	1	1	0
5	0	0	1	1	0	0
6	0	1	0	0	1	1
7	0	0	1	0	0	1
8	0	1	1	1	0	1
9	1	0	1	0	1	1
10	1	1	0	0	1	1
11	1	0	0	0	1	1

Table 2 The truth table of F₂

	X ₃	X ₄	X ₅	G
1	0	0	0	0
2	0	1	1	1
3	1	0	1	0
4	1	1	1	1
5	1	1	0	0
6	0	0	1	0
7	1	0	0	1
8	1	1	0	0
9	1	0	1	0
10	0	0	1	0
11	0	0	1	0

	X ₃	X ₄	X ₅	G
1	0	0	0	0
2	0	1	1	1
3	1	0	1	0
4	1	1	1	1
5	1	1	0	0
6	0	0	1	0
7	1	0	0	1

Table 3 a. Block {2,4,7} is encoded as 1, others as 0. b. Function G

	X_1	X_2	G	F_2
1	0	0	0	0
2	0	1	1	0
3	0	0	0	0
4	0	1	1	0
5	0	0	0	0
6	0	1	0	1
7	0	0	1	1
8	0	1	0	1
9	1	0	0	1
10	1	1	0	1
11	1	0	0	1

	X_1	X_2	G	H
1	0	0	0	0
2	0	0	1	1
3	0	1	0	1
4	0	1	1	0
5	1	0	0	1
6	1	1	0	1

Table 4 .a. X_1, X_2 and G are the inputs to function H b. Function H

3.3 Grover algorithm for Ashenhurst-Curtis decomposition

3.3.1 Finding optimal Π_G by search with an oracle

With the property provided by Theorem 2, an optimal Π_G for decomposition can be found by using a search with an oracle. An oracle for decomposition is simply a decision function that validates a given partition Π_G and produces an output depending on whether Π_G satisfies both conditions of Theorem 2. More precisely, if decomposition exists for a given Π_G , then a classical oracle outputs 1, otherwise, the oracle outputs 0. In this dissertation, Π_G is defined as valid if it guarantees the existence of a decomposition. We propose a method to find an optimal Π_G using an exhaustive search algorithm based on Grover's algorithm with a corresponding quantum oracle that verifies the condition of Theorem 2. We not only want to find a valid solution but also an exact optimal solution. The optimal solution can be found by minimizing the value of a cost function. The cost function is varied for different applications of functional decomposition. In Machine Learning, the minimum number of blocks in Π_G is required, while in circuit design, the minimum number of gates to implement the decomposed circuits is expected.

Below we introduce a theoretical algorithm with a classical oracle, a quantum and classical hybrid equivalent. To apply a search algorithm for optimization problems, the oracle test should be repeated with values of a cost function compared to different threshold values. The flowchart of the algorithm to find the optimal Π_G is presented in Fig.5. The search process in Fig.5 enumerates all possible candidates for the solution partition Π_G . Each candidate in the search space is verified by the oracle. If the output of the oracle is 1, which means a decomposition exists, then the found partition is passed to the next process in the flowchart. If the oracle returns 0, then the found partition is discarded, and the search process will create the next partition. The block number of each valid partition is then compared against a threshold value Max. If the block number of this partition is equal or larger than Max, then it is discarded while partition enumeration resumes; if the block number is less than the threshold, the threshold Max is reduced by one and the found partition is saved on a stack of solutions. After enumerating through all possible partitions, the partition saved on the top of the stack is the optimal Π_G , which guarantees the existence of the decomposition with the minimum number of blocks.

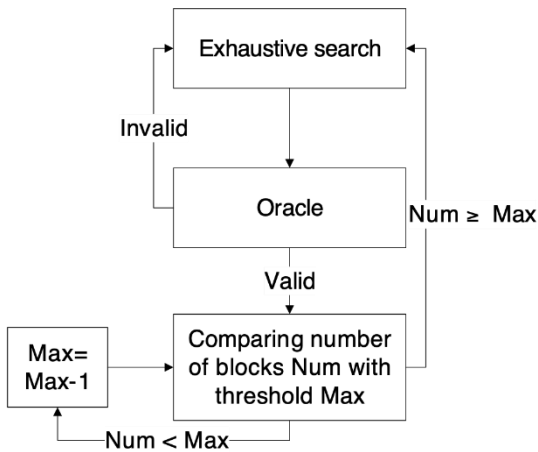


Fig. 5 Flowchart for finding the optimal Π_G by the exhaustive search on a classical oracle.

However, this exhaustive search method is not efficiently implemented with the classical computational model. This dissertation proposes to use a quantum search algorithm called Grover's algorithm [20, 21] to accelerate the enumeration process inside the search. Performing an exhaustive search on an unstructured search space using classical computation has the complexity of $O(N)$, where N is the number of data in the search space. In contrast, for Grover's algorithm, it was proved that any search could be computed with the complexity of $O(\sqrt{N})$ and the probability of correct solution very close to 1 [21]. Therefore, assuming a sufficient number of qubits, a quantum computer using a quantum oracle can solve the decomposition problem by exhaustive search with a quadratic speedup compared to a classical computer.

3.3.2. Proposed methodology to construct quantum oracle for decomposition

In order to implement Grover's algorithm for finding a valid Π_G , the result from Theorem 2 is adopted as the conditions to identify whether a given Π_G guarantees the existence of decomposition. To construct a quantum oracle that satisfies Theorem 2, i.e., $f(x) = (P(BUC) \leq \Pi_G) \text{ AND } (P(AUC) \cdot \Pi_G \leq P_F)$, we must be able to represent the partitions and operators on partitions using the reversible circuits [37-39] constructed by us. Boolean logic and logic function minimization have been well-known in engineering for many years. Every Boolean circuit can be converted to a reversible circuit, usually with additional ancilla qubits. My methodology is to create a classical Boolean circuit for each functional block in the oracle, and next convert each of these blocks to a reversible circuit, usually adding ancilla qubits. By combining various reversible sub-circuits, the resulting circuit is next realized in quantum technology to allow superposition and

entanglement, which would not be possible if the circuit were realized in classical technology such as CMOS.

To apply the partition calculus [18] to create Grover's oracle, we propose an encoding scheme that is general to all partition calculus problems. Each block of a partition is encoded as a binary vector. The partition calculus operations and relations are converted to boolean functions in proposed encoding. These functions are next converted to reversible circuits.

3.3.3 Encoding scheme for the partitions

In the encoding scheme, the projective minterm partitions or output partitions are represented with a vector of binary vectors, each of a minimum code length. For example, given function F_3 in Table 5 with a set of minterms, $M = \langle 00110, 00111, 11001, 00100, 11001 \rangle$, which M is a vector of binary vectors, each binary vector being a minterm. The projective vector of minterms M induced by a set of input variables $A = \{X_1, X_2, X_3, X_4\}$, is denoted by M_A i.e. $M_A = \langle 0011, 0011, 1100, 0010, 1100 \rangle$. The partition of M_A is denoted by $P(M_A) = \{\overline{1,2}; \overline{3,5}; \overline{4}\}$, where labels 1 and 2 correspond to vector 0011, labels 3 and 5 correspond to vector 1100, and label 4 corresponds to vector 0010. Projective partition $P(M_A)$ has only three distinct blocks while the bit size of the single vector in M_A is four bits, which is larger than the minimum number of bits to distinguish three blocks. In order to minimize the number of bits used to represent each block of $P(M_A)$, partition $P(M_A)$ is encoded with a vector of binary strings. The encoding procedure consists of two steps. First, all minterms are arranged with numerical order from the truth table. Then, each unique minterm is assigned with a binary encoding.

Every duplicate projective minterm is encoded with the same binary code. For example, the first block {1, 2} of M_A is encoded as 00, block {3, 5} as 01 and block {4} as 10. By N_A we denote the vector of binary vectors representing the partition $P(M_A)$, $N_A = \langle 00, 00, 01, 10, 01 \rangle$. The vector of binary vectors N_A is converted to a single binary vector $\langle 0000011001 \rangle$, when it represents the input partition to Grover's based decomposition algorithm.

F_3	input					output
label	x_1	x_2	x_3	x_4	x_5	y
1	0	0	1	1	0	0
2	0	0	1	1	1	1
3	1	1	0	0	1	0
4	0	0	1	0	0	0
5	1	1	0	0	1	1

Table 5 The truth table of an incompletely specified function F_3

3.3.4 Representation of partition multiplication using boolean encodings

A multiplication of two partitions can be derived by concatenating each element from two vectors. For example, given two partitions $P(M_A) = \{\overline{1,4,5}; \overline{2}; \overline{3}\}$ and $P(M_B) = \{\overline{1,2}; \overline{3,4,5}\}$, the binary representations are $N_A = \langle 00, 01, 11, 00, 00 \rangle$ and $N_B = \langle 01, 01,$

00, 00, 00 >, respectively. Every two elements from N_A and N_B are concatenated in order to form a new element in vector form, i.e. $N_A \cdot N_B = \langle N_{A1}N_{B1}, N_{A2}N_{B2} \dots N_{Ai}N_{Bi} \rangle = \langle 0001, 0101, 1100, 0000, 0000 \rangle$. The reader can verify this notation by converting binary representation $N_A \cdot N_B$ to partition $P(M_A) \cdot P(M_B) = \{\bar{1}; \bar{2}; \bar{3}; \overline{4,5}\}$.

3.3.5 Testing of refinement relation with boolean function

A Boolean function R for testing refinement relation between two given encoded partitions is introduced in this section. The inputs to R are partitions $P(A)$ and $P(B)$, that both are partitions of the same set S . Function R outputs 1 if $P(A)$ is finer than $P(B)$, i.e., $P(A) \leq P(B)$. Otherwise, it outputs 0. According to Definition 6, if every block within $P(A)$ is a subset of one block within $P(B)$, then $P(A)$ is finer than $P(B)$. Since it is difficult to evaluate the containment relation between arbitrary two blocks within each partition, instead of comparing blocks for inclusion, we evaluate the relation between every pair of elements in both partitions $P(B)$, and $P(A)$. The following Theorem 3 states the necessary condition for the refinement relation regarding relations between each pair of elements.

Theorem 3 Consider $P(A)$ and $P(B)$ are partitions of set S . $P(A)$ forms equivalence relation R_A and $P(B)$ forms equivalence relation R_B .

If for all $x_i, x_j \in S, i \neq j, (x_i, x_j) \in R_A$

$\Rightarrow (x_i, x_j) \in R_B$

then $P(A) \leq P(B)$.

In other words, Partition $P(A)$ is finer than $P(B)$, if every pair of elements (x_i, x_j) , belonging to one block of $P(A)$, implies that this pair (x_i, x_j) is included in a single block of $P(B)$.

Proof

Consider an equivalence class $[x]_A$, by hypothesis, for any $y \in [x]_A$, $(x, y) \in R_A \Rightarrow (x, y) \in R_B$, thus each equivalence class $[x]_A$ is contained in an equivalence class $[x]_B$. According to Theorem 1, equivalence classes of a relation and a partition formed by this relation are equivalent, each block of partition is corresponding to an equivalence class in the equivalence relation. So, each block of $P(A)$ is contained in a block of $P(B)$, by definition, $P(A) \leq P(B)$.

According to Theorem 3, an implementation of function R is to check all possible pairs (x_i, x_j) in $P(A)$ and $P(B)$ such that for every pair of elements (x_i, x_j) , which belongs to the same block in $P(A)$, it is implied that pair (x_i, x_j) also belongs to a single block in $P(B)$. The Boolean logic implementation of R is defined as follows: $R = \Pi(\text{EQB}(x_i, x_j, P(A)) \rightarrow \text{EQB}(x_i, x_j, P(B)))$, where, Π denotes the product of every imply logic operator for all combinations of i and j . Boolean function EQB determines if elements from a given pair of elements (x_i, x_j) are both included in the same block in partition P , where P denotes either $P(A)$ or $P(B)$ in above function R . Within the encoding of partitions, if two projective minterms belong to the same block, they are assigned to the same code. So, function EQB is realized in a circuit that checks if two partition encodings are equivalent. The well-known Boolean implementation of equivalent function EQ is a product of all bitwise XNOR operations, shown in Fig.6. By substituting EQB with EQ , the function R

can be written as $R(P(A), P(B)) = \Pi(EQ(a_i, a_j) \rightarrow EQ(b_i, b_j))$. The Boolean circuit of $R(P(A), P(B))$ is shown in Fig.7, where inputs $P(A)$ and $P(B)$ are encoded partitions. The imply operator is realized with an imply gate using well-known formula $a \rightarrow b = \bar{a} + b$. By encoding partition in binary and applying Boolean operation that realizes partition operation, the oracle for finding valid Π_G , $f(x) = (P(BUC) \leq \Pi_G) \text{ AND } (P(AUC) \cdot \Pi_G \leq P_F)$ can be implemented in a Boolean function:

$$f'(x) = R(P(BUC), \Pi_G) \text{ AND } R(P(AUC) \cdot \Pi_G, P_F). \quad (2)$$

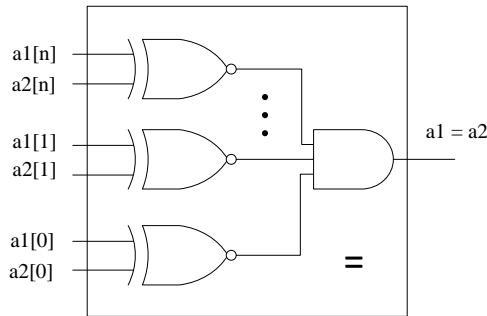


Fig. 6 Circuit schematic for function $EQ(a1, a2)$. Inputs $a1$ and $a2$ are n -bit bus each.

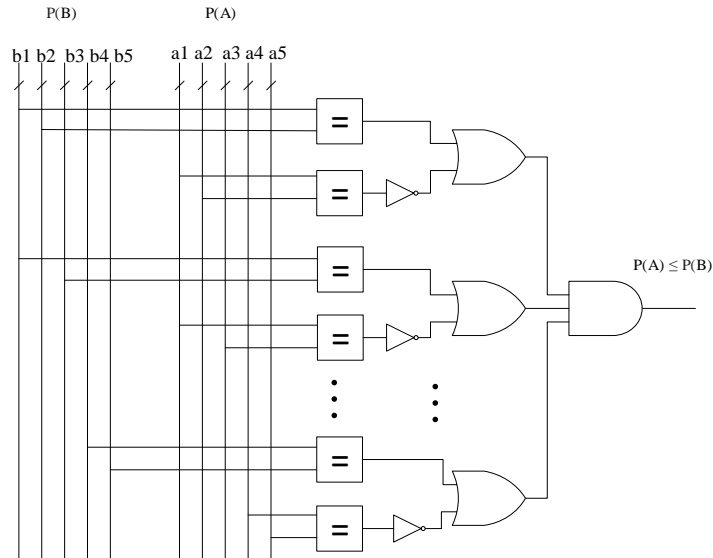


Fig. 7 The boolean circuit implementation of refinement test function $R(P(A), P(B))$. Inputs $a1, a2, a3, a4$ and $a5$ and $b1, b2, b3, b4, b5$ are 5-bit bus each.

3.3.6. Oracle synthesis for decomposition problem with known free and bound sets

I designed two variants of oracles with different sets of inputs, Oracle 1 and Oracle 2. I refer to the oracle with known free set partition $P(A \cup C)$ and bound set partition $P(B \cup C)$ as Oracle 1. The oracle function $f(x) = R(P(B \cup C), \Pi_G)$ AND $R(P(A \cup C) \cdot \Pi_G, P_F)$ is converted to a reversible circuit, which sets the output qubit to 1 when Π_G is valid and to 0 when Π_G is not valid with given free and bound sets.

The block diagram of the proposed quantum oracle is shown in Fig.8. The oracle consists of four major blocks. Blocks (1) and (2) in Fig.8, perform refinement test on two pairs of partitions $(P(B \cup C), \Pi_G)$ and $(P(A \cup C) \cdot \Pi_G, P_F)$. Block (3) and (4) are the inverse circuits, corresponding to blocks (1) and (2) to restore all the modified qubits to their original values which means that the ancilla and input partition qubits are restored to the original constants. The outputs of blocks (1) and (2) are stored in ancilla qubits i and j . Then, a Toffoli gate is applied on ancilla bit k controlled by i and j . If both i and j are 1, which means both conditions are met, ancilla bit k will be flipped. It changes the solution phase so that the solutions that satisfy both conditions can be marked. Because of the vector of Hadamard gate before the oracle; this corresponding to the quantum value -1 for Boolean value 1 and the quantum value 1 to the Boolean value 0. After being marked, the diffusion operator performs phase shift on the qubit representing Π_G , such that the amplitudes of the solutions increase while the amplitudes of the non-solution states decrease in each iteration. After repeating $\pi\sqrt{2^n}/4$ times of the Grover's iteration, where n is the number bits to represent the search space, the Π_G that meets both conditions from Theorem 2 is measured as a binary vector with high probability [21].

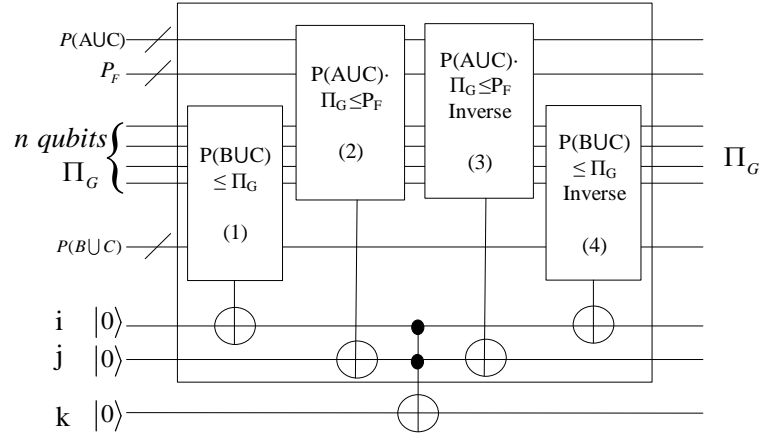


Fig. 8 Grover Oracle for finding Π_G .

The inputs of the oracle are encoded as a single binary vector: free set partition $P(AUC)$, bound set partition $P(BUC)$ and output partition $P(F)$. For example, given a function in Table 5, x_1, x_2 are selected as A, x_4, x_5 as B, and x_3 as C. Then $P(AUC)$ is $P(x_1x_2x_3) = \{\overline{1,2}; \overline{3,5}; \overline{4}\}$, which is encoded as $\langle 0000011001 \rangle$, $P(BUC)$ as $\langle 0001101110 \rangle$ and $P(F)$ as $\langle 01001 \rangle$. At the beginning of the algorithm, these three binary coded partitions are set as the inputs to the quantum oracle and the qubits representing search space Π_G are initialized with Hadamard gates.

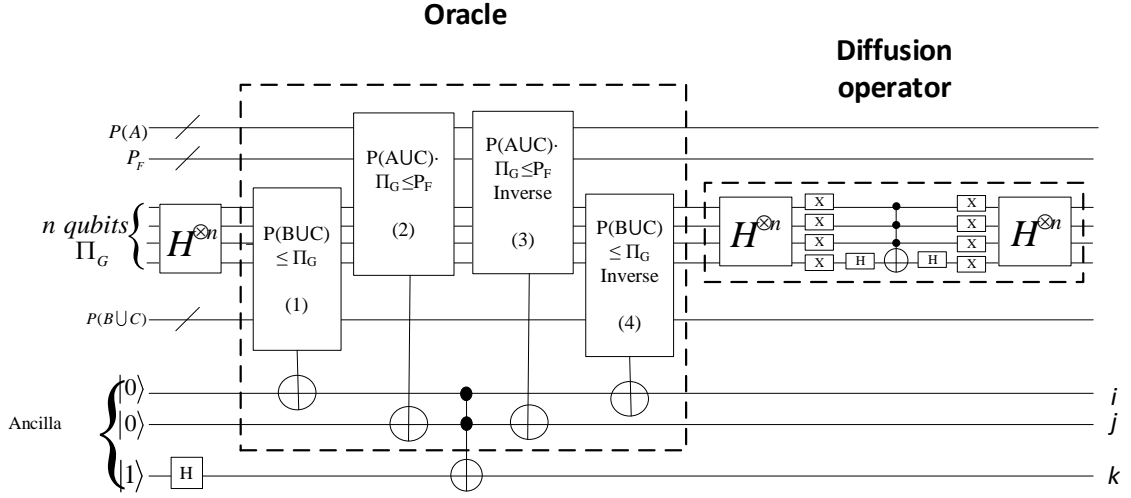


Fig. 9 Grover iteration G including diffusion operator.

Blocks (1) and (2) in Fig.9 that perform refinement test are both quantum circuits that realize function $R(P(A), P(B)) = \Pi(EQ(a_i, a_j) \rightarrow EQ(b_i, b_j))$. In contrast to CMOS circuits, quantum circuits are realized sequentially, one operation at a time. For instance, the function $EQ(a, b)$ is realized with bitwise XNOR operation followed by a logic AND on all the results of XNORs, shown in Fig.10. This bitwise XNOR can be realized with quantum gates by applying a CNOT gate followed by a NOT-gate to each corresponding bit one by one. Then a n -bit control Toffoli gate is used as n -input AND gate to produce the product of XNORs on an ancilla bit. An example of quantum realization of $EQ(a_i, a_j) \rightarrow EQ(b_i, b_j)$ is shown in Fig.10. After the outputs of $EQ(a_i, a_j)$ and $EQ(b_i, b_j)$ are computed and set on the ancillas bits, an imply gate is applied to two ancilla lines i and j , both initialized to 0. The imply gate is realized with a NOT gate followed by a Toffoli gate with a target line initialized to 1. After computing of the output, a mirror circuit is added to restore any modified input qubit and ancilla bit. Fig.11 shows a full quantum circuit in which $R(P(A), P(B))$ and input partitions are encoded by three projective

minterms, a_1, a_2, a_3 and b_1, b_2, b_3 , respectively. The EQ operations are applied to all three pairs of projective minterms, the product of all results of EQ operations is the final output of function R. The product of EQ operations is realized with m-controlled Toffoli gate, where m is the number of pairs of projective minterms.

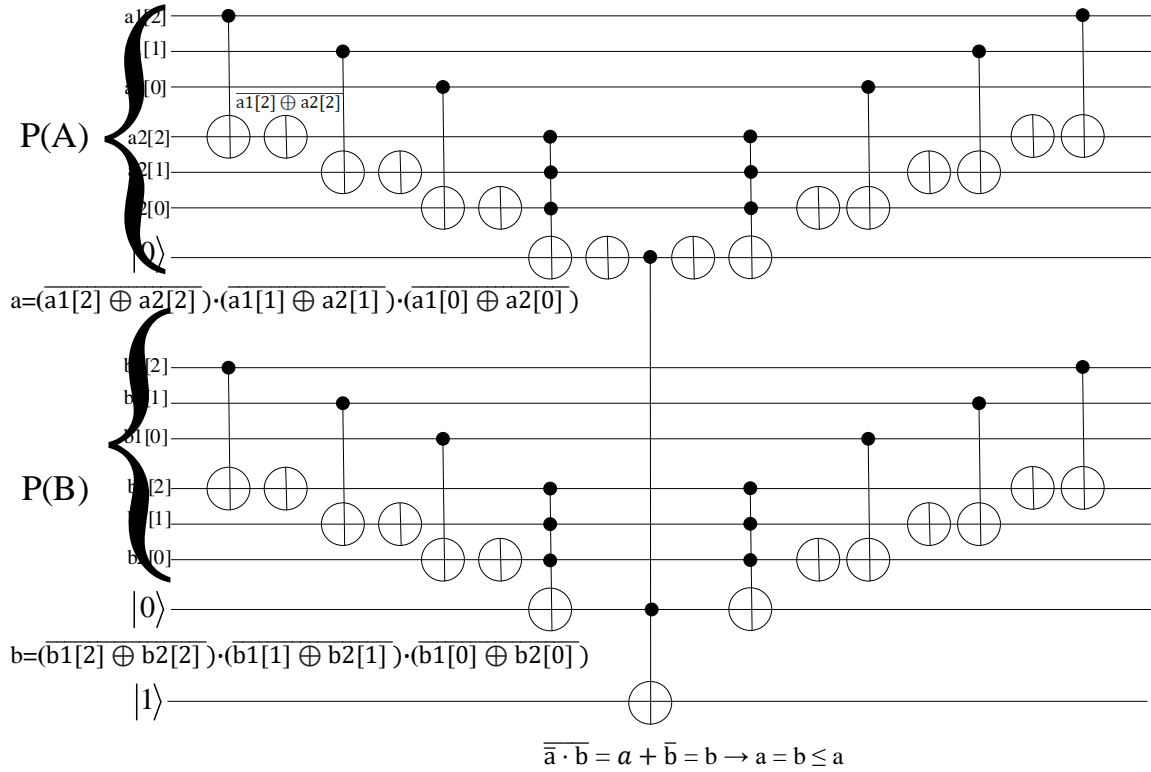


Fig. 10 Quantum circuit for EQ (a_1, a_2) \rightarrow EQ (b_1, b_2). Operations (1), (2) and (3) are drawn in parallel to save space. But, in fact, they operate in sequence not in parallel.

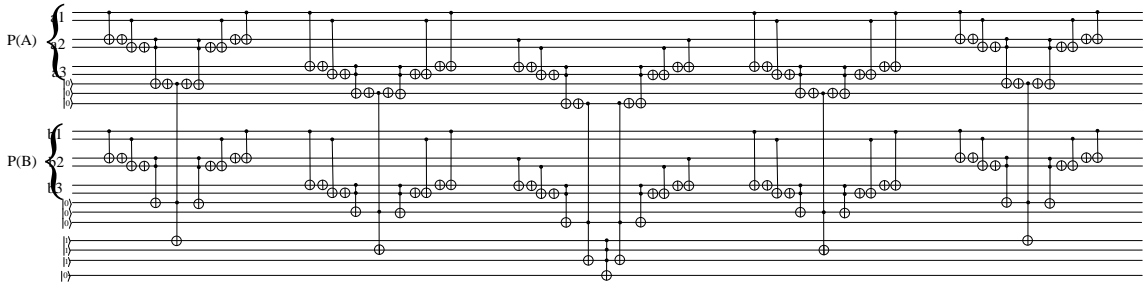


Fig. 11 Quantum refinement test for partition of projective minterms with 3 elements. $R = (EQ(a1, a2) \rightarrow EQ(b1, b2)) \cdot (EQ(a2, a3) \rightarrow EQ(b2, b3)) \cdot (EQ(a1, a3) \rightarrow EQ(b1, b3))$.

To realize function $R(P(A \cup C) \cdot \Pi G, PF)$ of block (2) in Fig.10, the partition multiplication is required. With the proposed encoding, a partition product $P(A) \cdot P(B)$ is realized using the concatenation of the qubit vectors representing $P(A)$ and $P(B)$. In the function of block (2), one of the inputs to function R is a partition product. The quantum circuit implementing R applies bitwise XNOR to both input qubits. A bitwise XNOR is applied to a product of partitions $P(A) \cdot P(B)$ as shown in Fig.12.

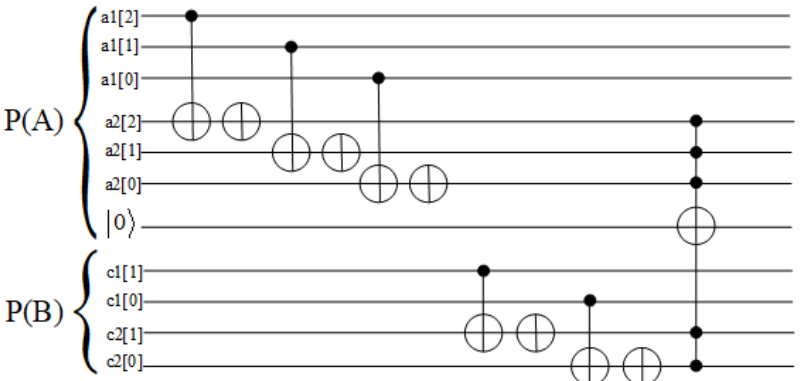


Fig. 12 bitwise XNOR on the concatenation of $P(A)$ and $P(C)$. The mirror circuit is not shown.

We envision the Grover’s algorithm for decomposition based on Oracle 1 operating on a hybrid computing system [32, 33], with the quantum computer as a coprocessor that is controlled by a classical computer. As shown in Fig.13, the quantum coprocessor performs only quantum computing, such as Grover’s algorithm. The classical computer

performs operations interacting with a quantum unit, such as generating input partitions A, B, and C for the quantum processor and verifying the result Π_G measured in the quantum processor. An exhaustive search algorithm for finding a valid Π_G can be implemented in a hybrid system. The classical computer generates an arbitrary free set and bound set for the input partitions. Then, the input partitions are set to the input qubits of the quantum processor. The quantum processor performs Grover's algorithm with Oracle 1. After a valid Π_G is found by Grover's algorithm, it is measured and converted back to classical bits. The classical computer then verifies the result Π_G with the software implementation of boolean oracle function $f(x)$. If the Π_G is valid, then it is saved as a solution. If the Π_G is invalid, then the same set of input partitions would be sent to the quantum processor again. In this variant, by iterating through all possible input partitions, a relatively good result can be found, but not necessarily the exact optimal solution. In the next section we will present another variant that finds the exact optimal solution.

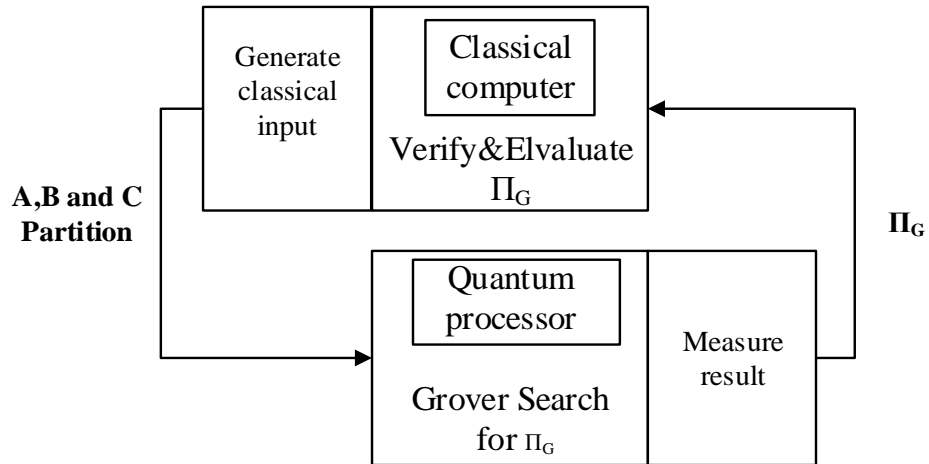


Fig. 13 Classical and quantum hybrid system for decomposition.

3.3.7 Oracle synthesis for the decomposition problem with minimum number of blocks

A minimum of number of blocks in Π_G is desired in both machine learning and digital circuit design applications. In order to find an exact optimal solution, which minimizes the number of blocks in Π_G , Oracle 1 is modified such that it not only validates Π_G , but also checks whether the number of blocks in the derived Π_G is smaller than a given threshold value. If a derived partition doesn't satisfy this condition, it would not be marked as a solution. A proposed Oracle 2 is shown in Fig.14. A Partition Counter and a Threshold Checker are added to Oracle 1,. The role of the counter is to count the number of distinct encodings in the derived partition Π_G . Then, a threshold checker compares the number of blocks from the counter against a threshold Max, which is given as an input to the oracle. Both outputs of the two refinement test blocks and the threshold checker must be 1 to set the output of the oracle to 1.

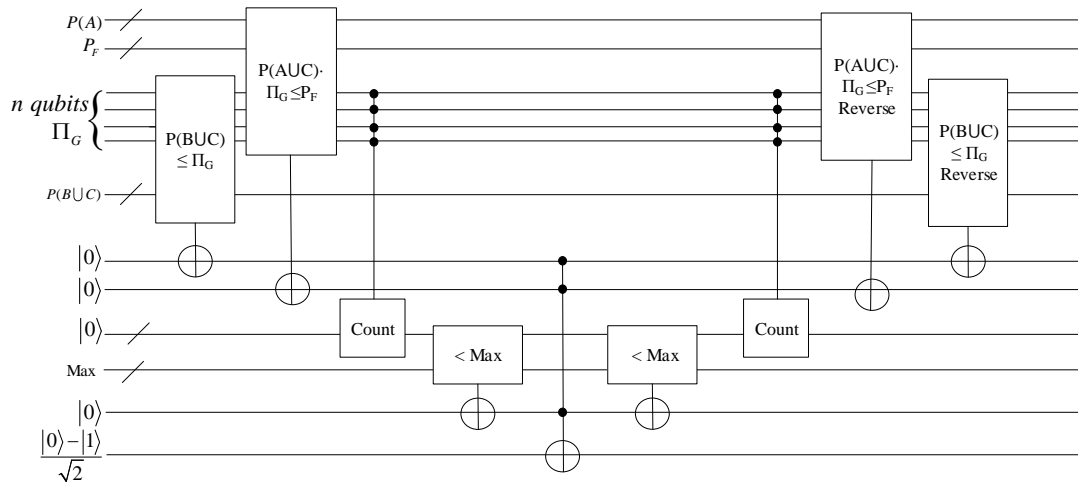


Fig. 14 Partition Counter and Threshold Checker are added, such that Oracle 2 tests both validities of the derived partition and the number of blocks.

A Partition Counter consists of a series of n -controlled Toffoli gates and control increment gates with n target bits and one control bit, where n is the size of the counter.

When the control increment gate is applied, the binary value represented in the target bits

is incremented by one if the control bit value is 1. Otherwise, the value stays unchanged. Fig.15 presents a Partition Counter and a Threshold Checker for a Π_G with three minterms. In a Partition Counter, two vectors of ancilla qubits x and y are needed. The size of vector x is the number of minterms in the input partition and size of vector y is the number of possible blocks. Each bit in ancilla vectors y corresponds to one partition encoding, if any partition coding appears in the input partition, the corresponding qubit in vector y is set to 1. By counting the number of 1's in vector y using control increment gates, the total number of blocks is counted. The Threshold Checker is just a quantum binary comparator comparing the counter value against the given threshold Max , the output bit is set to 1 when the counter value is smaller than Max , and otherwise, it is set to 0. The quantum control increment gate and the binary comparator are two commonly used quantum circuits proposed in various studies [25-27].

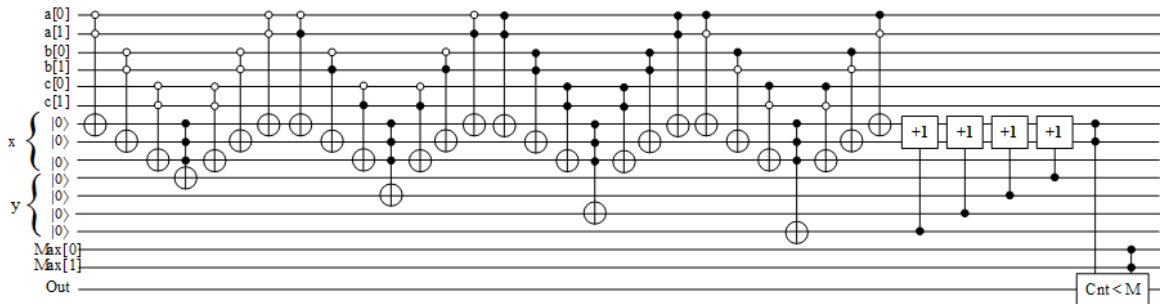


Fig. 15 Quantum Partition Counter and Threshold Checker.

The exhaustive search algorithm illustrated in Fig.15 can be implemented with Grover's algorithm using Oracle 2 running on a hybrid computing system. The classical computer generated an arbitrary bound set and a threshold Max as the inputs to the quantum processor running Grover's algorithm. Each Grover's algorithm run returns a valid Π_G with block number smaller than the threshold value. A program running on the

classical computer measures and verifies Π_G . If it is valid, then the threshold Max is reduced by 1. The Grover's algorithm is run with the updated Max value until the returned Π_G is no longer valid. Since the result measurement is probabilistic, even with very high probability to be correct, we apply the policy that if there are 10 invalid results consecutively, which means there are no longer valid Π_G meeting the threshold requirement, then the most recent Π_G is considered the optimal solution for the given bound set. The classical computer program enumerates to next bound set and resets the threshold Max value. In principle, by enumerating through all possible bound sets, the exact optimal solution is found. In the next section we will focus on Quipper which is a high-level quantum programming language that supports hybrid quantum and classical computing used by us to implement the above algorithms.

The original algorithm proposed by Grover is to assume a unique solution that satisfies the desired condition. Grover briefly considered the situation of multiple solutions and stated that it can be achieved by modifying the number of Grover iterations, but he did not provide any detailed implementation and analysis. Boyer et al. [43] expanded and generalized Grover's algorithm to k-solutions [43,44]. They provided a tight analysis of Grover's algorithm and proposed a new algorithm to find a solution in cases the number of solutions is more than one and is unknown ahead of time. Younes et al. [44] verified the algorithm and observed that the algorithm only works for $1 \leq k \leq 3N/4$ and when k is the number of solutions. Their algorithm is shown as follows:

1. Initialize $m = 1$ and set $\lambda = 6/5$. (λ is between 1 and 4/3)

2. Pick an integer $0 < j < m-1$ uniformly and randomly among the non-negative integers smaller than m .
3. Apply j times of Grover iterations G .
4. Measure the register: let i be the outcome.
5. If $f[i] = 1$, then the solution is found and exits.
6. Otherwise, set m to $\min(\lambda m, \sqrt{N})$ and go back to step 2.

The expected time complexity of this algorithm is $O(\sqrt{N/k})$ [43,44]. The searching algorithm I proposed for finding optimal Π_G for decomposition will have multiple solutions. I employed this technique to extend Grover's algorithm for multiple solutions which is implemented in a hybrid system composed of both standard and quantum computers.

3.4 Quipper and circuit modeling

Quipper is a functional programming language based on Haskell for quantum computing [28-30]. Quipper allows specifying quantum circuits by describing them in a simple programming style. It provides the capability to synthesize and simulate the circuits. Quipper follows Knill's QRAM model [35] for quantum computation, in which quantum computation is performed by a collaboration of classical and quantum processors. The classical processor performs classical computations such as control, result verification and loops, while the quantum processor is specialized in performing unitary operations and measurements. In Quipper, classical and quantum data can co-

exist. Classical wires and gates can be connected with pure quantum gates that provide data for input. A quantum bit can be turned into a classical bit with an operation called measurement.

Some quantum programming languages, such as QASM [34] operate by describing quantum algorithms with gate-by-gate instructions. However, we found that this approach is not very efficient when it comes to implementing larger-scale quantum algorithms such as Grover's algorithm, that requires a large amount of repetition of circuits. The quantum algorithms we introduced are described at a relatively high conceptual level. Many tasks in the algorithms are to perform manipulations at the level of entire sub-circuits, rather than individual gates. For example, the three circuits in Fig.10, that realize function EQ are identical. The only differences among them are the inputs. In Quipper, qubits are held in variables, and gates are applied one at a time [25]. Subroutines can be used to group gate-level operations. We find this property to be very useful. Quipper offers an abstraction that a quantum operation is a function. The inputs of the function are some quantum data. The function performs quantum operations on them and then outputs the changed quantum data. Block structure is another useful feature offered by Quipper. Functions that generate circuits can be reused as subroutines to generate larger circuits. A Quipper code implementing a single comparator of two pairs of minterms is shown in Fig.16.

```
EQ pair1 pair2 len1 len2 a b c d cnt result = do

    EQ_half pair1 pair2 len1 len2 a b c d

    qnot_at (result!!cnt) `controlled` (c, d)

    EQ_half_inv pair1 pair2 len1 len2 a b c d
```

Fig. 16 Quipper code for a block structure and an example shows calling the subroutine to construct a block of circuits.

The subroutine in Fig.16 is called *EQ*, the arguments to the subroutine are qubits and parameters. Argument *piG*, *abc*, *ancilla_piG*, *ancilla_abc* and *equal1* define the qubits for this operation. Arguments *pair1*, *pair2* define the labels of the minterms pairs in *piG* and *abc* with bit lengths of *len1* and *len2*, respectively. The circuit in Fig.17 is constructed by calling the subroutine *EQ*. In the oracles that we proposed, all the possible pairs of minterms are compared. By using circuits generating subroutine and auto-generate arguments, a large scale quantum circuit can be described with relatively small amount of code. Reusing block structures also reduces the possibility of human mistake.

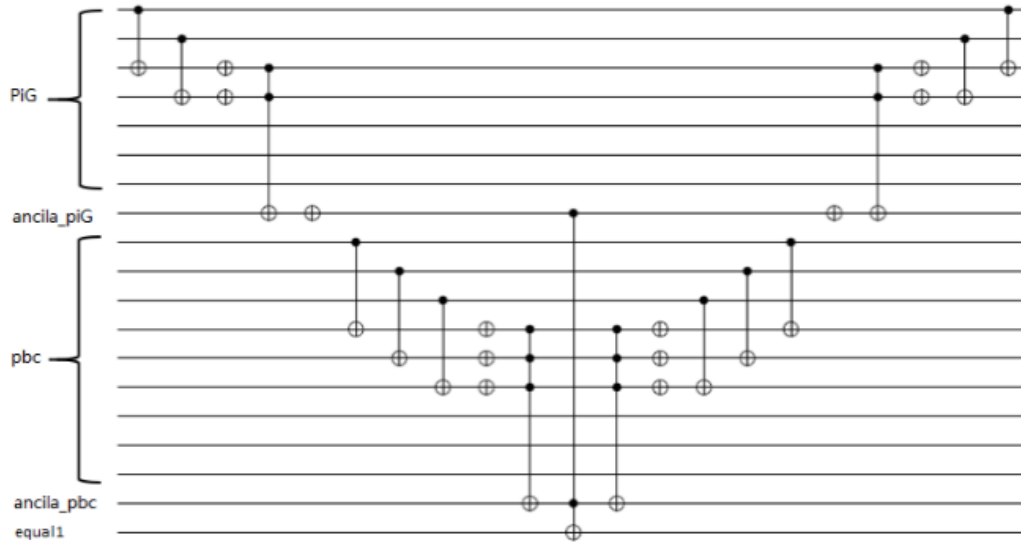


Fig. 17 Equivalence test circuit is constructed by calling the subroutine EQ.

Furthermore, Quipper provides a *for loop* and recursion to describe repetitive circuits. For loop can only be used to repeat the same operations multiple times without any modification to the operations in each iteration. An example of using for loop to repeat oracle and diffusion operation in Grover's algorithm is shown in Fig.18. The subroutine that constructs the oracle and diffusion operation repeats n times, where n is a parameter set by the user.

```

for 1 (n) 1 $ \i -> do

    oracle_3min piG controll_bit ancilla_piG

    pbc ancilla_pbc equal1 result1 pa

    ancilla_piGpa pf ancilla_pf equal2 result2

    Diffusion piG controll_bit

```

Fig. 18 Use for loop to describe Grover iterations.

Recursion is used to describe a repetitive circuit with modification in each iteration such as changing the control or target bits of operations. An example of using recursion in describing a parameterized control counter is shown in Fig.19. The control bits of each repetitive Toffoli gate and the variable *size* are decremented by 1 in each recursive call *C_counter* until *size* is less than 1. Since the number of times of recursion is given as an argument, the size of the counter can be customized by the user through argument *size*.

```

C_counter:: Int -> [Qubit] -> Qubit -> Circ Qubit

C_counter size cnt ctrl

    | size < 1 = return ctrl

    | otherwise = do

        qnot_at(cnt!!(size-1))`controlled`(ctrl,(take
        (size-1) $cnt))

        C_counter (size-1) cnt ctrl

C_counter 4 cnt control

```

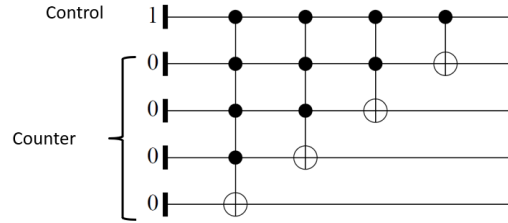



Fig. 19 An example of using recursion to describe a parameterized control counter.

The Quipper compiler synthesizes the circuit to either a schematic of the circuit or a file with the circuit described gate by gate. Also, the circuit can be simulated using a function call, *run_generic* in Quipper. The input of the *run_generic* is a function that constructs the whole circuit, and the outputs are the results of the measurement from the quantum circuit. This input/output interface in Quipper is the same as a real interaction between a classical computer and a quantum computer. Quipper is embedded in Haskell. Most of the functions in Haskell can be used in Quipper. So, all the computing tasks on the classical computer can be written using Haskell, such as: modification of the input to the quantum computer, control of the quantum computer, and verification of the results measured from a quantum computer. With this feature, the hybrid-system decomposition algorithm we proposed can be implemented in a unified language Quipper/Haskell and then can be simulated.

3.5. Resource analysis of oracle and algorithm simulation via Quipper

3.5.1 Resource analysis by Quipper

We implement the hybrid quantum-classical algorithm using Oracle 1 and Oracle 2 in Quipper, taking advantage of all the features provided by the language. By re-using circuit-constructing block we are able to describe the circuits for the algorithm

hierarchically. For example, a circuit-constructing block is used to describe subcircuits like the equivalence gate EQ, the single-bit binary comparator and the corresponding inverse circuits. The encoded free set and bound set data can be passed into the constructing block as multiple qubits data. The integer parameters are passed in as the index to select the particular minterm the quantum gates will be applied to. A series of quantum operations are applied to multiple qubits data instead of a single qubit. Other features like “for loop” are used to repeat the Grover loop, while recursion is used to describe adjustable circuits like the parameterized counter in Fig.19. We model the quantum circuit for the algorithms with both Oracle 1 and 2, taking input functions with 3, 4, 5, and 6 minterms. The process of customizing the Oracle for functions with a different number of minterms is not fully automated yet, but in the future, it is achievable by creating more parameter generation functions using Haskell. The circuit for the whole algorithm, including oracles and diffusion operation, was generated using the Quipper compiler. Quipper is equipped with a “print circuit” operation, which allows to generate the schematic of the circuit. Since the circuit with full iterations is large, in Fig.20 we only demonstrate one Grover’s iteration circuit printed by Quipper.

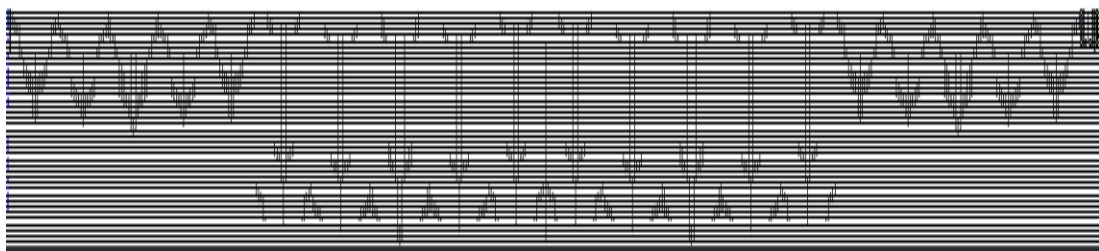


Fig. 20 A schematic of quantum circuit implementing one Grover’s iteration for decomposition of function with three minterms generated by Quipper.

A Quipper program is basically a description of the quantum circuit. Quipper provides the function for logical resource estimation, which is achieved by counting the number of gates in the described circuit. Tables 6 and 8 show the total numbers of qubits used in the algorithm with Oracles 1 and 2. Compared to Oracle 1, Oracle 2 has extra blocks for partition counting and threshold checking. Since we restore all the ancilla bits back to constants during each operation, ancilla bits can be re-used by the extra blocks in Oracle 2. Thus, Oracle 2 requires only a few more qubits for the threshold input. Tables 7 and 9 present the gate counts for circuits that use Oracle 1 and Oracle 2, respectively. Oracle 2 has an average of 12% more gates compared to Oracle 1 due to the extra counter and comparator blocks, since majority of the gates are parts of the refinement test block.

Number of minterm	Numbers of Data Qubits			Ancilia Qubits	Total Qubits
	Inputs	outputs	Π_G		
				3	15
4	20	8	8	17	53
5	20	10	10	22	62
6	24	12	12	31	79

Table 6 Numbers of qubits used to implement the Grover's algorithm using Oracle 1

Number of minterms	Quantum gate counts for one Grover's iteration				
	NOT gates	CNOT	Toffoli	Hadamard	Total gate

					counts
3	318	260	935	25	1513
4	484	396	1416	28	2296
5	856	760	2471	34	4087
6	1060	1160	3503	40	5723

Table 7 Numbers and types of quantum gates used to implement the Grover's algorithm using Oracle 1. We convert all the n-bits Toffoli gates to 3 by 3 CCNOT gate and 1 X gate with ancillas for gates count in the table

Number of minterms	Data Qubits				Ancilia Qubits	Total Qubits
	Inputs	outputs	Π_G	Max		
3	15	6	6	2	13	42
4	20	8	8	2	17	55
5	20	10	10	3	22	65
6	24	12	12	3	29	82

Table 8 Numbers of qubits used to implement the Grover's algorithm using Oracle 2

Number of minterms	Quantum gate counts for one Grover's iteration				
	NOT	CNOT	Toffoli	Hadamard	Total gate counts
3	318	272	1154	25	1769
4	484	412	1561	28	2485
5	866	868	2832	34	4600
6	1067	1260	4063	40	6430

Table 9 Number and types of quantum gates used to implement the Grover's algorithm using Oracle 2

3.5.2 Simulation with Quipper

We verified both quantum oracles by running a simulation on the oracle circuit with the input Π_G that enumerates all possible candidates. Then the measured results from the quantum circuit were compared with the results from a software implementation of the boolean oracle function $f(x)$, confirming that it is correctly implemented. We implemented two versions of the algorithm. The algorithm which finds a valid Π_G using Oracle 1 is called Algorithm 1. The algorithm that finds the valid Π_G with a minimum number of blocks using Oracle 2 is called Algorithm 2. To prove the correctness of the entire hybrid system algorithm, both algorithms are simulated with a set of test functions of 3, 4 and 5 minterms and the results were verified by the software implementation of the oracle function.

Please note that a function that is decomposed might have multiple solutions for a valid decomposition. With an unknown number of solutions, the number of iterations is unknown ahead. We implemented the algorithm proposed by Boyer et al [43] with a Perl script. The tasks of the script include: a) generating a random j , b) calculating m in the algorithm, c) modifying the number of iterations in the Quipper code, d) recompiling Quipper code, e) executing Grover algorithm using Quipper simulator. We ran the Algorithm 2 multiple times for the test functions with 3, 4, and 5 minterms, and we recorded the average numbers of iterations needed to find the correct solution. The number k of solutions for a valid decomposition is unknown for a given function, the strategy of Boyer et al's algorithm is to randomly attempt some numbers of iterations and narrow down the range of iterations. According to Boyer et al.[43], the expected time of their algorithm is $O(\sqrt{N/k})$. Since the number of solutions k is unknown for the test set

functions, the average number of iterations to achieve a correct solution shown in Table 10 is less than \sqrt{N} . According to Younes in [44], Boyer et al's approach to Grover's algorithm for an unknown solution was analyzed, and the algorithm works for $k < 3/4N$. In Algorithm 2, the threshold of the block count is part of the inputs, by setting the initial threshold close to the lower bound, the numbers of the solution are reduced, which guarantees satisfying the constraint of $k < 3/4N$.

Number of minterms	Average number of Iterations	Search space N	$\frac{\pi\sqrt{N}}{4}$	Number of runs	Average Simulation Time
3	5	2^6	6.2	50	84 s
4	11	2^8	12.5	50	33 min
5	20	2^{10}	25	10	27 hours

Table 10 Simulation results for Algorithm 2 with single threshold value

Because all the minterms of a function are encoded and stored in the Qubits, the algorithm requires many qubits for a function with many minterms and for larger functions it is beyond the limit that Quipper's simulator can simulate. For the functions beyond 5 minterms, we are able to construct the circuit but we are not able to simulate it. Quipper's simulator is not sufficient for simulating large quantum circuits. We can only simulate the Algorithm2 up to 5 minterms, which requires 65 qubits, on a desktop computer. The reason we can simulate 65 qubits is that there are only 22 qubits that are entangled and the remaining qubits are ancillas that depend deterministically on those 22 qubits. Then the simulator has only to keep track of 22 base vectors of the Hilbert space

instead of 265 base states. From this case, we also can learn that Quipper is more efficient in simulating circuits with relatively small numbers of entangled qubits but allows a large number of qubits that are only partially entangled. For the quantum oracle designed for six minterms function, there are 37 entangled qubits out of 82 qubits. We are not able to simulate this case on a desktop computer. Quipper simulator is not designed for simulating large quantum circuits efficiently. It does not support parallel computing or utilize GPU. It is hoped that in the future, larger circuits will be simulated on cluster computers with external simulators that could take the circuit description generated by Quipper or another quantum programming language.

Known exact decomposition algorithms are based on constructing trial partitions of the set of variables and then verifying the decomposability of the functions using the selecting set of inputs. To find the “exact” decomposition minimizing the overall cost of the realization of a given function appears to be very difficult to be solved exhaustively. Therefore, exact decomposition is usually refined to functions of a few variables. Most previous efforts to find exact decomposition restrict the decomposition to be a particular type, such as disjoint decomposition. For example, Bertacco and Damiani proposed an exact decomposition method that only works for a special type of decomposition. The decomposed sub-functions are disjoint, i.e., there is no shared input to the sub-functions. And the operations in the sub-function only can be one of the following operations: OR, NOR, XOR, XNOR [53]. The heuristic methods, such as the SAT-based approach [52] and BDD-based heuristic approach [57] are used for a more general form of decomposition, which does not guarantee a solution, moreover, an exact solution. With the problem formulation, each projective minterm and output are represented with a

partition. So the inputs of the decomposed function could be multi-valued. My exhaustive search method supports the general form of Ashenhurst-Curtis decomposition, which can have applications in both machine learning and logic synthesis. Grover's quantum algorithm provides a quadratic speedup for my search-based method. In addition, compared to an algorithm running on a classical computer which requires program memory, data memory, and cache for computing, my quantum algorithm requires only: (1) qubits to hold the information of the function, (2) qubits to represent the solution, and (3) ancilla qubits for intermediate values. We can approximate the size of the oracle for the larger size function since the oracle is constructed from multiple reversible circuit blocks, which are scalable. Table 11 demonstrates the estimated size of the oracle to decompose the benchmark functions. Monk1 is a machine learning benchmark. It is a well-known six-attribute classification problem with the 4-valued attributes. Functions Apex5, alu2, and the rest of the functions in Table 11 are from the logic synthesis benchmark IWLS93. A heuristic SAT-based decomposition algorithm uses 50Mb to 200Mb of memory [52] to decompose a function with 40 to 300 inputs, while my oracle only needs 26994 qubits for a function with 117 inputs. A Quantum computer will take much less memory (qubits). Recently, a broad wave of ambitious industry-led research in quantum computing is driven by D-Wave Systems and four of the tech giants Google, IBM, Microsoft, and Intel. These companies achieved relatively steady progress in the development and commercialization of the quantum computer. D-Wave quantum computers have gone from 28 qubits in 2007 to more than 2,000 in their latest 2000Q TM System machine [62]. We believe this speed of progress will allow 10,000 qubits

computer in a predictable future, which will make the algorithm implementation practical.

	Inputs	Outputs	Minterm	Qubits for oracle
Monk1	12	1	128	2048
Apex5	117	88	1227	26994
alu2	10	6	261	6003
f51m	8	8	76	1672
lal	21	16	65	1235
cmb	16	4	54	918

Table 11 Estimation of Qubits required to construct Oracle for benchmark functions

3.6 Conclusion

Functional decomposition is a hard problem that belongs to the class of NP-hard problems [16, 35]. We propose a quantum-based algorithm for a general form of decomposition, the generalized Ashenurst-Curtis decomposition. The algorithm takes advantage of the quadratic speedup achieved by Grover’s search algorithm. In contrast to all approaches to generalized Ashenurst-Curtis decomposition, this method guarantees an exact minimum solution, providing a sufficient number of qubits in the quantum computer. The problem is solved by implementing a quantum oracle, which is a non-trivial problem, not solved in classical logic designs and unique in the area of designing quantum oracles. The method to build this oracle differs from all those published oracles[41, 61]. We presented a systematic methodology for the design and construction of quantum oracles for finding the minimum-cost decomposition. My approach uses

partitions to represent the projective minterms and the output of the function. A Boolean logic implementation of partition operators was proposed — it is based on the encoding functions and data and then converting to quantum reversible circuits. Tests of the refinement relation were used to implement the Oracle. It is the first time that partition calculus is considered for constructing a quantum oracle for Grover's Algorithm. We implemented Grover's algorithm with two variants of oracles for decomposition using the quantum language Quipper. The decomposition oracles were synthesized and verified for various circuit instances and qubit sizes through simulation.

The algorithm given here is the first proposed method for functional decomposition that uses a quantum algorithm. Besides Ashenurst-Curtis' structure, a Boolean or multiple-valued function can be decomposed into other structures based on the relation between the decomposed sub-functions. My unified approach of constructing quantum oracles can be extended to several other decomposition problems with different structures, such as parallel decomposition [16] and generalized bi-decomposition [36], because all these decompositions can be formulated using only the two operations: the partition product and the refinement check.

Chapter 4 Quantum hybrid graph coloring algorithm for finding column multiplicity

We discover some disadvantages of using Grover's algorithm for some problems. Even with its quadratic speed up, the cases that the search space is very large are not realistic to be solved even with future quantum computers of many qubits. For example, in the direct implementation of graph coloring for Grover's algorithm, the search space size grows exponentially, so the standard graph coloring algorithm presented previously in section Chapter 3 is not feasible for a problem with a large number of nodes. The general idea is that instead of relying on one quantum search for large data, one should use a quantum computer as an accelerator to solve some partially smaller search problems. This is a tendency that one can observe in the newest published quantum algorithms. The future is in a hybrid “quantum accelerator” approach. Here in Chapter 4, I will propose a hybrid algorithm based on Grover's search for graph coloring, which is more effective and efficient when applied to those graphs that are typical for Ashenurst Curtis Decomposition. Instead of using Grover for the entire decomposition problem, a combined heuristic classical and quantum algorithm is used to decompose the global search into multiple local searches so that the number of qubits for each Grover sub-problem would be much smaller. This is a standard approach in recent research - to create hybrid rather than pure quantum algorithms. In addition, let us observe that most of the quantum algorithms developed based on Grover's algorithm currently don't have a systematic way to design an Oracle for a problem with a given size. Some approaches just proposed a matrix for the Oracle and then used decomposition algorithms to decompose it into a sequence of primitive quantum gates. In contrast, I created the oracle

using well-known quantum gates and functional blocks. To achieve the goal of scalability, one has to pay for the prize of extra ancilla bits.

Finding the Column Multiplicity for a chosen bound set is a crucial step for creating the Functional Decomposer program. In addition, a high percentage of the run time of a Functional Decomposer is spent on the Column Minimization part of Decomposition, so we choose Column Minimization as a candidate to be speeded up by the quantum algorithm. There are four methods to find Column Multiplicity in Functional Decomposition, Set Covering, Graph Coloring Clique Partitioning, and Clique Covering. In this dissertation, I will only focus on the graph coloring approach. The graph here is represented as an Incompatibility Graph. Therefore, the nodes that do not have a common edge can be colored with the same color. Graph Coloring for Ashenhurst-Curtis Decomposition was introduced by Muzio and Wesselkamper [92] and Perkowski [93]. Later, more approaches used the "Graph Coloring Approach" for more general decomposition problems such as information systems and machine learning [50].

4.1 Graph coloring and related work

The graph coloring problem is one of the most well-known combinatorial optimization problems. Graph colorings can be used to represent a mathematical model of various resource assignments, such as timetabling and scheduling, register allocation, and routing. We consider a non-oriented graph $G = (V, E)$, with a set V of n vertices and a set E of m edges. Given an integer k , a k -coloring c is a function that assigns to each vertex v of the graph an integer $c(v)$ chosen in set $\{1, 2, \dots, k\}$ (the set of colors), all vertices colored the same defining a "color class". A k -coloring c is a proper coloring if any two adjacent

vertices of G have assigned different colors. The chromatic number $c(G)$ of a given graph G is the smallest integer k for which G is k colorable. A proper k -coloring such that $k = c(G)$ is named an optimal coloring. The graph coloring problem is finding an optimal coloring of a given graph. Graph coloring problems can be defined more formally as follows:

Definition 4.1

$G = (V, E)$ be a graph, consisting of a set of n vertices V and E is a set of paired vertices.

Given such a graph, the graph coloring problem seeks to assign each vertex $v \in V$, an integer $c(v) \in \{1, 2, \dots, k\}$ such that:

- $E = \{\{v_i, v_j\} \mid v_i \neq v_j\}$
- $c(v) \neq c(u) \forall \{v, u\} \in E$; and
- k is minimal.

Classically, proper colorings can be generated using a greedy heuristic algorithm. This method involves assigning a color to each vertex in a predetermined order, either randomly or based on a specific criterion. At each step, the algorithm assigns the smallest possible color number to the current vertex that does not conflict with the colors already assigned to other vertices. More efficient greedy heuristics, such as DSATUR[87] and Recursive Largest First (RLF)[86], use more refined rules to determine the next vertex to color. While greedy algorithms are generally fast, they often require more colors than the chromatic number to color a graph. Better results can be achieved using more powerful

heuristics, such as local search and evolutionary algorithms. Local search approaches, such as simulated annealing and tabu search, have also been applied to the graph coloring problem. These methods involve gradually improving a candidate solution through local transformations. Population-based approaches, like memetic algorithms and quantum annealing, represent another family of heuristics for coloring. Finally, the latest approach, based on independent set extraction and progressive coloring, has proven to be effective for coloring very large graphs. It can be difficult to find a proper k -coloring for a large graph (e.g., with 1000 or more vertices) that is close to the chromatic number k . One approach to solving this problem is to use the "reduce and solve" principle, which involves a preprocessing phase followed by a coloring phase. During the preprocessing phase, independent sets are identified and removed from the original graph to create a reduced subgraph (called the "residual" graph). The coloring phase then determines the proper coloring for the residual graph. Because the residual graph is smaller in size, it is expected to be easier to color than the original graph. The extracted independent sets can then be treated as new color classes, with each set being assigned a new color. The coloring of the residual graph and the extracted independent sets together provide a proper coloring for the original graph.

I propose here a hybrid quantum algorithm for graph coloring that will reduce the graph in each step of the search and backtrack to achieve the final sub-optimal coloring. The distinct innovative advantage of my algorithm is that we can prove that it provides an exact solution for specific types of graphs. Moreover, it was found experimentally that

these graphs are more common in Ashenhurst and Curtis Decompositions than in randomly generated graphs.

4.2 Graph coloring based on domination covering

Definition 4.2 A node "A" in an incompatibility graph covers some other node "B" in the graph if all the following are satisfied:

- 1) Node "A" and node "B" have no common edge.
- 2) Node "A" has edges with all the nodes that node "B" has edges with.
- 3) Node "A" has at least one more edge than node "B".

We call this relation node A dominates node B. When two nodes satisfy the above relation, then both the nodes can be colored with the same color.

For example, in Fig. 21, node A dominates node E.

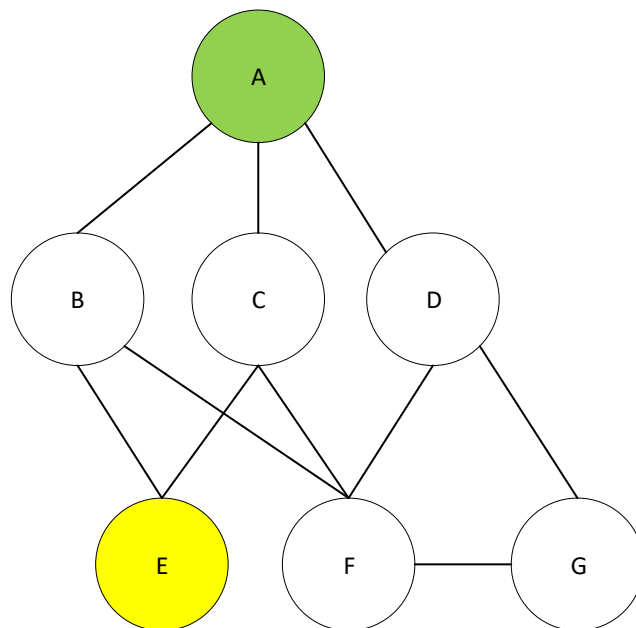


Fig. 21 Demonstration of node A cover node E.

Theorem 4.1 If any node "A" in a graph dominates any other node "B" in the graph, node "B" can be removed from the graph. In a domination relation, any one of the nodes "A" or "B" can be removed. In Fig.22, node E can be removed.

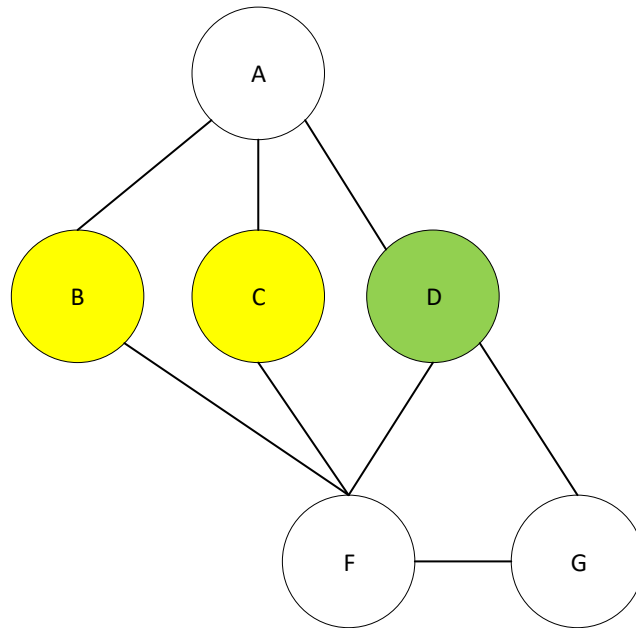


Fig. 22 Demonstration of E node E is removed from the graph of Figure 21, together with all adjacent edges.

Definition 4.3 If conditions 1) and 2) for coverings are satisfied and node "A" has the same number of edges as node "B", then it is called a pseudo-domination.

Definition 4.4 A Complete graph is one in which all pairs of vertices are connected.

Definition 4.5 A non-reducible graph is a graph that is not complete and has no domination or pseudo-dominated node(s).

Theorem 4.2 If a graph is reducible and can be reduced to a complete graph by step-by-step removing all its dominated and pseudo-dominated nodes, then the proposed algorithm can find the coloring with the minimum number of colors (the exact coloring).

Algorithm 4.1

Algorithm 1: Generate a coloring for a graph

```
input : Graph  $G$ 
output: Coloring  $C$  for the graph  $G$ , assigning a natural number to
        every node, using at most  $\chi(G) + E$  colors

 $R \leftarrow \{\}$ 
 $E \leftarrow 0$ 
while  $|G| > 0$  do
     $(d, a, b) \leftarrow \text{Find Domination}(G)$ 
    if  $d$  then
         $R \leftarrow \text{Append}(R, ('Domination', a, b, G))$ 
         $G \leftarrow \text{Remove Node}(G, b)$ 
    else
         $n \leftarrow \text{Random Node}(G)$ 
         $R \leftarrow \text{Append}(R, ('Random', \square, n, G))$ 
         $G \leftarrow \text{Remove Node}(G, n)$ 
         $E \leftarrow E + 1$ 
        Check clique( $G$ )
    end
end
 $C \leftarrow \{\}$ 
foreach  $(r, x, y, g) \in R$ , from the last removed node to the first do
    if  $r = 'Domination'$  then
         $C[y] \leftarrow C[x]$ 
    else if  $r = 'Random'$  then
         $N \leftarrow \{C[n] \mid n \in \text{Neighbors}(g, y)\}$ 
         $C[y] \leftarrow \min(\{c \notin N \mid c \in \mathbb{N}\})$ 
    end
end
```

An example is shown in Algorithm 1. Assume there is a subroutine Find_domination () that can return a random pair of nodes in which one dominates the other. The first step, Find_domination () is called on the full graph (clique). In this case, it is a non-reducible graph, so a random node 1 is removed from the graph and is labeled as "Random". After node 1 is removed, three nodes with a domination relation appear. Node 4 dominates node 2 and 6. Find_domination () will return any one of them. In this case, node 6 is return and get removed from the graph. And a pair of nodes (6, 4) is recorded and labeled as "Domination". The same steps are repeated until there is only a clique left, and record

each node of the clique as single node and labeled as "domination". Then all the domination pairs of nodes that are dominated by the same node are merged into the same group. Lastly, the nodes are merged and are removed randomly into the existing group by checking if any neighbor nodes are in each group, if not, then add the node into that group. For example, the randomly removed node 1 has neighbor nodes 2, 7, and 6, so it can only be added to group {3,5}. Lastly, each group of nodes is assigned a unique color. The domination Based Algorithm DOM finds the exact coloring with the minimum number of colors for a non-cyclic graph that can be reduced to a complete graph through the successive removal of all dominated and pseudo-dominated nodes. For cyclic graphs, while we do not have proof of optimality, Algorithm DOM still finds a good coloring if only a few cyclic graphs were consecutively created in the process. The number of colors that is different from the exact solution can be estimated by the number of nodes that were randomly removed.

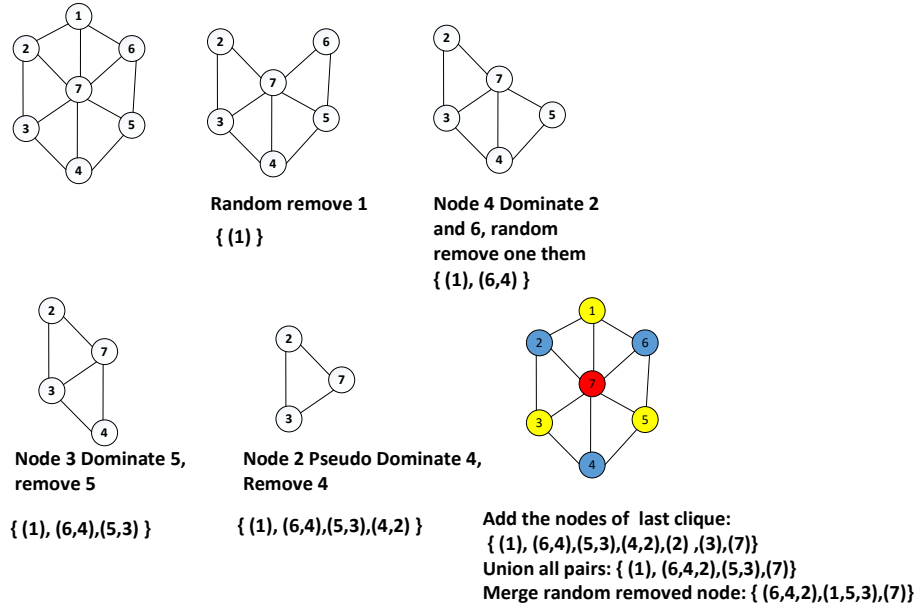


Fig. 23 Example demonstration of Graph coloring algorithm based on domination.

4.3 Quantum oracle for finding the domination pairs

The bottleneck of the above algorithm is the subroutine to find the node with a domination relation. We have designed an oracle for Grover's algorithm to locate the dominant node, which will be accelerated by Grover's algorithm.

First step to design the Oracle for the algorithm is to represent the input. For a graph with N nodes, each node is encoded with a N -bit binary string. Each bit represents the connectivity to other nodes. For example, in Fig.24, node A is encoded as $\langle 0111000 \rangle$, shown in Table 4.1. I select this representation because it is easier to verify/determine domination.

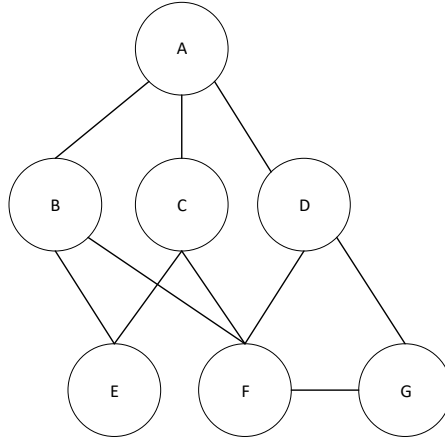


Fig. 24 Example graph.

	A	B	C	D	E	F	G
A	0	1	1	1	0	0	0
B	1	0	0	0	1	0	0
C	1	0	0	0	1	1	0
D	1	0	0	0	0	1	1
E	0	1	1	0	0	0	0
F	0	1	1	1	0	0	1
G	0	0	0	1	0	1	0

Table 12 Table. 4.1. Table for encoding of a graph in Fig. 24.

If node A pseudo-dominates node B, then node A connects to all the nodes that node B connects to. The operation to verify for the pseudo-domination can be achieved by bitwise imply. If $b_i \rightarrow a_i = a_i +$ for each bit, then node A pseudo dominates node B.

A block diagram for the oracle of finding pseudo dominates pair is shown in Fig.25. Because the encoding bit size N for each node is large (N is the number of nodes), instead of directly performing a search on the inputs, the Grover search is performed on an index

with the size of $\log_2(N)$. Therefore, a quantum multiplexer is needed to convert the index to the actual value of the inputs. After the domination check, a block of an inequality comparator is used to make sure that the selected two nodes are not identical. Since the number of nodes N might not be a power of 2, some padding values need to be added to the construction method for the multiplexer. The last two blocks are to invalidate the output if the padding is selected.

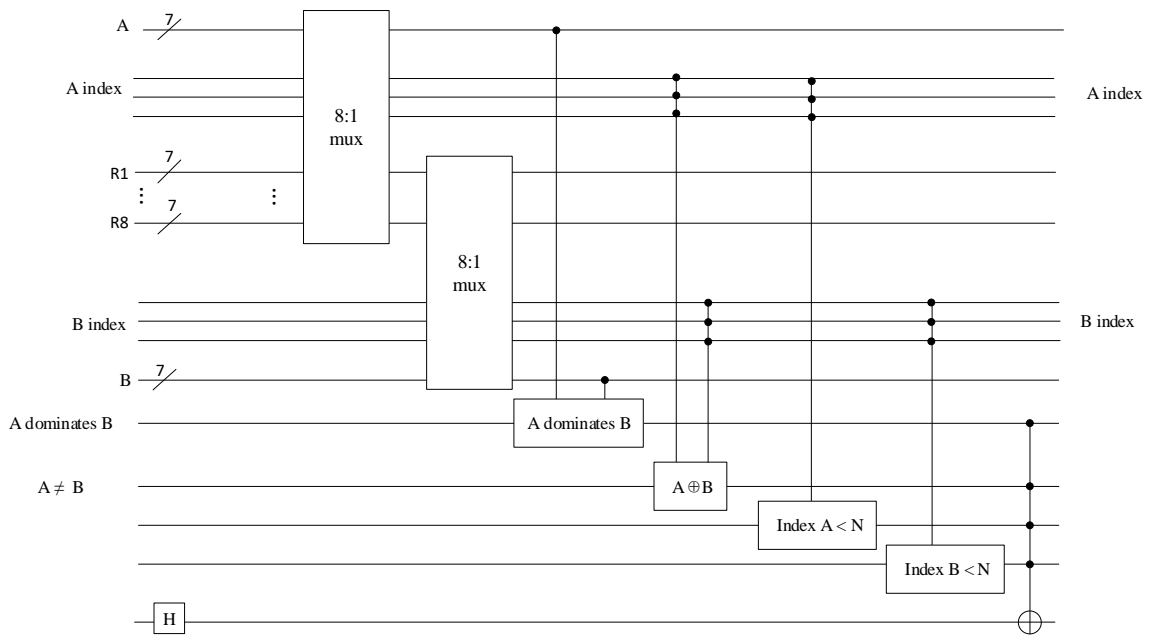


Fig. 25 Block diagram for the oracle.

4.4 Quantum counting

The oracle for finding domination has a very high chance of having multiple solutions. To estimate the correct number of iterations, quantum counting [22] is needed. The quantum counting algorithm calculates the number of solutions of a given Grover algorithm. This algorithm combines a quantum search and a quantum phase estimation. The central intuition applicable to quantum counting is to use the quantum phase

estimation algorithm to find an eigenvalue of Grover's search iteration. An iteration of Grover's algorithm rotates the state vector by θ in the $|\omega\rangle, |s'\rangle$ basis. The percentage number of solutions in the search space affects the difference between $|s\rangle$ and $|s'\rangle$. For example, if there are not many solutions, $|s\rangle$ will be very close to $|s'\rangle$ and θ will be very small. The eigenvalues of the Grover iterator can be extracted using quantum phase estimation to estimate the number of solutions. A basic procedure of quantum counting is demonstrated in Fig 4.6. The first step is to create a superposition to both registers (one for Grover's operator and one for the phase estimation) and then apply Grover's operator 2^n times. Lastly, estimating the value of θ . As a result, the output of the quantum phase estimation will be a superposition, and when it is measured, the output register provides the estimation of the solution count.

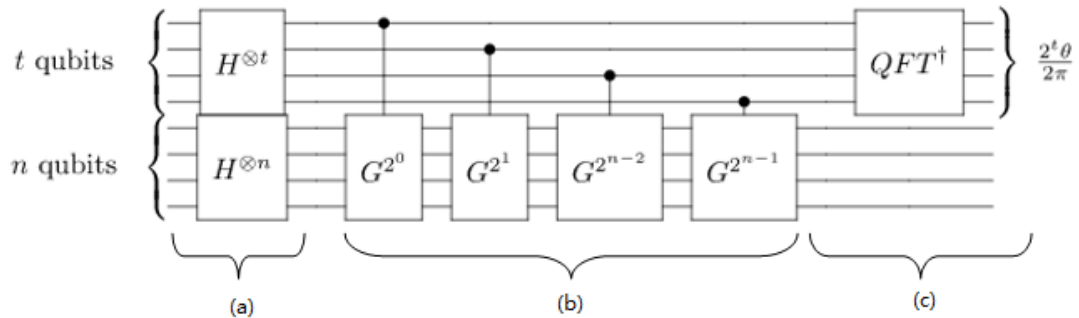


Fig. 26 block diagram for quantum counting. (a) is Hadamard, (b) is Grover iterator (c) quantum Fourier transform.

4.5 Quantum circuit block for building quantum oracle

Some of the quantum circuit blocks for oracle to find domination pairs can be reused from the design for Oracle of Decomposition from section 3.4. Only the new blocks will be introduced in this section.

Multiplexer 1: The first approach of the multiplexer is to build a 2^N Multiplexer with only 2 to 1 multiplexer, shown in Fig.26. The simplest 2 to 1 multiplexer we design is a five-gate design with CNOT, V, and V+.

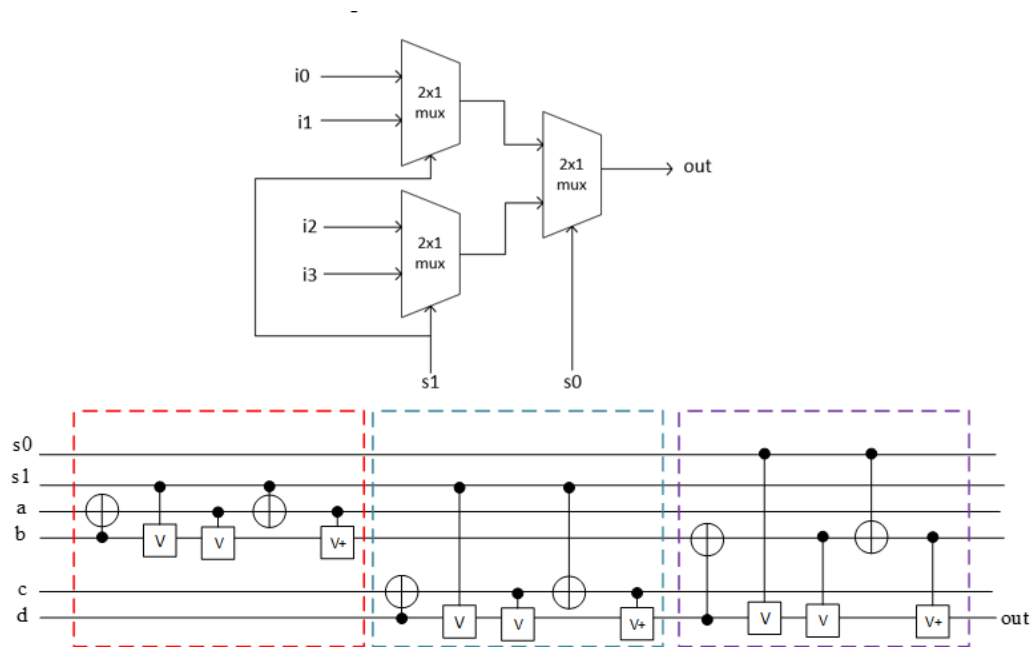


Fig. 27 quantum multiplexer 1.

Multiplexer 2: Fig.28 demonstrates a Toffoli base Multiplexer; the gates in the dash line can be replaced with a CNOT gate due to the input 0.

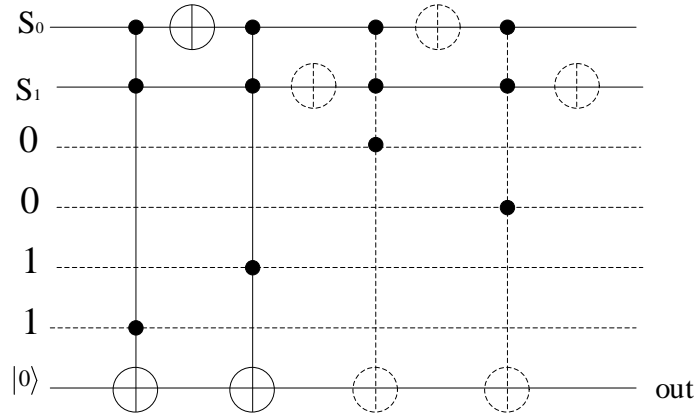


Fig. 28 4:1 Mux with half number of 1 and 0 as input.

Multiplexer 3: Instead of following Shanon's expansion to design the multiplexer, I use Davio decomposition. Because xor is the primary operator of Davio expansion, it can be easily transformed into quantum gates and reduce overall gate count compared to quantum multiplexer following Shanon expansion.

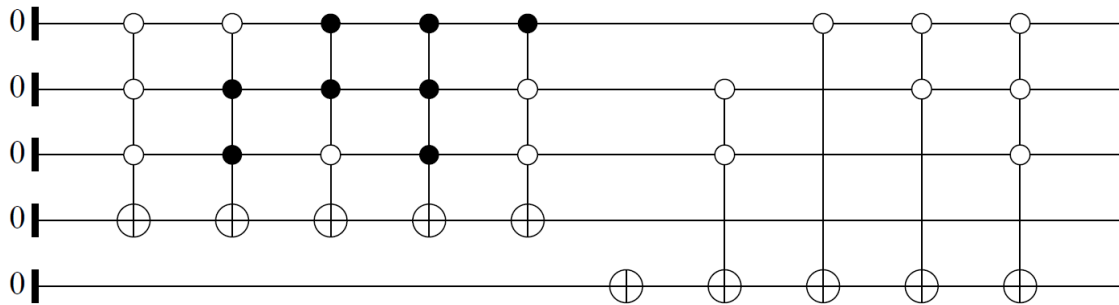


Fig. 29 The left Mux is based on Shannon, and the right is based on Davio expansion.

Accumulative Comparator: A swap gate-based accumulative comparator is proposed in my design shown in Fig.30. A Comparator is designed to compare two n-qubits inputs and output if they are equal, larger, or small. The result is encoded in two qubits. The encoding is shown in Fig.30. This comparator starts from the most significant bit (MSB). If the MSBs of the two numbers are not equal, the result of the comparator has been

determined. Then the swap gate will be active and transfer the comparison result through the rest of the comparator to the final output qubit. If the MSBs are equal, then move on to compare the following two bits until it counters a condition of not equal.

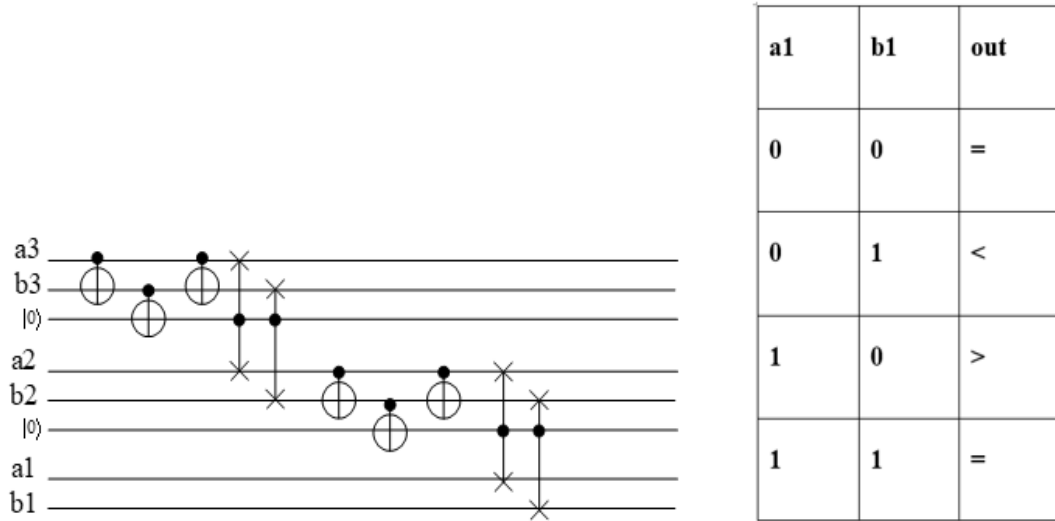


Fig. 30 quantum 2bit comparator and the encoding of output.

4.6 Hybrid quantum algorithm for graph coloring

I propose a hybrid algorithm on a hybrid system introduced in section 3.4. The classical computer performs operations interacting with the quantum unit, such as modifying the input graph for the quantum processor in each iteration. The quantum processor performs Grover's algorithm to find the domination pairs. Then classical computer measures and records the domination pairs from the quantum processor. When only one node or clique is left, the classical computer stops generating new input data for the quantum processor. The classical computer runs Algorithms 3.2 below to reconstruct the coloring from the list of saved domination pairs and random nodes.

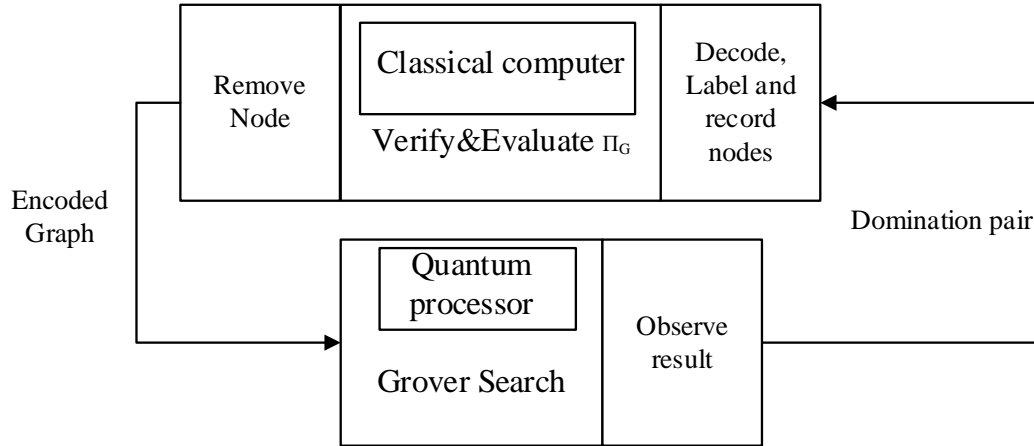


Fig. 31 Hybrid algorithm for graph coloring.

Algorithm 3.2

Algorithm : Reconstruct coloring from removals

input : Ordered list of node removal actions R for graph G , in reverse order

output: Coloring C for the graph G , assigning a natural number to every node

$C \leftarrow \{\}$

for $r \in R$ **do**

if r was removed at random **then**

$N \leftarrow$ all colors already used by neighbors of r in C

$c \leftarrow \min(N \setminus N)$

end

else if r was removed because of domination by d **then**

$c \leftarrow$ color of d in C

end

 color r as c in C

end

4.7 Quipper modeling and simulation

To simulate my graph coloring algorithm, I used the Quipper language, which is a domain-specific language for describing quantum circuits within Haskell. One of the

main benefits of Quipper compared to other quantum programming languages is its ability to abstract and reuse smaller components of an algorithm easily. Quantum algorithms are often more naturally described by composing smaller blocks than by specifying gates directly. Quipper is built around the Circ monad, which encapsulates a quantum circuit. For example, the function "Hadamard:: Qubit -> Circ Qubit" represents a circuit with a single Hadamard gate applied to a qubit. The Circ monad allows using standard Haskell tools for working with monads. For example, a circuit that applies Hadamard gates to a list of qubits can be written as "mapM Hadamard" with the type "[Qubit] -> Circ [Qubit]". This system makes it straightforward to abstract the oracle from the implementation details of Grover's algorithm, allowing the algorithm to be easily reused without worrying about its internal workings. At the most abstract level, Grover's algorithm is a function with the type "[Qubit] -> Oracle -> Circ ()," which can be used as follows:

```

type Oracle = Qubit -> Circ ()

grover :: [Qubit] -> Oracle -> Circ ()

oracle :: Graph -> Qubit -> Circ ()

initSearchSpace :: Graph -> [Qubit]

decodeSearchSpace :: Graph -> [Qubit] -> (Int, Int)

findDomination :: Graph -> Circ (Int, Int)
findDomination graph = do
  searchSpace <- initSearchSpace graph
  grover searchSpace $ oracle graph
  measure searchSpace >>= decodeSearchSpace graph

```

Fig. 32 Quipper code example demonstrates abstraction.

Another useful tool is the ability to generate mirrors for a given circuit automatically. In Quipper, the main way to compute is with a function with the simplified type `Circ [Qubit] -> ([Qubit] -> Circ b) -> Circ b`. The actual type is slightly more complicated, but not in a way that is relevant here. The first argument is a reversible circuit, while the second argument is a circuit operating on the results of the first circuit. The circuit takes the first circuit, feeds its output into the second circuit, and then runs the mirror of the first circuit, reversing all of the first circuit's qubits back in their original state. In order for this to behave correctly, the second circuit must also mirror any operation on the first circuit's qubits. Using this abstraction, it is relatively easy to compose large chains of mirrored circuits without worrying about how the mirrors are implemented. In structuring the implementation of the graph coloring algorithm with Quipper, I broke the oracle into many smaller blocks and then implemented each block more generally. This way, the program is more structured and maintainable, and these components can be easily reused for implementing other algorithms. An example of a few elementary pieces is shown below:

```

-- increments a binary counter
counterInc :: [Qubit] -> Circ ()
counterInc [] = return ()
counterInc (b:bits) = qnot b `controlled` bits >> counterInc bits

-- returns bits corresponding to the n-bit representation of x
toBits :: Int -> Int -> [Bool]
toBits x n = reverse $ take n $ map (testBit x) [0..]

-- generates a set of controls for comparing binary qubits with a constant
compareEq :: Int -> [Qubit] -> ControlList
compareEq n x = to_control $ map (uncurry Signed) $ zip x $ toBits n $ length x

-- initializes n qubits to the binary representation of x
initBinary :: Int -> Int -> Circ [Qubit]
initBinary n x = qinit_list $ toBits x n

```

Fig. 33 Quipper code example demonstrates map function.

These examples also demonstrate another important part of implementing quantum algorithms in a real programming environment, which is that there is a lot of interchange between the quantum and classical parts of the system. Reconfiguring the quantum circuit on the fly for every run is simple, and so it makes sense to generate it in a way that is specialized for the particular input data for that run, as opposed to writing the circuit once and then using some of the qubits to specify the input. The fact that Quipper is implemented as a library inside a normal programming language makes this very straightforward.

Using Quipper and the block base approach makes my program scalable. In addition, the circuit generation can be automated. Fig.34 shows the quantum gate count versus the number of nodes of the input graph, the y-axis is the gate count, and x is the number of vertices of the input graph.

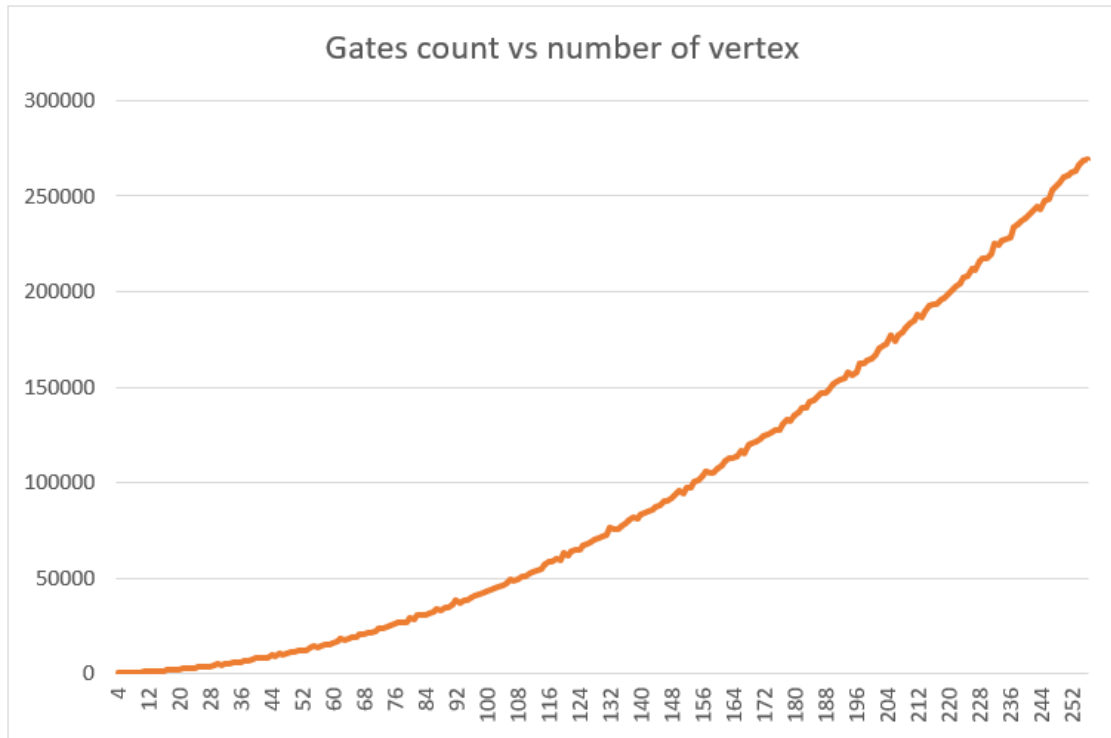


Fig. 34 gate count vs node count of a random generated graph.

I also constructed quantum circuits for some benchmark CP2002 and simulated them, result is shown in Table 13.

Benchmark	Qubit	Gate count	Edge	Node
1-FullIns_3.col	35	12632	100	30
1-FullIns_4.col	31	8206	1227	202
1-Insertions_5.col	39	97278	593	93
DSJC125.1.col	35	66788	736	125
DSJC500.9.col	47	899710	22487	500

Table 13 gate count for some case in graph coloring benchmark

Interesting topic for future research is to check how exact optimal coloring affects the quality of encoding.

4.8 Experiment Analysis

The conventional graph coloring solution is a greedy algorithm with a complexity of $O(N^2)$, the hybrid algorithm I proposed, without considering any delay between data transmission between quantum and classical computer, the complexity is $O(N*\sqrt{\log(N)/k})$, where k is the average number of solutions amount to all iterations. A comparison of quantum domination-based coloring (QDOM) and *Exact Graph Coloring* was made to show the quality of the result of my heuristic algorithm. Fig 4.2 shows colors generated by quantum domination-based coloring (QDOM) were one color away from the total numbers of colors generated by *Exact Graph Coloring* and so on till 5 Errors. From Fig 4.2, I can observe 90% of the runs can produce an exact solution. In this case, the method employed for exact Graph Coloring combines the greedy algorithm, backtracking, and cut-off. Perkowski [9] proved that an Exact Graph Coloring is not required to find the Column Multiplicity where Ashenhurst and Curtis Decompositions are considered. Exact Graph Colorings only take up more time and fail to produce significant changes in the results. So, this quantum coloring heuristic is a good candidate for quantum acceleration.

	1-FullIns_3.col	1Insertions_5.col	DSJC125.1.col	DSJC500.9.col
<i>exact</i>	90%	92%	89%	90%

<i>Error1</i>	5%	1%	5%	6%
<i>Error2</i>	3%	1%	2%	2%
<i>Error3</i>	1%	3%	2%	-
<i>Error4</i>	1%	3%	2%	2%
<i>Error5</i>	-	-	-	-

Table. 4.2 A Comparison of Total Colors generated by quantum domination-based coloring (QDOM) and with total colors generated by Exact Graph Coloring

Chapter 5 Design and optimization of memristive FSM

5.1 Memristor and related work

5.1.1 History of memristor

Memristors are nanoscale devices that have gained attention in the circuit design community in recent years. In 1971, Leon Chua mathematically predicted the existence of a fourth fundamental circuit element, which he called a memristor, that relates flux to charge (as shown in Fig.35 [76]). This idea remained theoretical until 2008 when HP Labs announced that they had found the missing practical realization of the element predicted by Chua [76] [77]. The HP team provided experimental evidence for the memristor, governed by the mathematical formulation of Chua's memristive hypothesis. They proposed a two-terminal device in which the applied voltage across the device changes the resistance of the device, and the resistance is retained when the voltage is absent. Memristors have the advantage of being able to store values without requiring electrical charge and can be manufactured with a small area and high density compared to CMOS circuits. The resistance of a memristor depends on the previous current that flowed through it, making it non-volatile as it retains the latest resistance when the power is turned off. When the power is turned on again, the retained resistance is used to restore the state of the memristor. This property makes memristors unique and expands their potential use in fields such as programmable logic, neural networks, large programmable state machines, and control systems.

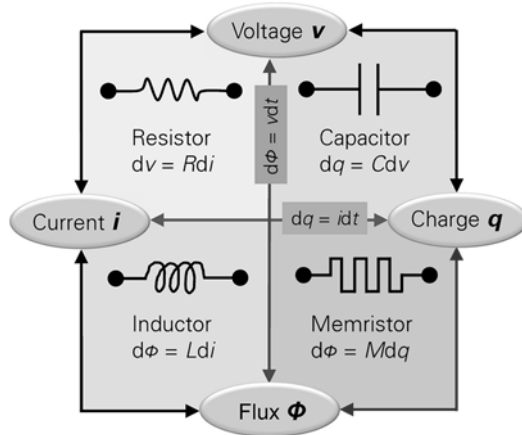


Fig. 35 Memristor and its relation between the magnetic flux and the electric charge.

5.1.2 Memristor circuit and related work

Initially, research on memristors focused on their application in memory [59]-[62]. However, more recent research has also explored the use of memristors in logic circuit design. One notable application is the realization of arbitrary Boolean functions using IMPLY operations in a crossbar architecture [63]-[66]. This research shows that memristors are capable of holding values and executing imply-logic operations, providing a new approach to addressing the "von Neumann bottleneck" [66]. There have been many efforts to implement small logic circuits such as adders, counters, and LFSRs using memristors, with some works using material imply gates built with memristors as the basic circuit elements [67]-[71] and others exploring alternative structures that utilize other Boolean operations [72]-[74]. However, most of these works have focused on individual logic gates and small circuits. Only a few papers propose frameworks or methodologies for system-level design with memristors, such as those by Rahman [74], Tissari [75], and Xie [84]. Rahman and Tissari propose configurable logic architectures similar to standard FPGAs, while the logic unit in Xie's architecture is fixed like an ASIC.

These architectures are based on the use of stateful imply logic gates in crossbars with memristors. Memristive state machines are a particularly exciting application of these programmable memristive architectures.

5.1.3 Memristor characteristic and memristor circuit

The I–V curves of memristors form pinched hysteresis loops shown in Fig.36, and the forms of these loops depend on the amplitudes and frequencies of the input voltage signals. This phenomenon defines a state variable, which determines the memristor’s instantaneous resistance, also known as the memristance.

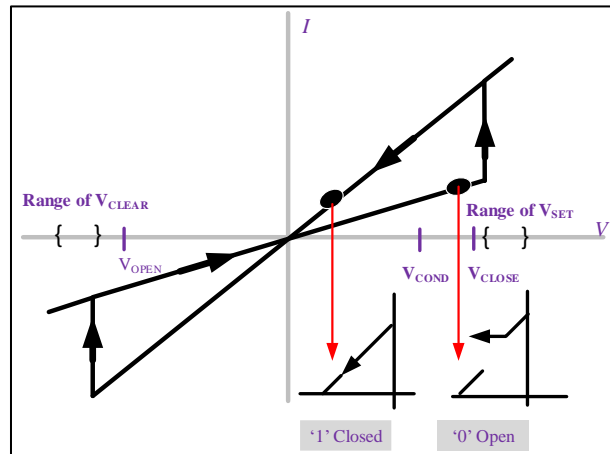


Fig. 36 I-V Characteristic of Memristor.

Memristor can be used to implement the IMPLY logic, as shown in Fig.37. The logic state is the memristor's resistance in terms of the control logic based on the IMPLY gates. The HRS (High Resistance State) or R_{off} is the logic 0, and LRS (Low Resistance State) or R_{on} is the logic 1. The initial states of the memristors are the inputs, i.e., if memristor a is in HRS then the initial value is 0. Both the memristors are further connected to resistor R_G i.e., The value stored in memristors turn into output value after the V_{SET} and

V_{COND} are applied. In IMPLY logic gates, always $R_{ON} \ll R_G \ll R_{OFF}$, $V_{COND} < V_C < V_{SET}$, and $V_{SET} - V_{COND} < V_C$. Here while performing $a \rightarrow b$ source memristor is ‘a’, and target memristor is ‘b’.

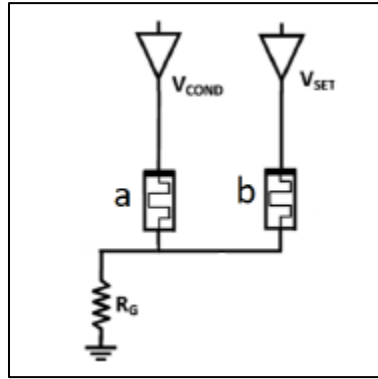


Fig. 37 Circuit implementation of IMPLY logic gate.

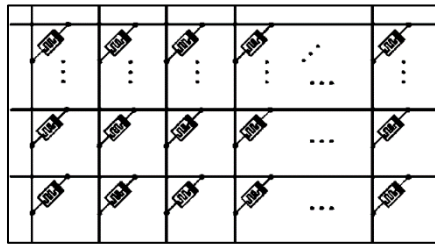


Fig. 38 A crossbar with memristor.

In memristor computing, the basic architecture is formed with the crossbar of wires with memristors at the crossing nodes. Fig.38 shows a crossbar of memristors. In this crossbar, the source and target memristor can be chosen by applying the respective voltages V_{SET} and V_{COND} at respective ends in the crossbar. Each gate corresponds to one set of V_{SET} , V_C , and ground. Using the serial control signal generated by a pulse generator, multiple logical operations can be performed in any row of memristors in the crossbar. Our team with Kamela Rahman [74] proposed a memristive stateful IMPLY-Logic based reconfigurable architecture called MsFPGA. This architecture comprises three major

components: a memristive RAM, a CMOS controller, and a parallel SIMD datapath realized with a nanowire crossbar array. The architecture's basic structure is shown in Fig.39. The configurations of each IMPLY gate are stored in a memristive RAM in the form of binary code similar to computer processor instructions [74]. The CMOS controller read the configuration from the memristive memory and decoded the binary code to create a set of control signals. Then those signals are supplied to the crossbar to realize configured logic with a series of IMPLY gates. In this dissertation, I will refer to the saved binary code representing the crossbar configuration as the memristor crossbar's instruction.

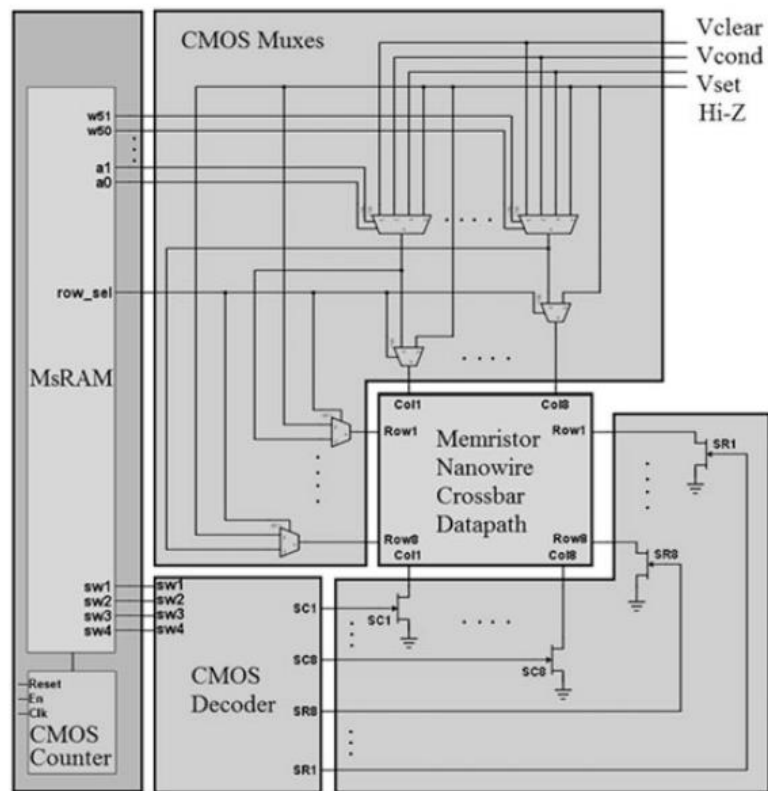



Fig. 39 MsFPGA architecture.

I will use the “Imply Sequence Diagram” (ISD) notation to analyze and synthesize the memristor circuits. In this notation, horizontal lines represent physical memristors, while this symbol  represents a pulse applied to it. The top side of this symbol is the negated input. The left side is the non-negated input and the value of the memristor before the pulse. The right side is the value of the memristor after the pulse. A 0 in the square indicates an additional pulse required to reset the input state of the memristor to 0. Horizontal lines correspond to memristors, and vertical dash lines correspond to moments of time. I define a memristive cascade as a sequence of stateful IMPLY gates presented with ISD notation in which rows correspond to memristors and columns correspond to moments of time (pulses). An example of a memristive cascade to realize $a+b'$ is shown in Fig.40.

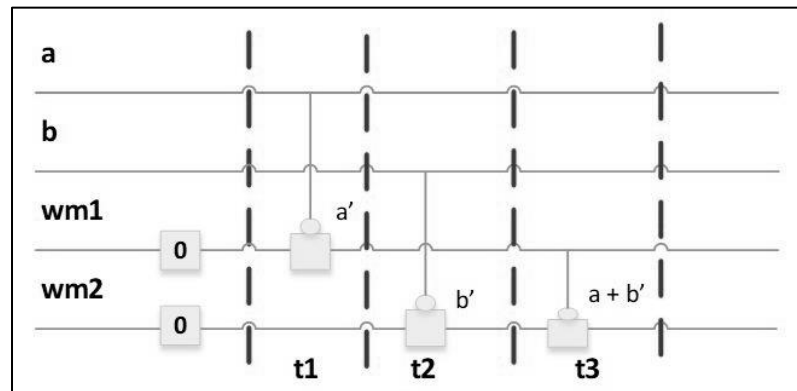


Fig. 40 one pulse corresponds to one imply gate in that given pulse period. Before t_1 , wm_1 and wm_2 are reset to 0. At t_1 , the wm_1 becomes a' , because $wm_1 = a$, $wm_1 = 0+a'$.

5.2 Low-power memristive state machine

A memristor can hold a state and perform a logic operation, making it suitable for use as

both memory and combinatorial logic. A classical state machine consists of both combinatorial logic and memory and can be realized using memristor technology. The memory-like property of stateful memristors allows for the elimination of explicit registers, leading to lower area and power consumption compared to CMOS. Additionally, because memristors are non-volatile, the state of the machine will not be lost due to a power outage. If the power supply is cut off, the state of the state machine will be preserved, and the machine will continue to operate at the last state before the power outage when the power is restored.

This chapter discusses memristor-based finite state machines (FSMs) at the circuit level. It focuses on how to design a machine that consumes as little power as possible and has as little delay as possible for a given machine specification. The goal is to provide general and practical methods and models that can serve as the foundation for designing memristive state machines, similar to classical logic design.

A state machine is defined as a quintuple $M = (S, I, O, \delta, \lambda)$, where S is a finite set of states, I is a set of inputs, O is a set of outputs, δ is the transition function, and λ is the output function. The state machine operates in two basic steps: computation of the transition and state update. A traditional synchronous state machine is implemented using a sequential CMOS circuit, with a combination logic block used for computing the transition and output function and a memory block with a synchronized clock used to hold the next state. In a memristor-based implementation of an FSM, a memristor crossbar is used for the datapath. Each memristor cell holds the value of a binary state. A pulse generator generates the control signal that drives the transition of the state of each memristor, which represents the state of the FSM (the state is a vector of memristors for

multiple-bit state). A memristor cell can be viewed as a memory with computation capabilities. For example, in Fig. 5.7, each rectangle represents a memristor cell. The input for the function 'ab' is the value stored in the first two memristor cells. A series of Imply operations are performed on inputs a and b, and the result, ab, is stored in the third cell by applying a sequence of control signals to the memristor crossbar.

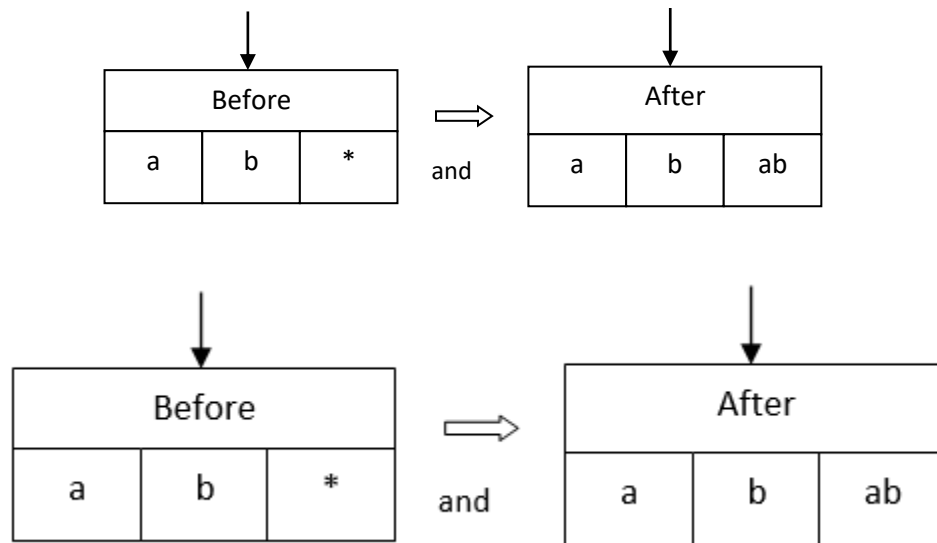


Fig. 41 rectangles represents value transition of a memristor cell.

A pulse generator generates control pulses that are pre-programmed by the user. The control sequence determines which cells are the operands, which cell is the result, and what boolean operation to perform. The boolean logic is executed in the form of an imply-cascade, such as $(0 \rightarrow ((0 \rightarrow a) \rightarrow b))$. Any Boolean function can be realized with a cascade of imply gates. For example, $a + b = ((a \rightarrow 0) \rightarrow b)$.

Once the basic computing model for the memristor has been established, I use it to demonstrate my state machine model. I label four cells in the memristor logic unit: I for

input, S for the current state, S' for the next state, and O for output as shown in *Fig.41*. Each cell does not necessarily have to be one bit. it can be multiple bits.

5.2.1 Execution sequence of the memristive state machine

Step 1: Read from the input cell.

Input changes are the driving force behind the state machine and are the interface to the external world. Most digital circuits still use voltage signals to communicate, so I have chosen voltage input for the state machine. The volistor logic gate converts the input into resistance stored in the memristor cell. The volistor logic gate is a memristor gate design by Muayad [89] that takes voltage as input and converts it to resistance stored in the input memristor cell. It can be implemented in the same crossbar architecture as stateful imply-logic gates.

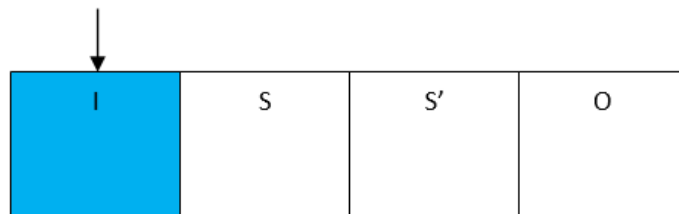


Fig. 42 I is the memristor cell I store input, and cell S store current state, cell O store the output.

Step 2: Execute the output function λ . A sequence of control signals corresponding to the output function λ is supplied to the memristor cells. This control sequence takes cells I and S as input to the function λ , and the result is stored in cell O. For example, if the output function is $O = I + S$, then the control sequence represented in an Imply cascade would be $O = ((0 \rightarrow I) \rightarrow S)$.



Fig. 43 Cell in green demonstrates the input cells of the output function λ , and the blue cell shows the output cell.

Step 3: Read the output. A memristor reading circuit [89] reads the resistance stored in cell O.

Step 4: Execute the transition function. A control sequence δ is applied, and the next state is computed and stored in cell S'.

Step 5: Update the input cells with the subsequent inputs.

Step 6: Execute the output function λ . Then, apply control sequence λ_2 . λ_2 operates in the same way as λ , but it takes S' as input instead of S. The next state from the previous cycle becomes the current state of this cycle.

Step 7: Read the output from the output cell.

Step 8: After the output is read, execute the transition function. Control sequence δ_2 is applied, and the next state is stored in cell S.

Step 9: The machine loops back to step 1.

Fig.44 illustrates a single cycle of state transition in the state machine using ISD notation.

The memristor cell, which stores the next state, is updated during the execution of the combination logic period. This updated value is held for a brief period, allowing for reading before transitioning to the next state.

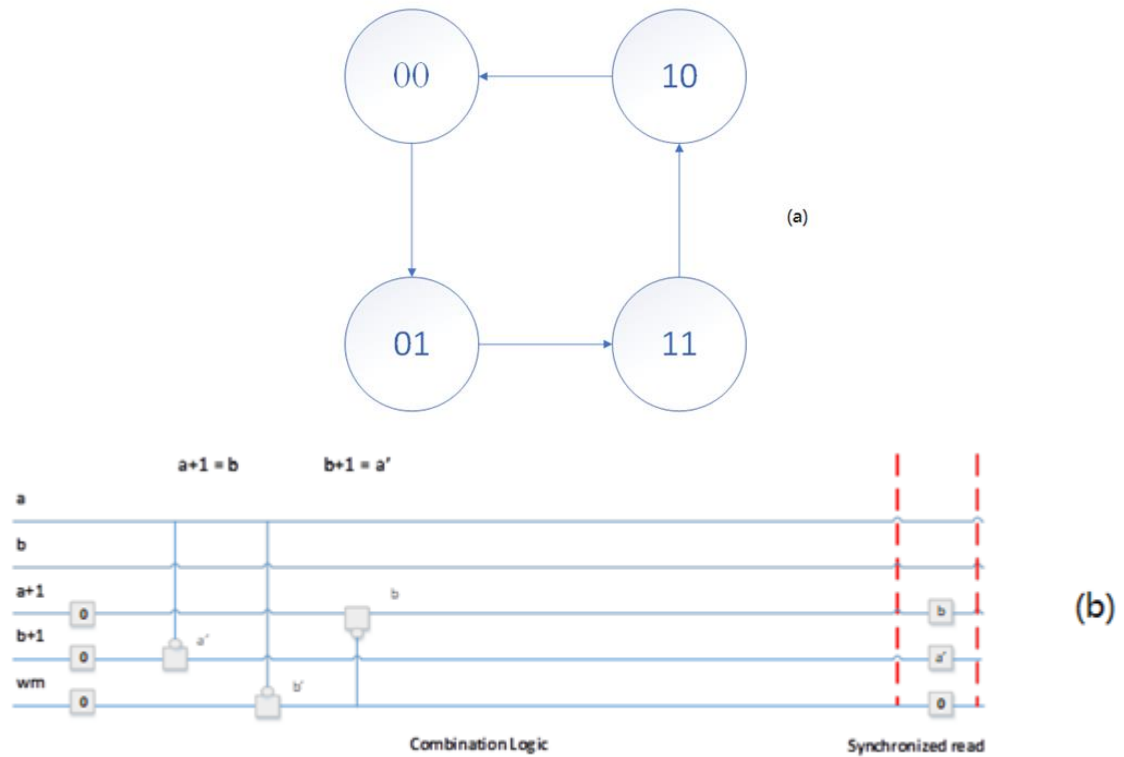


Fig. 44 (a) state transition diagram for a given FSM, (b) the memristor realization represented with ISD.

5.3 Background for the state assignment problem and related work

The complexity of a state machine implemented with sequential circuits is influenced by the way in which states are assigned binary codes. These codes are used to determine the combinational logic and transitions between states. The control unit of such a state machine is composed of synchronous sequential circuits with combinational logic blocks and memory elements, all of which are synchronized by a clock. The combinational logic blocks compute outputs and update memory element values at each clock cycle. A sequential circuit can be modeled as a finite state machine (FSM) with inputs, outputs, and internal states, where each state represents the information stored in the memory elements at that point. In order to synthesize a sequential circuit from an FSM, a crucial

step is the state encoding or state assignment (SA), which involves determining the minimum number of binary digits (bits) needed to encode the states and assigning a unique binary code to each state. The minimum number of bits required is calculated as $\log_2(S)$, where S is the number of states in the FSM. The way in which states are encoded has a significant impact on the complexity of the synthesized sequential circuit, affecting its size, speed, and power consumption.

Several deterministic algorithms have been developed to solve the state assignment (SA) problem. These algorithms aim to minimize the number of product terms in two-level circuits and the number of literals (i.e., variables in true or complement form) in multi-level circuits. One early technique for SA is the Hartmanis method [98], which relies on state partitioning. However, this technique may not work well for all FSMs as some may not have well-behaved closed partitions. Other techniques for SA in two-level circuits include KISS [98] and NOVA [99], which use symbolic minimization. Techniques for SA in multi-level circuits include MUSTANG [100], JEDI [101], and MUSE [102], which use heuristics to maximize the factoring of expressions in order to reduce the number of literals in the resulting circuit. Examples of deterministic techniques for power optimization include the work in [103]. The work in [103] assigns state codes with a Hamming distance of 1 to states with the highest transition probability, while the work in [104] uses integer linear programming to optimize the minimization of flip-flop switching activity.

Given the complexity of the SA problem and the limitations of deterministic algorithms, non-deterministic evolutionary algorithms such as genetic algorithms [105], simulated

annealing [106], tabu search [107], simulation evolution [108], and other evolutionary algorithms [109] have also been used to address it.

The FSMs, in their initial specifications, have don't cares and may be minimized with respect to states and inputs. I compared my methods with the well-known state assignment and logic realization algorithms for memristive FSM from the past.

Finite State Machines (FSMs) are commonly used in chip design as control units for circuits in modern CAD VLSI systems and also have other applications, such as counters and sequence recognizers. These FSMs can be implemented using different types of logic layouts, such as PLAs or gate matrices. The initial specifications of FSMs may contain "don't cares" and can be minimized with respect to states and inputs. In this dissertation, I compared my methods for implementing FSMs using memristive technology with previously established state assignment and logic realization algorithms. The principles of state machines are still relevant in synthesizing circuits using various nanotechnologies, including memristors.

The minimum solution for SA problem can often be found using partition theory. This involves dividing the total number of internal states (K) of a machine into partitions, such that $2^{k-1} < K \leq 2^k$. These partitions are known as proper partitions, as they consist of two blocks where the number of states in the larger block does not exceed 2^{k-1} . The proper partitions are selected from a set (TP) in a way that minimizes the total number of variables in the machine's transition functions. The goal becomes the minimize number of flip-flops in the CMOS-based state machine. And for the memristive state machine, the goal becomes the minimization of the number of control pulses. This condition is

expressed in selecting set TF (called the ***final family of partitions***) from set TP of such partitions τ_i , that if $\pi_i \rightarrow \tau_i$ and $\pi_i \leq \tau_{i,1} \cdot \tau_{i,2} \dots \tau_{i,n}$ then

$$\sum_{\{\tau_i \in TF\}} \sum_{\{\tau_{i,1}, \dots, \dots \tau_{i,n}\}} \text{CARD}\{\tau_{i,1}, \dots, \dots \tau_{i,n}\} = \min,$$

$$\text{where } \forall i, \tau_i, \tau_{i,1} \dots \tau_{i,n}, \tau_{i,1}, \dots \dots \tau_{i,n} \in TF.$$

(For the meaning of partition calculus symbols \rightarrow , \leq and \cdot see chapter 3).

The final family (set) of partitions, TF , is defined as such a family, that the partition product of all partitions from that family is a zero partition. **The optimum family, TO** , is such a final family that minimizes the value of an assumed cost function.

The family TO in my approach is chosen among all two-block partitions - this gives a possibility of finding better solutions but is computationally less efficient than using the proper partitions.

Let us denote the zero partition by $\bar{0}$:

$$\bar{0} = \{\{1\}, \{2\}, \dots, \{K\}\}$$

The problem of state assignment of the **minimum** synchronous FSM can be formulated as follows:

Given are:

- a) A - the set of internal states of the FSM M ; $K = \text{CARD}(A)$,
- b) $T2$ - the set of all two-block partitions,
- c) the flow table and the output table of the FSM.

The set TO to be found is such a subset of the $T2$ set, that both of the following conditions are satisfied:

$$\prod_{\tau_i \in TO} \tau_i$$

$$\prod_{i=1}^n \tau_i$$

1. $\prod_{i=1}^n \tau_i \{\tau_i \in TO\} = \tau_1 \cdot \tau_2 \cdot \dots \cdot \tau_i = \bar{0}$
2. $\prod_{\{\tau_i \in TO\}} \tau_i = \tau_1 \cdot \tau_2 \cdot \dots \cdot \tau_r = \bar{0}$. That is, the partition product of the selected partitions is a zero partition, which means that a different binary sequence is assigned to each state.
3. $\text{Min}(CF_r) = \min(\sum_{\{\tau_i \in TO\}} q_r(\tau_i) + \sum_{\{y_i \in \{y_1, \dots, y_v\}\}} q_r(y_i))$, where $\{y_1, \dots, y_v\}$ is the set of output signals.

This is called a condition of cost function minimization. Conditions 1 and 2 ensure that a distinct binary code is assigned to each state. Condition 3 guarantees that the overall cost for implementation is kept to a minimum.

By $q_r(\tau)$ I denote the real costs of the realization of the τ partition as one of the TO set. The costs are calculated for the respective circuit realization. The selected gates and the respective structure of the excitation functions' realization. By $q_r(y_i)$ I denote the cost of realization of an output signal y_i .

The state assignment method uses a combination of heuristic search and branch-and-bound techniques to efficiently search for solutions. This approach limits the search space to a subset of all possible two-block partitions. The quality functions are employed to guide the search strategy and ensure that a minimum solution is not missed.

5.4. State assignment of Memristive FSM

The main idea of the method of state assignment proposed in this dissertation is to select such partitions for a not necessarily minimum FSM that the imply gate cascade for

memristor realization of the excitation functions is minimized according to some cost function.

Let *IMPLY* be a relation of implied compatibility of **blocks** (blocks are groups of internal states). Let $TF = \{\tau_1, \tau_2, \dots, \tau_v\}$ be a set of partitions.

Definition 5.1.

Separation condition states that each pair of machine's incompatible states is separated by at least one partition from *TF*.

Definition 5.2.

Closure condition states, that groups, implied according to relation *IMPLY* by the blocks from the product of partitions from set *TF*, are included in those blocks.

Definition 5.3.

Every set *TF* that satisfies the separation condition and the closure condition will be referred to as the **final family of partitions**.

A solution to the state assignment problem consists of finding the family *TO*. This family should be such a subset of *T2*, that each of its partitions separates at least one pair of incompatible states. To check the closure condition, the set of all pairs of compatible states should be created and for each of its elements, the respective compatibility conditions must be found. Please note that the existing algorithms encode the already minimized machines, and the state-minimization process is executed separately before the state assignment process. This leads to not necessarily the least cost solution. My algorithm allows to find better solutions because it combines the stages of minimization and encoding, thus it selects the best of all machines that can be obtained by partial or

complete minimization processes of the initial non-minimal machine. My presented approach is the only one in the literature that the stages of state minimization, state encoding, and function realization for a given technology are combined into a single process. This allows for a superior minimization of the final resultant memristive logic of the machine.

5.4.1 Operators and relations on cubes

The terms used for the operators and relations in this chapter were chosen based on their specific use in encoding the rows and columns of a flow-table for a finite state machine.

•	0	1
0	0	1
1	1	∅

Table 14 The operator of row compatibility

The operator of row compatibility, denoted by \circ , is defined for cubes a and b (a and b are n -tuples of elements $a_i, b_i \in \{0, 1\}$), as follows:

$$\emptyset \text{ if } (\exists i) [a_i \circ b_i = \emptyset]$$

$$a \circ b = c = (c_1, c_2, \dots, c_n) \text{ in the opposite case}$$

where $(\forall i = 1, \dots, n) [c_i = a_i \circ b_i]$ in accordance with the table. 5.3.

Examples

$$1100 \circ 0010 = 1110,$$

$$0101 \circ 0001 = \emptyset \text{ because } a_4 \cdot b_4 = \emptyset.$$

The operator of columns compatibility, denoted by ∇ , is defined for cubes a, b (of elements $a_i, b_i \in \{0, 1, X\}$), as follows:

∇	0	1	X
0	0	X	\emptyset
1	X	1	\emptyset
X	\emptyset	\emptyset	X

Table 15 The operator of columns compatibility

Γ	0	1
0	0	ε
1	\emptyset	1

Table 16 The relation of rows absorption

The relation of rows absorption, denoted by Γ , is defined for cubes a and b ($a_i, b_i \in \{0, 1\}$).

\perp	0	1	X
0	ε	\emptyset	ε
1	\emptyset	ε	ε
X	\emptyset	\emptyset	ε

Table 17 The relation of columns absorption

The relation of columns absorption, denoted by \perp is defined for cubes a and b

5.4.2 generation of prime implicants of multi-valued function

I will begin by assuming that the number of partitions is minimal. Let us consider a flow table of the FSM with the inputs already encoded (in the columns). To find the excitation function from this table, the code for each cell of the table is required. The code for each cell is a concatenation of the row's code and the column's code. However, the state assignment is not yet known and, therefore, neither is the row's code. My goal is to select an auxiliary (transitory) code for the rows, which will enable the designer to explore all possibilities of minimizing the excitation functions (i.e., covering minterms with prime implicants) before determining the final state assignment of the internal states of the table. This code will allow the examination of all possibilities of joining groups of minterms into prime implicants, i.e., applying logic adjacency to minterms from each pair of rows.

Suppose I have encoded a flow table according to some proper partition τ_j . Several possibilities of including rows into prime implicants exist now. This results from the fact, that I assume the possibility of arbitrary permutation of the rows, such that the other partitions are assorted in the best possible manner with τ_j , to minimize the *multi-valued function*, described by a flow table encoded with τ_j (the encoded flow tables will be called *transition tables*). To formalize such *generalized joining of rows into groups* I must find the respective code for the rows. For K rows of the table I select a code of length K , where 1 is written to the i -th row on the i -th position. For example, three internal states are encoded as follows: **s1** - 100, **s2** - 010, **s3** - 001. Actually, every cell of the flow table (i.e., a 0-cube) can be described by means of the binary string: $c_i = c_\alpha^i \mathbf{0} c_\beta^i$

where c_α^i is the machine's inputs cube in Gray code, and c_β^i is the cube of internal states, in the code described above. The symbol \circ stands for concatenation. The operations may be utilized to join the cubes into greater cubes. Both constituents of the string are used.

There may be $n_1 < K$ arbitrary rows in each group after joining. The adopted code is the simplest one with which this property can be checked. All the cubes generated by means of joining have the part corresponding to the inputs and the part corresponding to the internal states. The implicants of the multi-valued function will be called "*generalized implicants*" (*abbrev. G-implicants*). The G-implicants which are prime will be called "*generalized prime implicants*" (*GP-implicants*). Let ON be the set of minterms and DC be the set of "don't-cares" of the Boolean function, which corresponds to the flow table, whose states are assigned in accordance with the chosen partition τ_j . I shall find the set of all GP-implicants, and next the covering of minterms with the GP-implicants.

For simplification, I assume that the algorithm to generate GP-implicants starts with $ON \cup DC$. Then the operator of the column compatibility is applied to all cubes from $ON \cup DC$, that have identical cubes of internal states. Next the operator of rows compatibility is applied to the cubes with identical input cubes. Each of the cubes created in this way is stored in a new array. Having both operators applied in the newly created array, the rows absorption relation and the columns absorption relation are applied and the absorbed cubes are removed from the array. The outcomes are stored in the new array. Operators of compatibility and relations of absorption are carried out iteratively until the old array, and the new array become identical. Next, the GP-implicants that cover only don't cares

are removed. The entire process is a kind of generalization of the Quine-McCluskey algorithm for the case of multi-valued function with one MV variable and don't cares.

Observe also that this method can be directly used to calculate the cost of encoding with the "one-hot" code. This is done by taking directly one hot partition with a single state and for blocks of partitions with more than one state, creating hardware sums of the single states that exist in these blocks.

The generation of GP-implicants is described by means of the following algorithm.

Algorithm 5.1.

Generation of GP-implicants for an MV-function with one MV variable and don't cares.

Begin

1. Find set ON of minterms and set DC of don't care of the function. These sets include 0-cubes $c_i = c_{\alpha^i} \mathbf{o} c_{\beta^i}$, encoded as explained above for the given partition t_j .

2. $P := ON \cup DC$

3. Apply columns compatibility operator to all the pairs of cubes from P, which have the same sub-cubes of internal states:

$$P1 := \{a \nabla b \mid a, b \in P \text{ and } C_{\beta^a} = C_{\beta^b}\};$$

4. Apply rows compatibility operator to all the pairs of cubes from P, which have the same input sub-cubes: $P2 := \{a \mathbf{o} b \mid a, b \in P \text{ and } C_{\alpha^a} = C_{\alpha^b}\};$

5. Add P1 and P2 to P: $PP := P \cup P1 \cup P2;$

6. Delete the G-implicants absorbed with respect to columns:

$$PP := \{a \in PP \mid (\exists a' \in PP) [a \bar{a}']\}$$

7. Delete the G-implicants absorbed with respect to columns:

$$PP := \{a \in PP \mid (\exists a' \in PP) [a _ a']\}$$

8. **If** $P = PP$ **then go to** 11;

9. $P := PP$;

10. **Go to** 3;

11. Remove the G-implicants from PP, which covers only the don't care cubes or are absorbed by other G-implicants. New PP is the set of all GP-implicants, i.e., those that can be included in the minimum cover.

12. End.

The cubes found as above form the set of GP implicants of the multi-valued Boolean function with a single MV variable. These implicants correspond to realizations of the excitation functions, encoded with respect to the best hypothetical code, assorted with the given partition τ_j . By the best code I understand the set of all possible partitions that minimize the cost of the excitation function for j . I assume here that for partition τ_j all other partitions were selected in the optimum way. The selected in the best possible manner partitions, together with τ_j , of course, do not have to be the final family of partitions. Any multi-valued minimizer, such as Espresso-MV [110] can be used to generate good covering with GP-implicants. A step-by-step demonstration of Algorithm 5.1 is shown as follows:

Example 1:

1. Suppose I encoded a flow table according to some proper partition $\tau_{13}=0$.

Flow table	00	01	11	10
1	1	1	2	2
2	1	1	1	4
3	1	1	4	4
4	1	1	3	2

↓

$\tau_{13}=0$	00	01	11	10
1	0	0	1	1
2	0	0	0	1
3	0	0	1	1
4	0	0	0	1

Fig. 45 encode the next state in the transition table with partition.

- For K rows of the table, I select a code of length K , where 1 is written to i th row on the i -th position. The encoded flow tables will be called *transition tables*.

	00	01	11	10
1000	0	0	1	1
0100	0	0	0	1
0010	0	0	1	1
0001	0	0	0	1

Fig. 46 encode the current state with one-hot code.

- The next task is to find the set of all GP-implicants and the covering of minterms with the GP-implicants. Apply Algorithm 5.1.
- Apply columns compatibility to get P1.

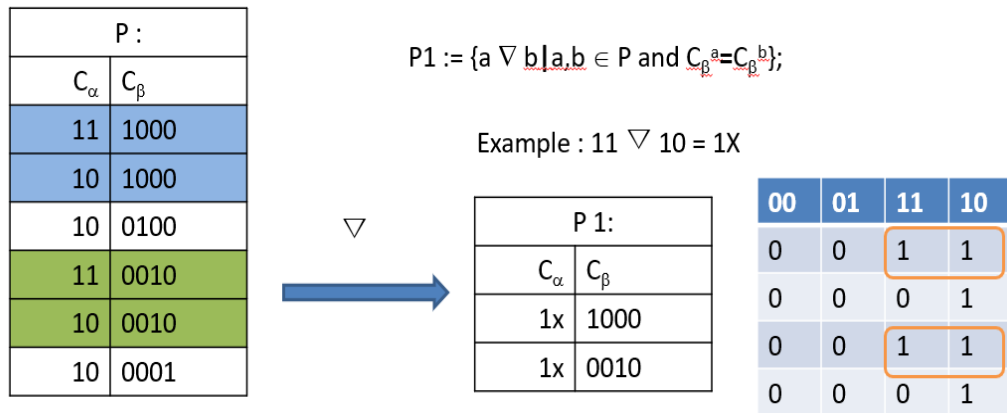


Fig. 47 encodes the current state with one-hot code.

5. Apply row compatibility to get P2.

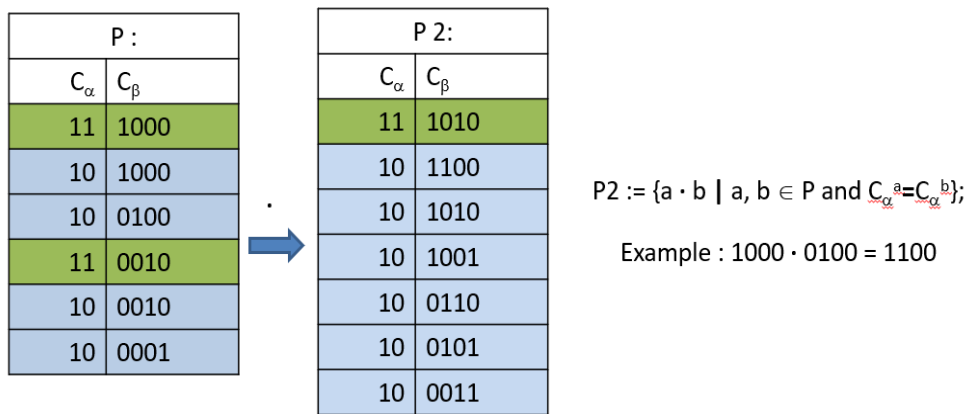


Fig. 48 demonstration of applying row compatibility.

6. Add P1, P2, and P to derive PP.

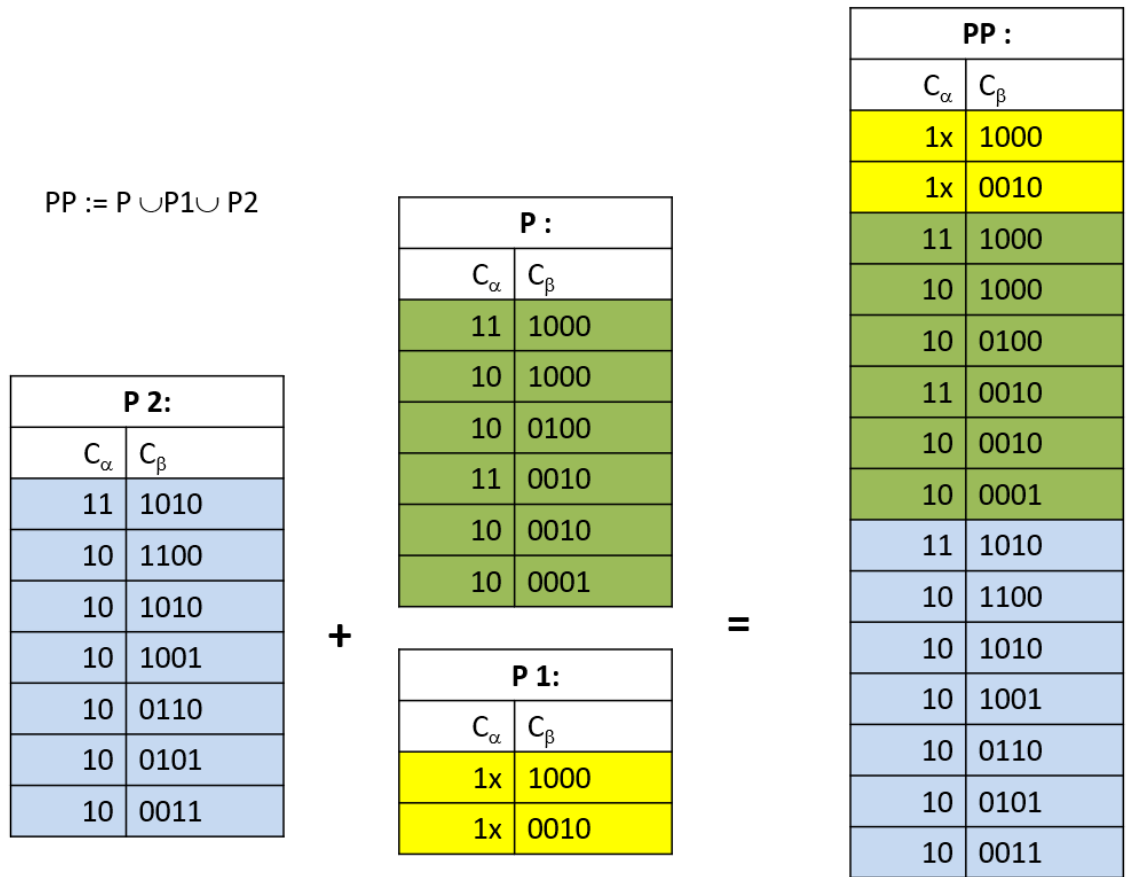


Fig. 49 Union P1, P2 and P3.

7. Delete the G-implicants absorbed with respect to rows and columns.

PP :	
C_α	C_β
1x	1000
1x	0010
11	1000
10	1000
10	0100
11	0010
10	0010
10	0001
11	1010
10	1100
10	1010
10	1001
10	0110
10	0101
10	0011

PP :	
C_α	C_β
1x	1000
1x	0010
11	1010
10	1100
10	1010
10	1001
10	0110
10	0101
10	0011

Fig. 50 Demonstration of deleting the G-implicants.

8. Repeated steps 4 to 7 until $P = PP$.
9. Found GP-implicants

1x 1010

10 1111

The cubes found to form the set of GP implicants of the multi-valued Boolean function with a single MV variable. These implicants correspond to realizations of the excitation functions in SOP, encoded with respect to the best hypothetical code, assorted with the given partition τ_j . Then it is converted to Imply gate cascade with synthesis tools [90].

$D = A\bar{Q} + A\bar{B} = A \rightarrow (BQ) \rightarrow 0$. And the Imply gate cascade is shown in ISD form in Fig.51.

$\tau_{13} = 0$	AB			
	00	01	11	10
1000 (S_0)	0	0	1	1
0100 (S_1)	0	0	0	1
0010 (S_0)	0	0	1	1
0001 (S_1)	0	0	0	1

1x 1010
10 1111

With $\tau_{13} = 0$

excitation functions : $D_{13} = AS_{1,3} + A\bar{B} = A\bar{Q}_{13} + A\bar{B}$

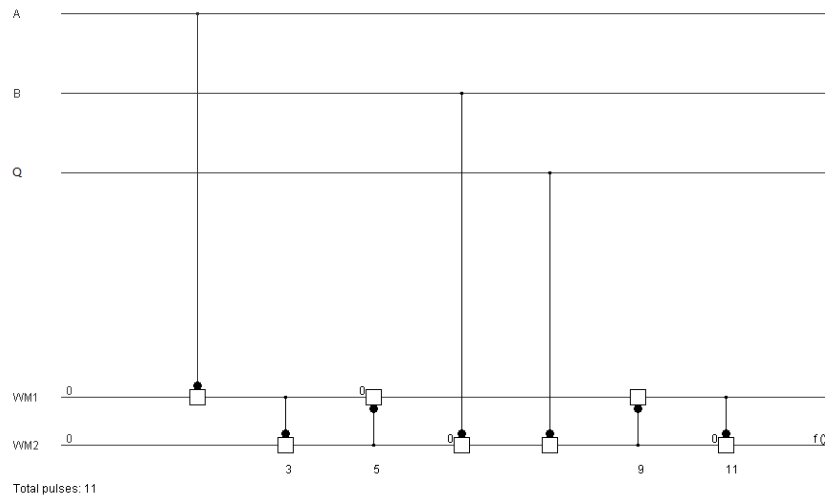


Fig. 51 ISD for excitation function.

Quality function for the set of partitions $\{\tau_1, \dots, \tau_r\}$ will be defined as follows:

$$QF(\{\tau_1, \dots, \tau_r\}) = \sum\{\tau_i \in \{\tau_1, \dots, \tau_r\}\}q(\tau_i) + QF_{OUT},$$

Where QF_{OUT} is the total cost of realization of the outputs, it is calculated as a sum of costs $q(y_i)$ for all output signals (each cost $q(y_i)$ is calculated analogically to $q(\tau_i)$ cost, for the case of selecting the best assorted with it additional partitions). QF_{OUT} is calculated once and for all for the given machine, and it is independent of the family $\{\tau_1, \dots, \tau_r\}$.

Algorithm 5.2

Calculation of the quality functions for τ partition.

Begin

1. Encode the first block of τ partition with 1 and the second block with 0;
2. Calculate excitation functions and find minimum cover with GP-implicants, run Memristor synthesis program to generate Imply gate cascade to derive the cost for selected partition.
3. Encode the first block of τ partition with 0 and the second block with 1;
4. Execute step 2.
5. Select the minimum of the costs found in steps 2 and 4 and save the assignment of the partition's blocks to symbols 0 and 1.

End

Cost calculation based on the size of the implicant, I will show an example of the scenario, which demonstrates the process of computing the cost for a given partition.

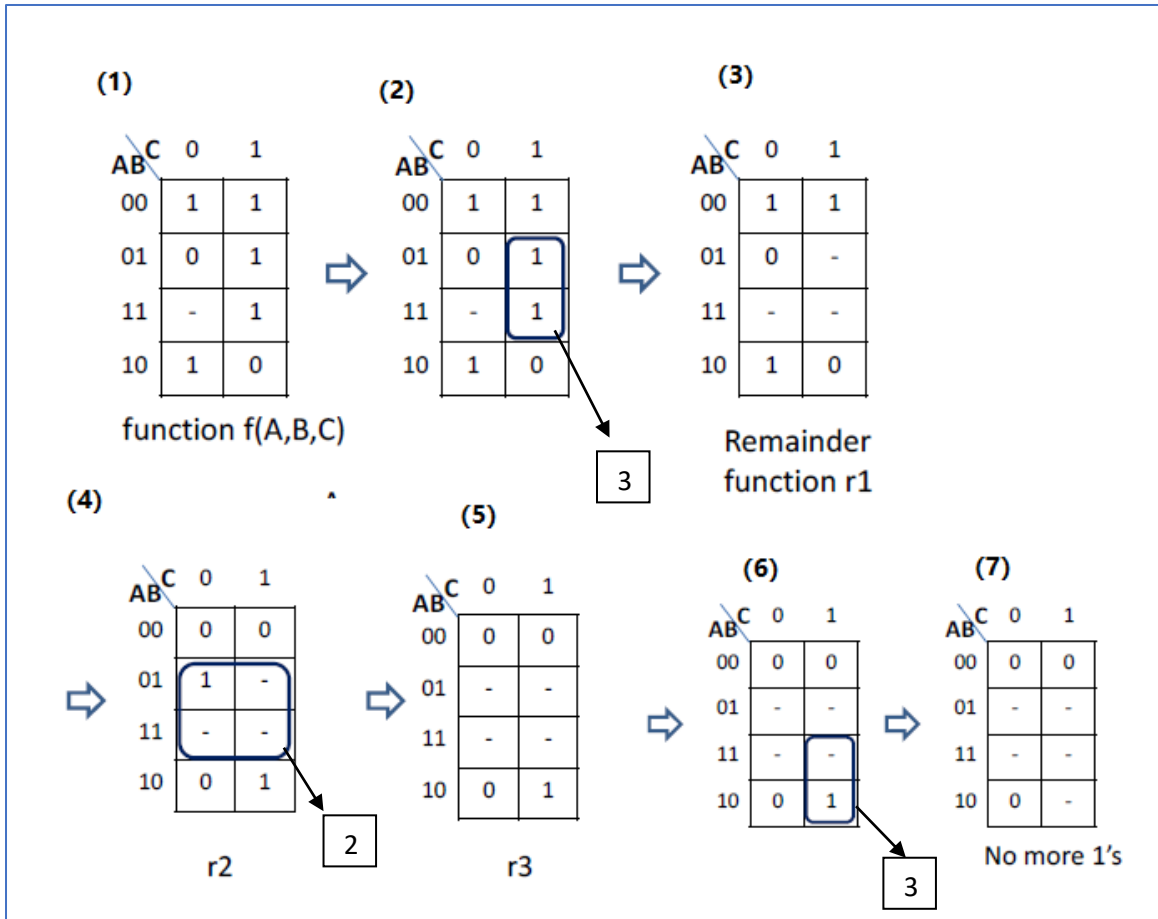


Fig. 52 cost computation with synthesis for an excitation function.

The first step of synthesis and cost computation is to find the group of 1's in which the input literals are all positive. One such group is selected. In the example, this is BC. It is an essential positive prime, as there are no other positive primes covering minterm ABC. BC is realized as a NAND gate and hooked to the negative input of the IMPLY gate. So, at the end of this step - $f(A, B, C) = r1 + BC$, where $r1$ is the first remainder function. This is shown in Fig. 52(1). Now I move on to realize the first remainder function, $r1$. I realize that no groups of 1's correspond to positive variables. As there are no more positive primes, this completes the synthesis of the first layer. I must now negate the entire K-Map, as shown in Fig.52(3). In the synthesis software, the negation of the K-

Map is simply reversing the roles of the temporary Onset and Offset tables. At the end of this step, the function is realized by $f(A, B, C) = \overline{r_2} + BC$. I can now find another group of 1's (and don't cares), which becomes the next positive prime implicant. This is shown in Fig.52(4). The four middle squares are selected. These correspond to a positive essential prime B . Since this is a single variable, I put a NOT gate instead of NAND gate as shown. At the end of this step, the function realization is $f(A, B, C) = \overline{r_3} + B + BC$

I still have a '1' left in the K-map. To continue the process, I now select the remaining possible group (AC), as shown in Fig. 52(6). Group AC is again a positive essential prime. Since no remainder function is left, I put a 0 at the remaining input of IMPLY gate. The final function is now equivalent to $f(A, B, C) = \overline{(AC + B)} + BC$. And the Cost of this function is the sum of the cover in Fig. 52 (3), Fig.52(4), and Fig.52(6). $QF = 3 + 2 + 3 = 8$.

State assignment of an FSM is done with the following algorithm.

Algorithm 5.3

Begin

1:

Creation of the initial state of the solution tree:

$N := 0; V(0) = (QS(0), GS(0)) := (\{\}, A); T2 := \{\}; OPEN := \{V(0)\};$

2:

If $OPEN = \{\}$ **then return;**

$V(N) = (QS(N), GS(N)) :=$ first element from list OPEN;

3:

If $GS(N) = \{\}$

then

begin

delete first element from list OPEN;

go to 2

end ;

4:

$s :=$ first element from $GS(N)$; $QS(NN) := QS(N) \cup \{s\}$; $GS(NN) := GS(N) - \{s\}$;

5:

add $(QS(NN), A - QS(NN))$ to the beginning of list T2.

6:

Add new state $NN = (QS(NN), GS(NN))$ to the end of list OPEN;

7:

Go to 3;

End

A step-by-step demonstration of Algorithm 5.3:

1. List T2 after arrangement in Step E2 of Algorithm 3.1 has the following form: $T2 = \{\tau_{14}, \tau_{13}, \tau_{12}, \tau_1, \tau_3, \tau_2, \tau_4\}$. The partition is ordered based on quality function $q(\tau_i)$

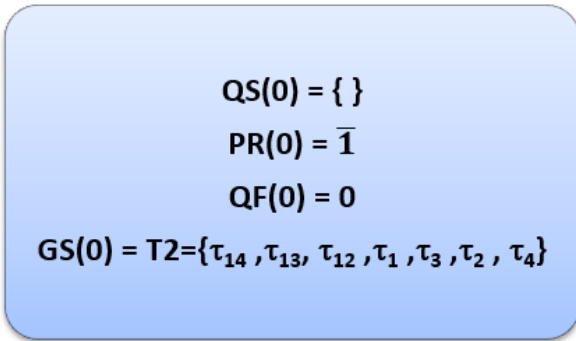


Fig. 53 Step 1.

2. Execute operator τ_{14} and generate node $QS(1)$; $RP(1) \neq 0$, execute τ_{13} , and generate node $QS(2)$; $PR(2) = 0$, Found solution $\{\tau_{14}, \tau_{13}\}$ (with cost $CF_r^3 = 17$).
 $Cf_{\min} = 17$.

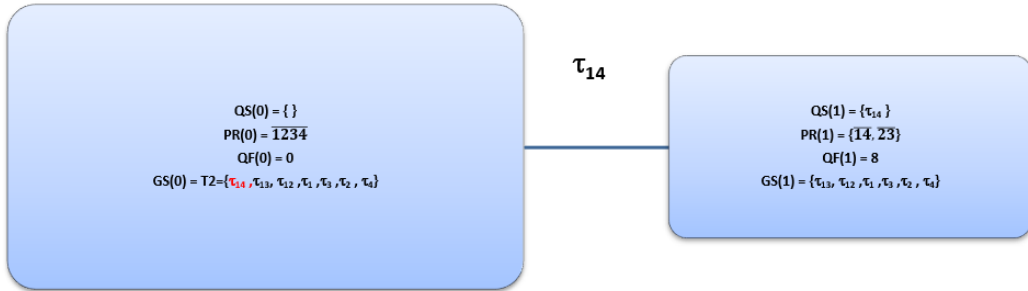


Fig. 54 Step 2.

3. Execute operator τ_{14} and generate node $QS(1)$; $RP(1) \neq 0$, execute τ_{13} , and generate node $QS(2)$; $PR(2) = 0$, Found solution $\{\tau_{14}, \tau_{13}\}$ (with cost $CF_r^3 = 17$).
 $Cf_{\min} = 17$.
4. Backtracked to node 1, the operator τ_{12} is deleted from $GS(1)$ because $11 + 8 > 17$. Analogically, operators τ_2 and τ_4 are deleted from $GS(1)$.

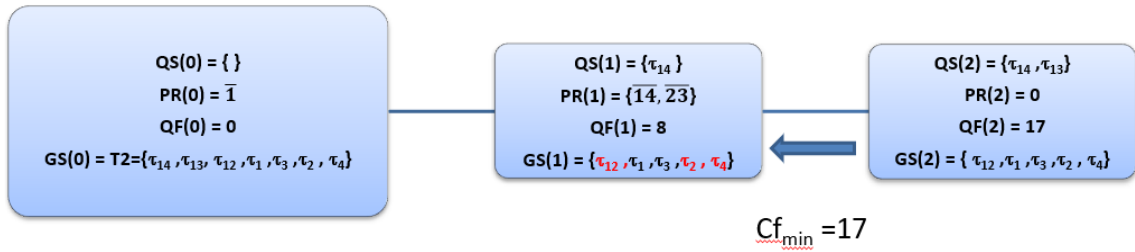


Fig. 55 Step 3.

5. Execute remaining operators to generate new nodes 3. $PR(3) > 0$. It is not a solution. $QF(3) + QF(\tau_1) > QF_{\min} = 17$, Stop searching and remove τ_3 , $GS(3) = \{ \}$, back track to $QS(1)$.

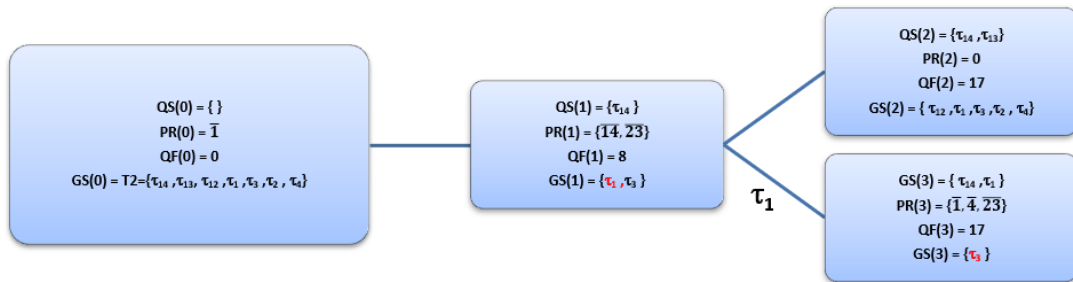


Fig. 56 Step 4.

6. Execute remaining operators to generate new nodes 4. $PR(4) > 0$. It is not a solution. $QF(4) + QF(\tau_3) > QF_{\min}$, Stop searching and back track to $QS(1)$. back track to $QS(1)$, found $GS(1) = \{ \}$, back track to $QS(0)$.

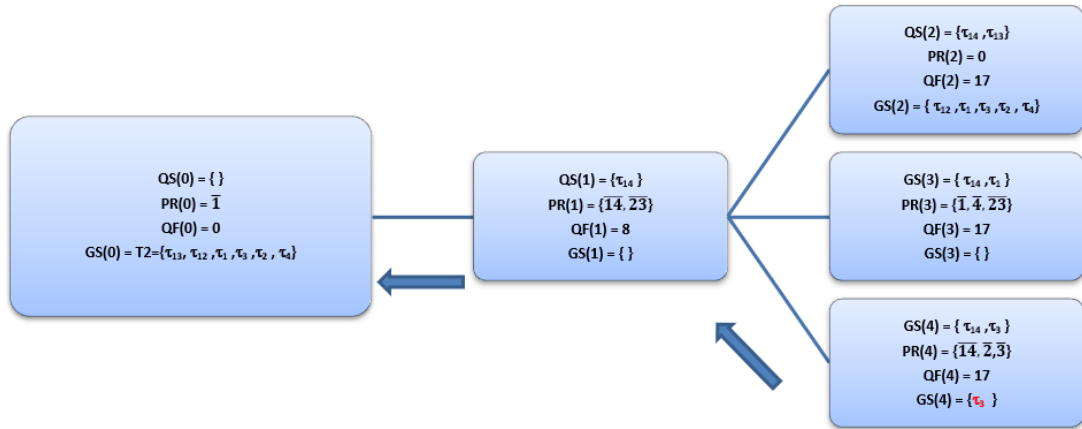


Fig. 57 Step 5.

7. Now program back tracks to node 0. Node 5 is generated. Operators τ_{12} , τ_1 , τ_3 , τ_2 and τ_4 are removed from $GS(5)$. Set $GS(5)$ becomes empty; this results in the backtrack to node 0.

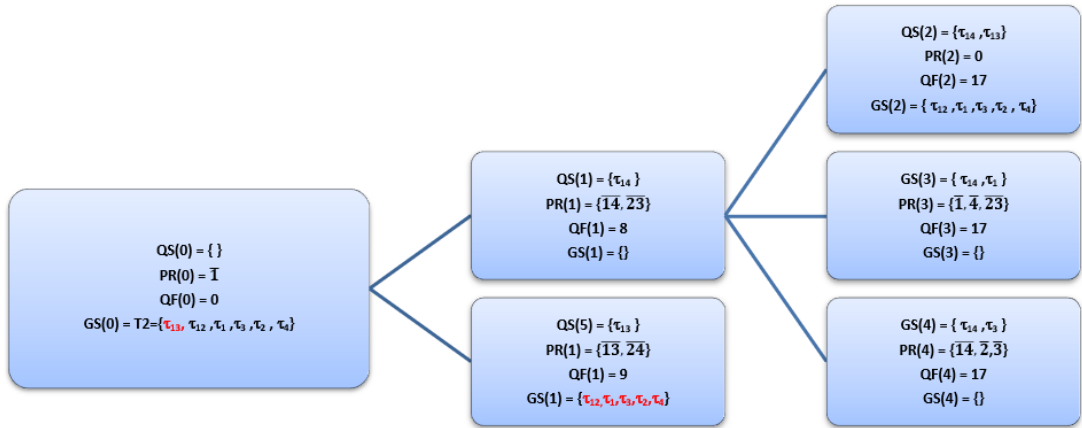


Fig. 58 Step 6.

8. Backtrack from node 10 results that $GS(10) = \{ \}$ and $PR(10) \neq \bar{0}$. Therefore, the optimum solution is $\{ \tau_{14}, \tau_{13} \}$.

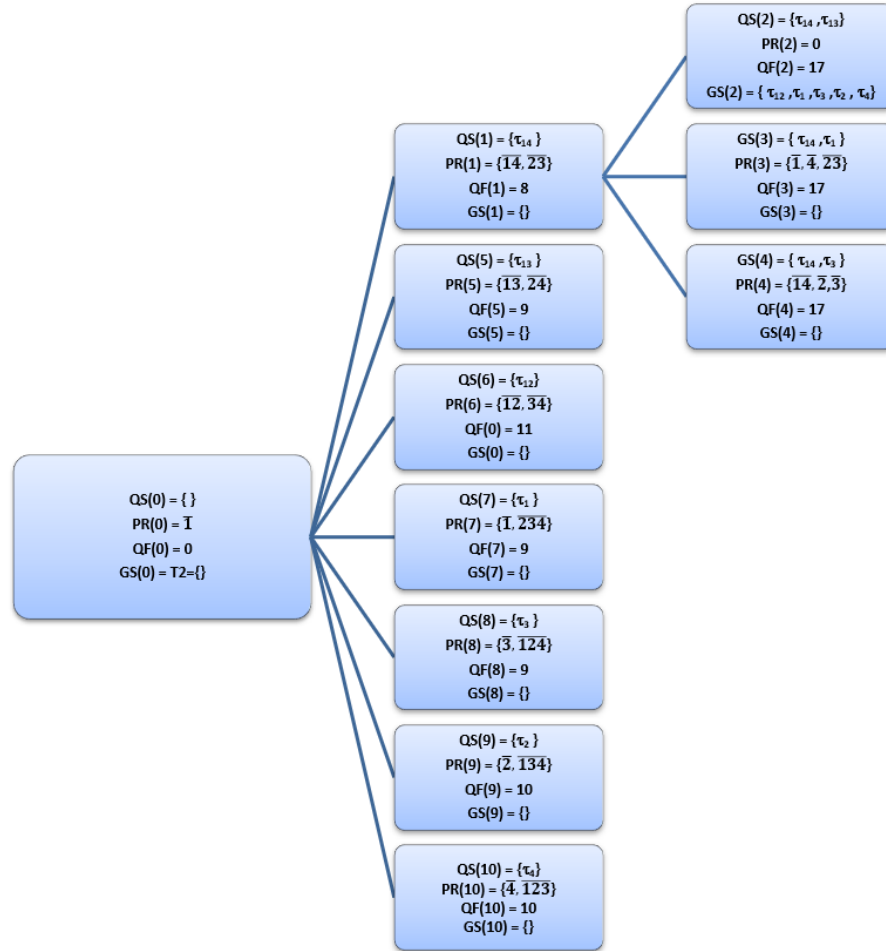


Fig. 59 step 7.

5.5 Experimental Results

My goal is to find the encoding of a state machine (not necessarily the completely minimized machine) that this encoding minimizes the cost of the realization of this machine with the selected by me variant of memristive technology. Therefore, in this chapter of my dissertation, I will compare various design tools for Finite State Machines (FSMs), including JEDI, NOVA, one-hot encoding, and the collaborating with them memristor synthesis tool [90]. I explore thus the use of Boolean logic synthesis specific

to memristors and state assignment techniques built for this synthesis cost function in order to optimize the entire design process for memristive FSMs. Such a problem was not formulated or solved before. Table 18 compares the number of imply gates required to implement each state machine in benchmark circuits from MCNC and ISCAS. The results show that the pulse count in bold represents the most cost-effective design, and the runner-up is highlighted in red.

	My metho d 2WM	My metho d SOP	Nova 2W M	Nov a SOP	nova ESO P	Jedi SO P	Jedi 2W M	JEDI ESO P 2W M	OH SOP 2	OH 2W M
kirkman	320	542	580	455	1645	771	1243	1542	698	443
Tbk	634	776	1044	898	994	651	822	854	743	638
donfile	145	220	341	184	1180	364	360	1026	651	515
Bbara	115	135	354	197	1411	115	216	1022	410	543
dk15	90	77	277	177	524	173	171	661	193	321
s27	56	87	49	77	406	266	98	352	155	288
Bbtas	38	65	164	61	248	158	109	243	127	157
lion9	30	39	103	48	354	150	142	562	79	99
modulo1 2	58	88	141	73	495	155	145	442	297	356
shiftreg	17	32	16	16	162	78	44	108	85	99
train4	22	31	26	29	66	55	50	50	51	74
Lion	22	35	30	30	75	32	54	44	60	72
Seqd	16	16	31	21	48	26	17	32	19	22

Table 18 The pulse count of permissive state machine in benchmark circuits from MCNC and ISCAS.

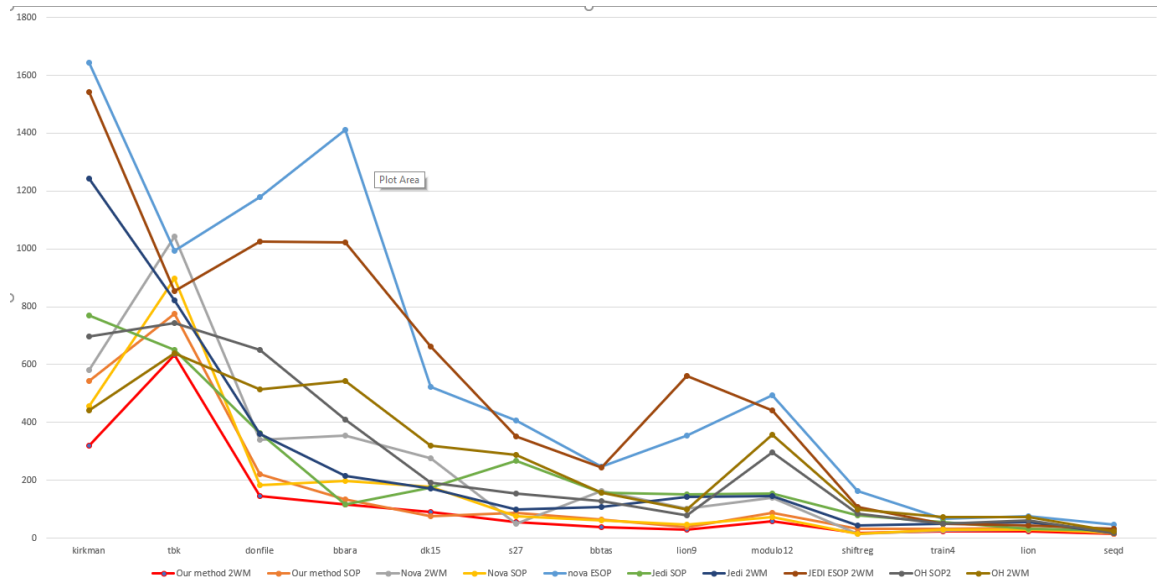


Fig. 60 Curve plot for table 18.

The development of efficient design methods for memristive state machines is a crucial but overlooked area of research. I have proposed a general model for designing memristive circuits as state machines. While existing minimization and encoding tools for traditional FSMs can be used, specialized software that takes into account the unique properties of memristive state machines and logic should be developed. I have combined various Computer-Aided Design (CAD) tools to create a comprehensive system for realizing memristive FSMs using IMPLY and CLEAR gates. My initial benchmarking results, as shown in Fig.60, indicate that the choice of state assignment and minimization tools can greatly impact the realization cost of the circuit.

Chapter 6 Memristor-based pulse rate system

There is in recent years an increased interest in designing systems that are not traditional binary logic nor traditional analog design. Such systems are built for neural networks, control, and machine learning. Among other technologies, both quantum and memristive technologies are used to build these innovative systems in which possibly numbers are represented not in a standard way characteristic for binary logic. Here I assume memristors in their variant presented already in this dissertation. The question is how I can solve various algebraic and differential equations very quickly and for extensive data but not necessarily very precisely. Such problems exist in several areas, as differential and algebraic linear and non-linear equations must be solved in many practical applications.

The pulse rate system is a method of implementing computation systems that utilize the rate at which a signal occurs, rather than its electrical characteristics, to represent and process information. This approach, known as pulse-rate signal computing, has become increasingly popular recently and was first proposed in the 1960s as a cost-effective alternative to traditional binary computing. It enables complex arithmetic operations to be performed using simple logic.

6.1 Pulse rate measurement systems and related research

When a digital code is used to represent a number, each digit is weighted in significance. For example, the most significant digit of a 10-digit binary word would represent a weight of 512, while the least significant digit would represent a weight of only 1. This

means that each digit must be handled separately in order to preserve its significance, which can lead to complicated timing arrangements and complex gating configurations. With pulse-rate signals, the important quantity is an averaged parameter, which means that individual gating arrangements are less critical. This is one of the properties that make pulse-rate signal processing an attractive option.

In this chapter, I will only consider clocked pulse-rate sequences as they are easier to implement in processing devices. I will focus on a pulse-rate signal (A), where the probability of a pulse occurring at a particular clock time is represented by $P(A)$,

$$P(A) = \lim_{n \rightarrow \infty} \left(\frac{m}{n} \right)$$

Where m is the number of pulses recorded in an interval of n clock pulse, by the definition of pulse-rate signal, the probability $P(A)$ should be a constant [92]. When the pulse-rate signal is used to represent an integer, m is the integer, and n as the capacity, which means the largest number this pulse can represent. For example, signal A, which represents the number 3 with a capacity of 4, is shown in Fig.61.

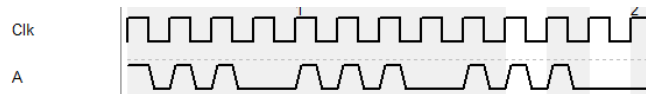


Fig. 61 Pulse rate signal with clock.

Pulse rate signals can be divided into two categories: stochastic signals, also known as random pulse sequences, and regular pulse-rate signals [93]. Fig. 61 illustrates an example of a regular signal, as the pulses occur at consistent intervals of clock pulses. In

my implementation, I chose to use a clocked regular pulse signal for ease of control and to eliminate the need for additional circuitry to generate a truly random bit stream.

6.2 Development of memristor-based components for a pulse rate system

The system consists of two types of blocks, namely the rate multiplier (RM), and the reversible counter (RC). Fig. 62 shows the Rate Multiplier. There are two inputs to RM: a frequency f_1 , and a number Z in parallel binary form. The frequency of output F is controlled by the parallel number in input Z . The input and output relation of RM is expressed in equation (1), where P is the capacity of the RM:

$$F = f_i \frac{Z}{P} \quad (1)$$

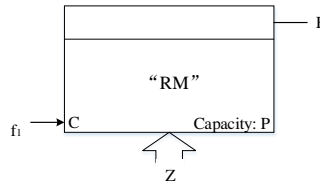


Fig. 62 Rate multiplier.

RM is used to convert a number in parallel representation Z to pulse rate representation F with a reference clock of f_1 shown in Fig. 63. The other main block is the reversible counter (RC), shown in Fig. 63. Pulses entering the "C+" input make it count up in the binary counter, and pulses entering the "C-" input make it count down. The total number indicated by the counter output "CNT" is the sum of pulses to input C+ minus the sum of pulses to input C-. (Number represented by pulse)

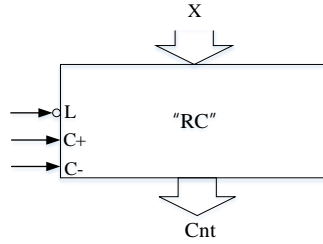


Fig. 63 Reversible counter.

6.2.1 Circuits to perform arithmetic operations on two pulse rate signal

Taha and Perkowski [91] proposed using pulse rate representation and a feedback loop system for arithmetic operations. As shown in Fig.64, these units can be connected to perform calculations. For example, when the output of the RC circuit, Z, is initially set to 0, the pulse signal coming from the lower RC circuit will also start at 0 and is connected to the terminal of the RM circuit. The upper two RM circuits, connected in series, perform a multiplication operation on the pulse rate of constants A and B. The result is then connected to the C+ terminal of the RC circuit. Since the pulse rate of A x B is greater than 0, the RC circuit will begin counting up, and Z will increase until

$$Z=A \times B \quad (2)$$

The input pulses to RC's C+ and C- terminals now occur to be the same, and Z stays unaltered. Hence, the system will reach an equilibrium where the two input frequencies to the RC are alike.

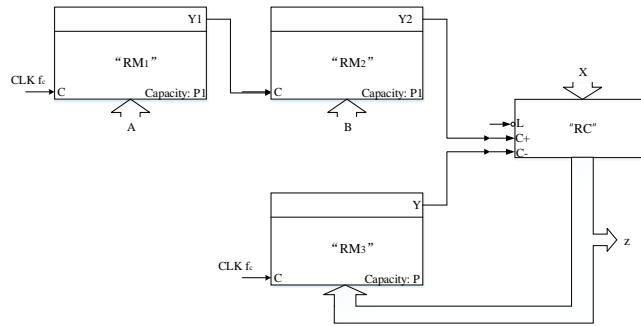


Fig. 64 Blocks connected for multiplication $Z = A * B$.

By connecting the feedback signal Z from RC to different RM s, which means that the unknown can be at any position in the equation (2). This circuit can change and be used to solve multiplication, division, square root, n th order root and any one unknown equation that has only multiplication and division operations. Fig. 65 shows the connections for solving square roots.

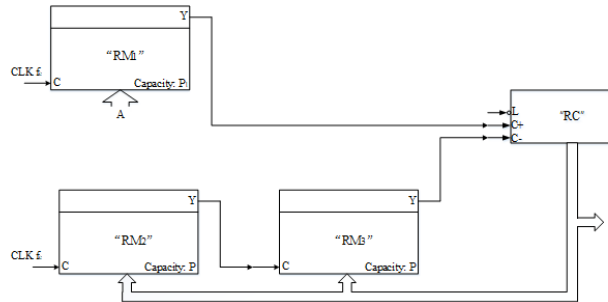


Fig. 65 Blocks connected for square root $Z = vA$.

6.2.2 Addition

The pulse rate system can be extended to solve sets of algebraic and differential equations by incorporating additional operations. By expanding the arithmetic system to include operations such as addition, multiplication, and integration over time, I can use the pulse rate system to solve equations.

Fig.66 illustrates a pulse rate adder that utilizes a multiplexer to alternate between selecting pulses from two input signals, A and B, at a specific control clock rate. As shown in Fig.67, the combination of signals A and B results in a new pulse rate signal (A+B) with a period and capacity double that of the input signals. The rate multiplier (RM) converts the input into a pulse rate before sending it to the pulse rate adder. The output of the pulse rate adder is then connected to one input of the RC circuit. Once the output of the RC circuit, Z, reaches a stable state, the value of Z is equal to the sum of A and B.

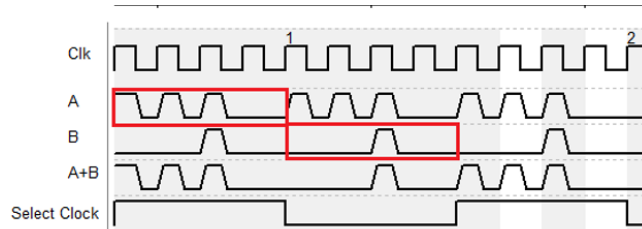


Fig. 66 addition of two pulses A and B.

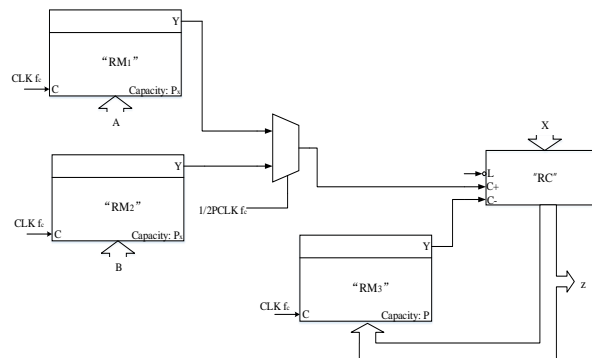


Fig. 67 Blocks connected for addition $A+B = Z$.

6.2.3 Pulse rate integrator.

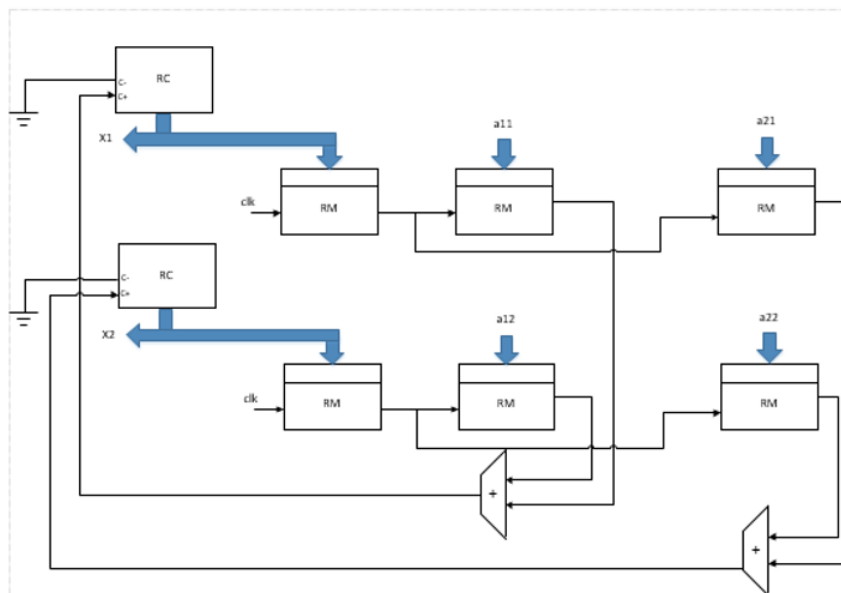
The counter is a special case of an accumulator, adding only a count of unity at each operation. Consider a counter with an input pulse rate signal (X)r, counting a total of N

input pulses in T seconds, where the n th pulse has a period t_n . The count displayed by the counter will be:

$$\frac{T \times N}{\sum_{n=1}^N t_n} = T \times Avg(X)_r = \int_0^T (X)_r dt$$

Where $Avg(X)_r$ is the average value of the signal $(X)_r$ over time T . Thus, a counter calculates the integral over time of the input pulse rate and converts it to a parallel coded signal. Fig.68 illustrates an example of a linear differential equation. The solution can be obtained through integration using the Reverse Counter and then multiplied by a Rate

- $x1 = a11 \int x1 dt + a12 \int x2 dt$
- $x2 = a21 \int x1 dt + a22 \int x2 dt$



Multiplier.

Fig. 68 Blocks connected for Linear Differential Equations.

6.2.4 System of nonlinear equations

I have further expanded the concept to achieve equilibrium between the inputs of two RC circuits to solve equations with two unknowns. Fig.69 illustrates a pulse rate system for solving the equations $x + y = B$ and $xy = A$, where A and B are known constants, and x and y are unknowns. When both RC circuits reach equilibrium, meaning the output of both RC circuits satisfies both equations, the unknowns x and y are determined.

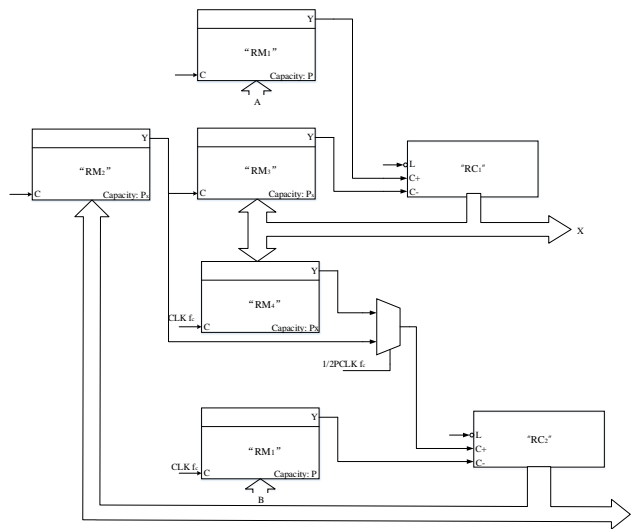


Fig. 69 the circuit to solve nonlinear system of two algebraic equations $x + y = B$, $x*y = A$.

6.3 Implementation using memristor imply gates

6.3.1 Implement RM with imply gates.

The design goal for my system, which utilizes memristors, is to maintain the functionality of the pulse-rate logic. The binary rate multiplier is a circuit that produces a programmable number of pulses within a specified window of cycles. To convert the rate multiplier to memristor logic, it is essential to comprehend the fundamental behavior of the rate multiplier. Fig. 70 illustrates the basic behavior logic of a 3-bit rate multiplier.

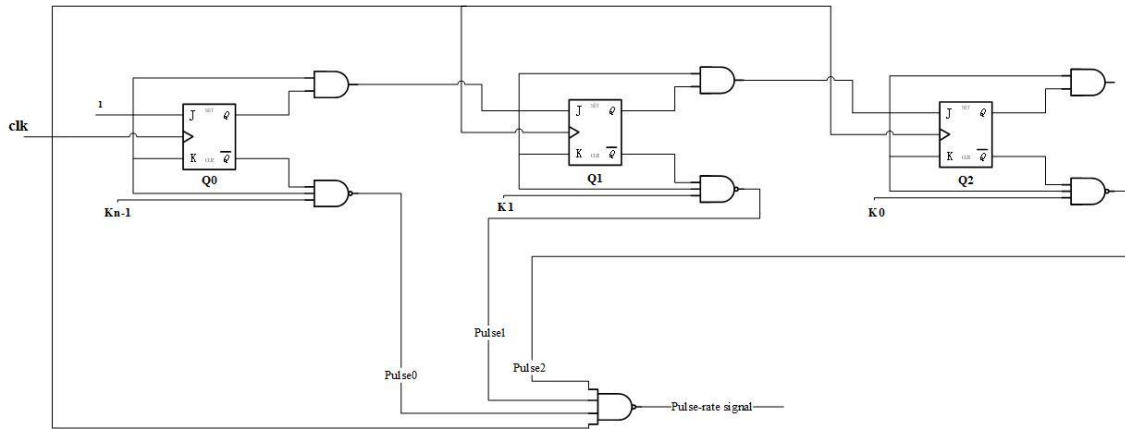


Fig. 70 the basic behavior of 3-bit Rate Multiplier.

In Fig.71, the circuit of the Rate Multiplier can be broken down into three components: the counter, the pulse generator, and the glitch-free part. The counter is a binary n-bit synchronous up counter implemented using sequential JK flip-flop logic. The pulse generator can be constructed using two levels of NAND gates. The first level generates a single-bit pulse, and the second level multiplies the single bits together. The glitch-free part is composed of a latch logic and an AND gate. It is used to eliminate glitches within the window of cycles. The first two parts generate the programmable number of pulses. Therefore, the focus of converting to a memristor circuit is on the counter and the pulse generator, as shown in Fig.71. A N-bit counter is constructed using N submodules, each composed of an AND gate and a JK flip-flop.

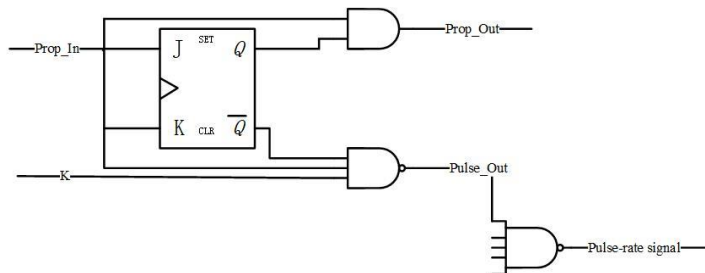


Fig. 71 The rate generator and one bit of submodule from the counter.

In the circuit of Rate Multiple shown in Fig. 71, the propagation input Prop_In signal is connected to the previous stage propagation out Prop_Out signal. The Pulse_Out signal generates the single bit pulses by gating the bit x of the counter and the corresponding reverse order of the programmed value K , $n-1-x$. For example, if the current order of the counter is Q_0 , then $Pulse_Out = \overline{Q_0 Prop_In K_{n-1}}$. The Pulse-rate signal is generated by applying NAND operation on all single-bit pulses together.

The JK Flip Flops in the counter operate in toggle mode that is $J = 1$, $K = 1$ or mode that is equivalent to a T Flip Flop. Therefore, the variables J and K in the JK Flip Flop always keep the same value, and these two variables can be regarded as one variable JK (Prop_In).

In the memristor system, there is no flip flop with a clock be needed because one of the basic functions of the memristor is keeping the value until the control voltage changes. So, when I convert, I ignore the clock signal in the JK Flip Flop. According to the function of JK Flip Flop, the relationship of the Prop_In and Q_n , the new output of the JK Flip Flop can be inferred. Assuming the output value kept in the memory is Q , the truth table of the JK Flip Flop is as in table 19.

Input	Current Memory	Next Memory	Description
JK (Prop_In)	Q	Q_n	
0	0	0	Memory no change
0	1	1	
1	0	1	Toggle (\bar{Q})

1	1	0	
---	---	---	--

Table 19 The Excitation table of the JK Flip Flop

As shown in the truth table, when $JK = 0, Q_n = Q$; when $JK = 1, Q_n = \bar{Q}$. The behavior function is the same as a multiplexer. So, I convert the JK Flip Flop to a multiplexer. Here is the transform diagram of JK Flip Flop shown in Fig.72.

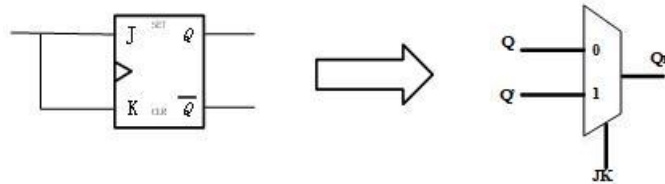


Fig. 72 The transform diagram of JK Flip Flop.

Therefore, the transform equation of the JK Flip Flop is:

$$Q_n = JK\bar{Q} + \bar{J}KQ = \overline{\bar{J}\bar{K} + Q} + \overline{JK + \bar{Q}}$$

The transform equation can be transformed to the imply-gate form:

$$Q_n = (Q \rightarrow JK) \rightarrow ((JK \rightarrow Q) \rightarrow "0")$$

The Prop_Out signal is generated from an AND gate. However, the point to be taken care of is that the inputs of the AND gate are Prop_In (JK) and output Q in the recent memory, not the next states of them. So, the transform circuit of the AND gate should be put in the front of the multiplexer.

The Prop_Out signal is represented by JKn (the next state of JK variable). Here is the transform equation of the AND gate:

$$JKn = JKQ = \overline{(\bar{J}\bar{K} + \bar{Q})}$$

The transform equation can be transformed to the imply-gate cascade form:

$$JKn = (Q \rightarrow (JK \rightarrow 0)) \rightarrow 0$$

The pulse generator consists of two NAND gates. The First NAND gate is gating the Prop_In signal and the current order of the counter Q_x and the reverse order of the programmed value K_{n-1-x} . Similarly, the counter bit Q_x and the Prop_In signal are the ones store in the current memory, not the next states of them.

The transform equation of the Pulse_Out is:

$$Pulse_Out = \overline{(JK \overline{Q_x} K_{n-1-x})} = \overline{JK} + Q_x + \overline{K_{n-1-x}}$$

The transform equation can be transformed to the imply-gate form:

$$Pulse_Out = K_{n-1-x} \rightarrow (JK \rightarrow Q_x)$$

The final Pulse-Rate signal is generated by gating all the single bit pulse signal together. For n-bit Rate Multiplier, there are n-single bit pulse signal produced. The transform equation of the final Pulse-Rate signal is

$$Pulse - Rate\ signal = \overline{pulse_0\ pulse_1\ \dots\ pulse_{n-1}} = \overline{pulse_0} + \overline{pulse_1} + \dots + \overline{pulse_{n-1}}$$

The transform equation can be transformed to the imply-gate form:

$$Pulse - Rate\ signal = pulse_{n-1} \rightarrow (\dots (pulse_1 \rightarrow (pulse_0 \rightarrow 0)) \dots)$$

From the entirety aspect, the unit behavior circuit can be converted to another behavior circuit without clock. This circuit is realized as in the Fig. 73.

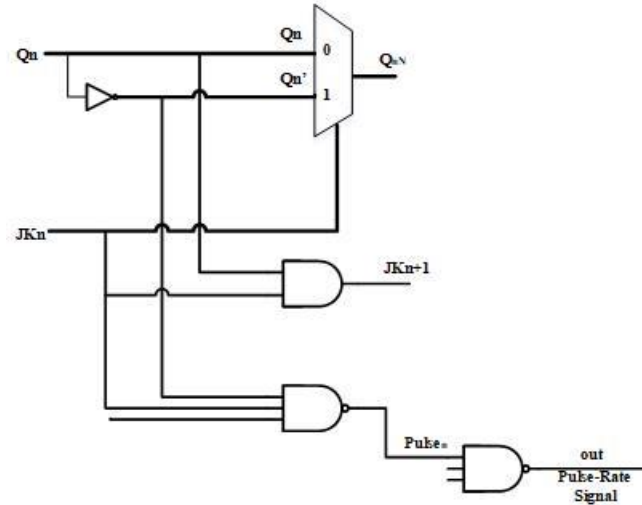


Fig. 73 the transform unit behavior circuit without clock.

From the transform unit behavior circuit, I can calculate the Q_n signal, JK_n signal and pulse signal at the same time. After generating the pulse signal, I can do a calculation pulse \rightarrow out one time. I use the notation with one square and one circle to represent the imply gate. The circle is used to invert the connected signal. For example, as shown in Fig.74, the output of the imply gate is $\bar{A} + B$.

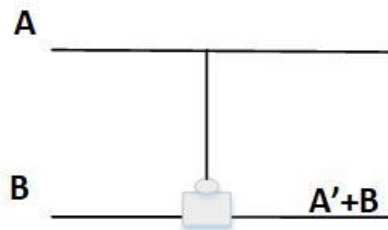


Fig. 74 An example for the imply gate representation.

I use one square with text '0' as the clear gate, which means clear the signal as 0. Here is an example.

Fig. 75 clear gate symbol.

Taking a 3-bit Rate Multiplier as an example, Fig.74 and Fig.75 are the memristor-level circuit of Rate Multiplier. Each figure is one unit circuit. The line JK is the Prop_In signal. The lines {Q2, Q1, Q0} are the counter signal. The lines {K2, K1, K0} are the Programmed number signal. The lines {PL2, PL1, PL0} are the produced single-bit pulses. The lines {W1, W2, W3} are inter-variables. The line out is the output of the Rate Multiplier, which corresponds to the produced Pulse-Rate signal.

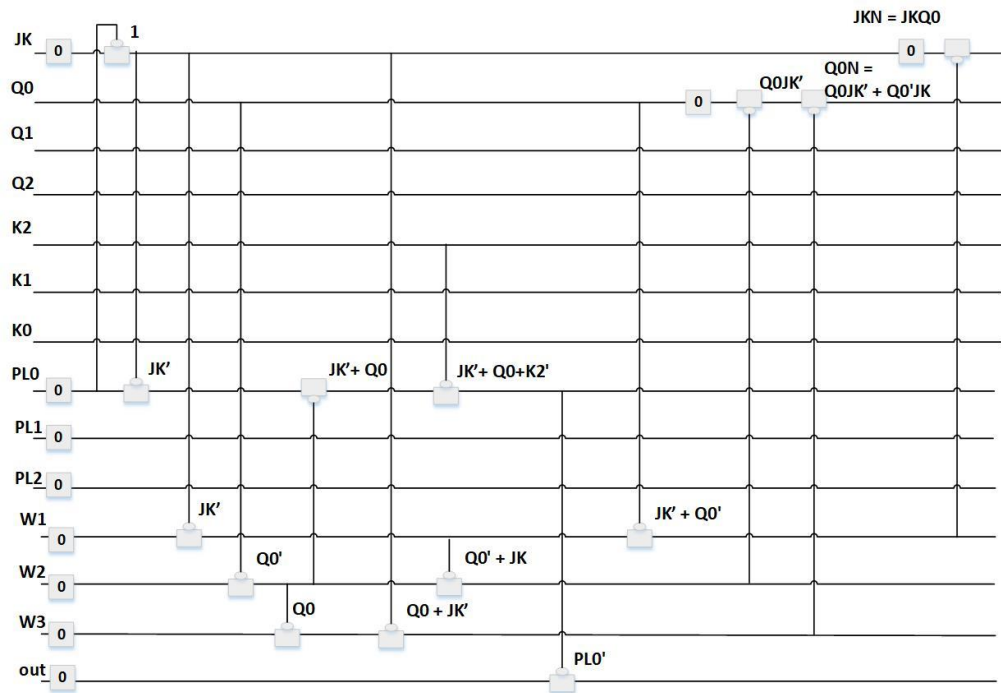


Fig. 76 The unit circuit of 3-bit Rate Multiplier for bit 0 position.

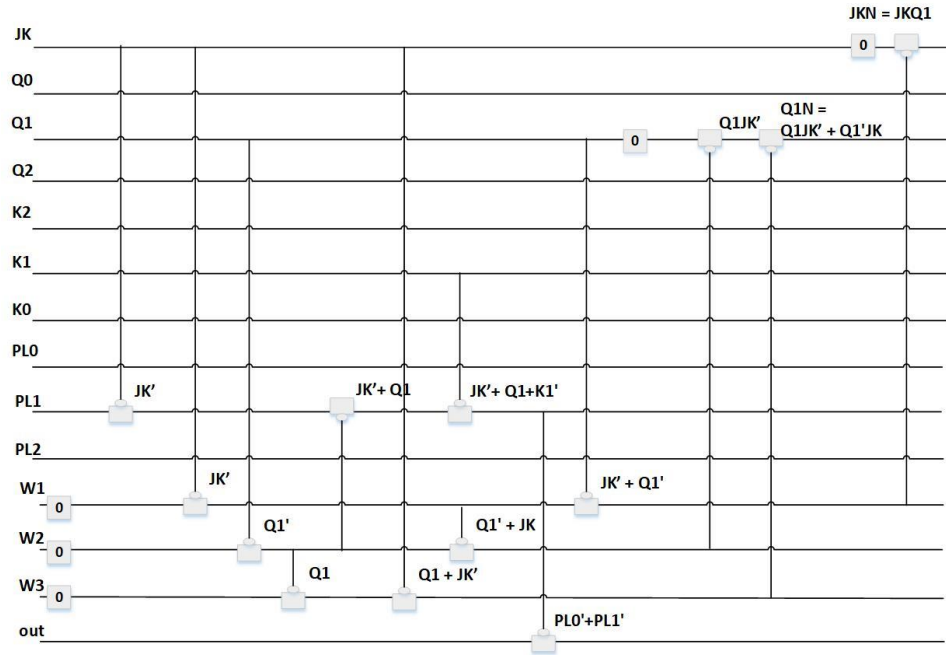


Fig. 77 The unit circuit of 3-bit Rate Multiplier for bit 1 position.

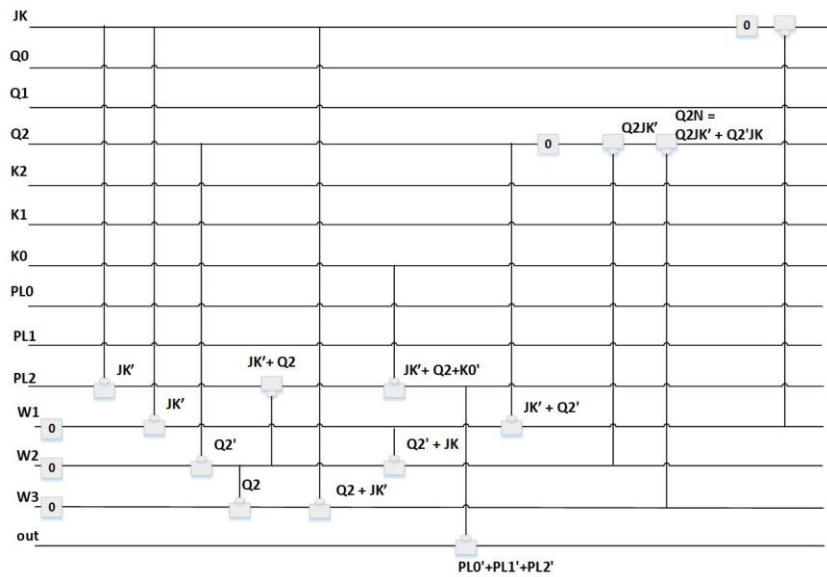


Fig. 78 The unit circuit of 3-bit Rate Multiplier for bit 2 position.

6.3.2 Implement RC with imply gates

To convert the Rate Counter to the memristor-level system, I must understand the basic functionality of Rate Counter. Rate Counter is a counter according to the two input signals. Here is the black block of the Rate Counter to display the basic functionality of the Rate Counter.

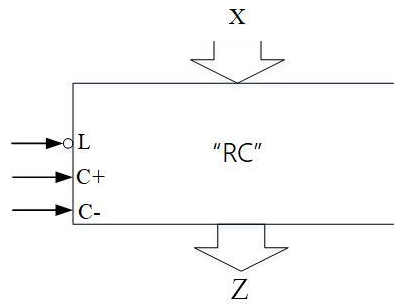


Fig. 79 the black block of the Rate Counter.

As Fig.79 shown, the inputs of the Rate Counter consist of two single-bit selected signals, $C+$, $C-$, and current count value X ; the output is the count-up or count-down value. The basic functionality can be described as if $C+ > C-$, then $Z = X+1$; if $C+ < C-$, $Z = X-1$; if $C+ = C-$, the output stabilizes, $Z = X$. Therefore, the truth table of the Rate Counter is as below shown:

Selected Signals		Output
$C+$	$C-$	Z
0	0	X
0	1	$X - 1$
1	0	$X + 1$

1	1	X
---	---	---

Table 20 the truth table of Rate Counter.

The transform starts from building the behavior circuit of Rate Counter. To realize the counter, I build the sequential JK Flip Flop. When the input of JK (Prop_In signal) is 1 or the value of gating the current counter and Prop_In signal, the logic is a binary synchronous up counter. If the input of JK is 1 or the value of gating the invert current counter and Prop_In signal. If the input of JK is 0, according to the basic function of JK Flip Flop, every bit of the counter keeps its value. The Fig.80 present the basic structure of the binary up counter and the binary down counter.

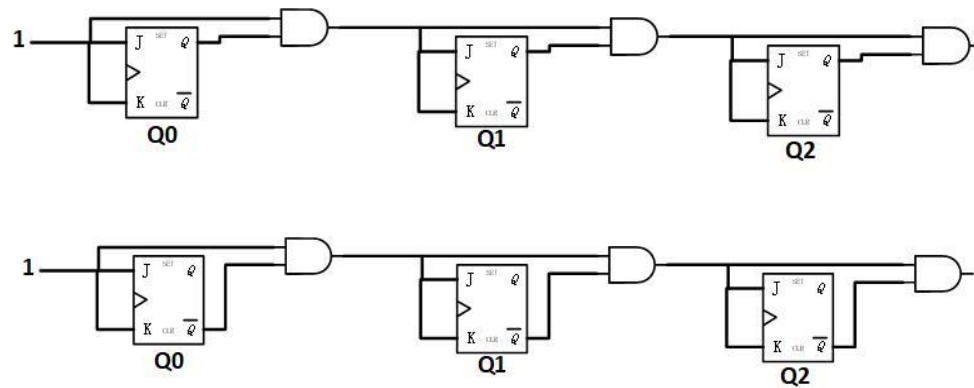


Fig. 80 the 3-bit binary down and up counter.

The Fig.81 presents the entire behavior of the Rate Counter. Except the least bit, the unit circuit can be divided into three parts: the up-down selector, the count-remained selector and the JK Flip Flop. The up-down selector is to determine the machine counts to up or down. The count-remained selector is to determine if the machine keep its value. The JK Flip Flop is the basic structure of the counter.

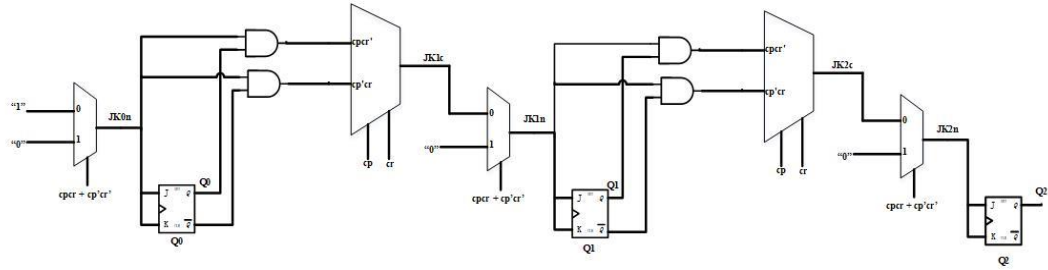


Fig. 81 the basic behavior circuit of the 3-bit Rate Counter.

For the least bit, both Prop_In signals of the binary up counter and the binary down counter are logic “1”. So, it is not necessary to build a up-down selector for the least bit. Except the least bit, if $C+ = 1, C- = 0$, choose the AND gate which gating the Prop_In signal JK and the current bit of the counter X; if $C+ = 0, C- = 1$, choose the AND gate which gating the Prop_In signal JK and the invert current bit of the counter \bar{X} . Set the output of the selector as the JK for the counter, JKc. The below Fig.82 is the unit circuit of the up-down selector.

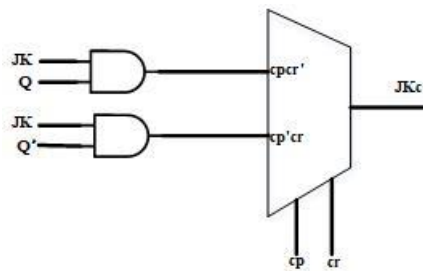


Fig. 82 the behavior circuit of the up-down selector.

Set $C+ = 1$ and $C- = 0$ as condition 0, the equation can be:

$$C0 = \overline{CR} CP = \overline{CR + \overline{CP}}$$

Set $C+ = 0$ and $C- = 1$ as condition 1, the equation can be:

$$C1 = CR \overline{CP} = \overline{\overline{CR} + CP}$$

The JK for the counter should be

$$JKc = C0 X JK + C1 \overline{X} JK = (\overline{\overline{C0} + \overline{X} + \overline{JK}}) + (\overline{\overline{C1} + X0 + \overline{JK0}})$$

The count-remained selector is to determine if the machine keep its value. If the signal C+ and signal C- have the same value, the selector choose logic “0” for the JK Flip Flop to keep the value in the memory. If the signal C+ and signal C- have the different value, the selector chooses the JK for the counter, JKc.

Set CP = CR = 0 or CP = CR = 1 as condition 2, the equation can be:

$$C2 = CRCP + \overline{CR} \overline{CP} = (\overline{\overline{CR} + \overline{CP}}) + (\overline{CR + CP})$$

The new JK should be

$$JKn = C2 \text{ gnd} + \overline{C2} JKc$$

The transform for the JK Flip Flop is similar with the transform of Rate Multiplier. The JK Flip Flop can be transformed to a multiplexer. The equation can be written as

$$Xn = JK\overline{X} + \overline{JK}X = (\overline{\overline{JK} + X}) + (\overline{JK + \overline{X}})$$

Finally, I can summary the transform process: First, use the input C+, C- to figure out every condition, C0, C1, C2. Then, calculate out the JKc. Next, figure out the new JK, JKn. Taking 3-bit Rate Counter as an example, the below figures are the memristor-level circuit of Rate Counter. The lines {X2, X1, X0} are the value store in the memory by JK Flip Flop. The line CP is the C+ signal. The line CR is the C- signal. The

lines {C2, C1, C0} are the three condition I need. The lines {JK2, JK1, JK0} are the Prop_In signals for each bit. The lines {W1, W2, W3, W4} are the inter variables.

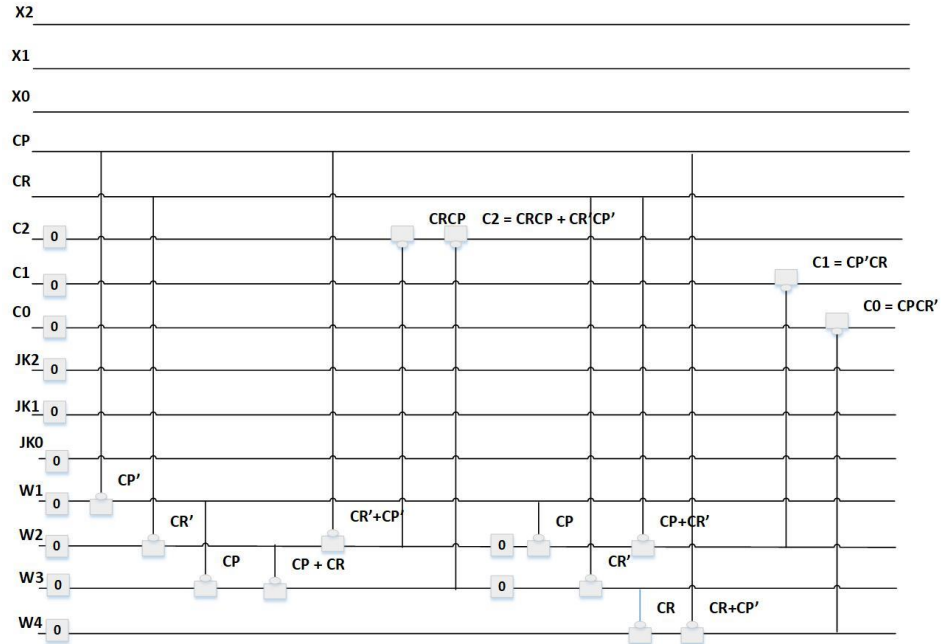


Fig. 83 Calculating for all selected condition C0, C1 and C2 part of the memristor-level circuit of 3-bit Rate Counter.

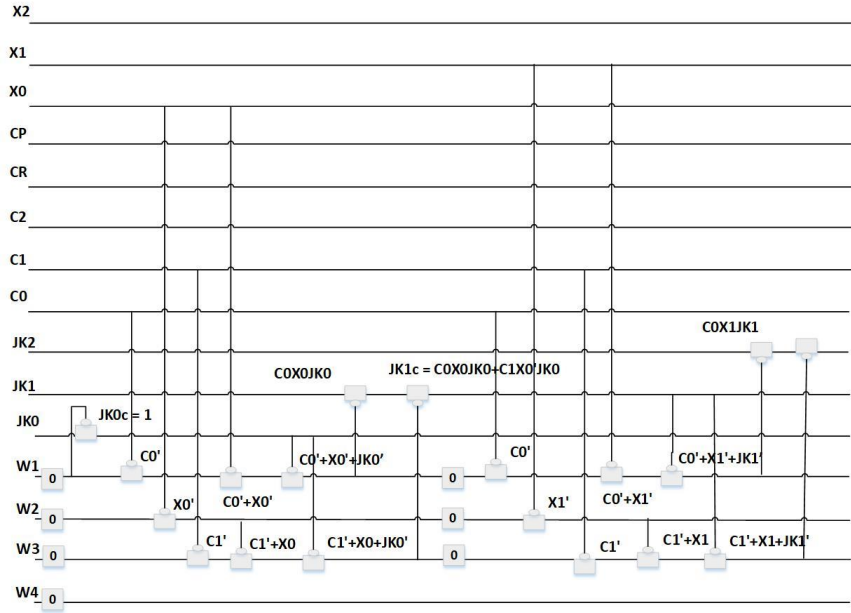


Fig. 84 Calculating the JK for each bit part of the memristor-level circuit of the 3-bit Rate Counter.

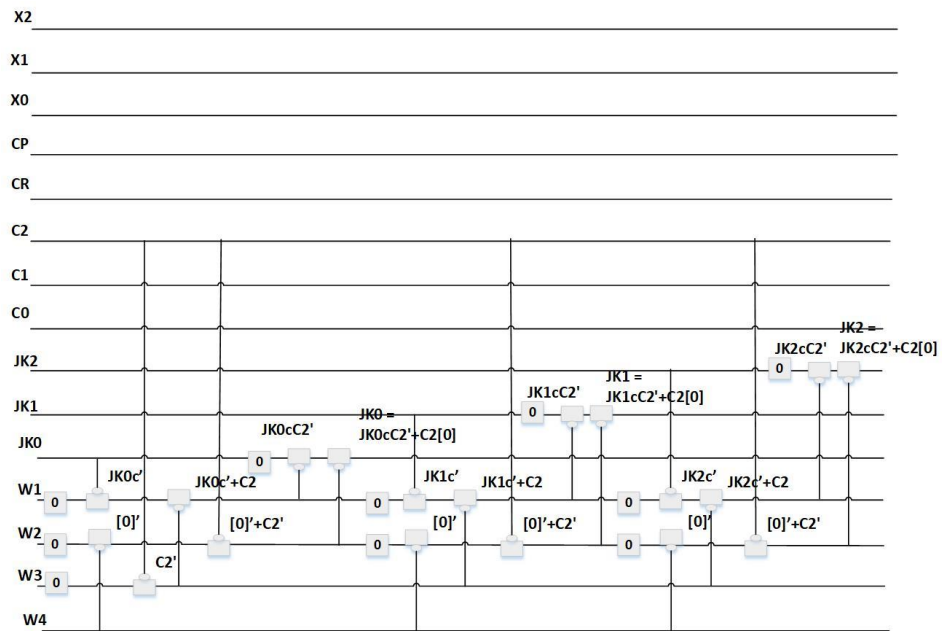


Fig. 85 Calculating the JK_n for each bit part of the memristor-level circuit of the 3-bit Rate Counter.

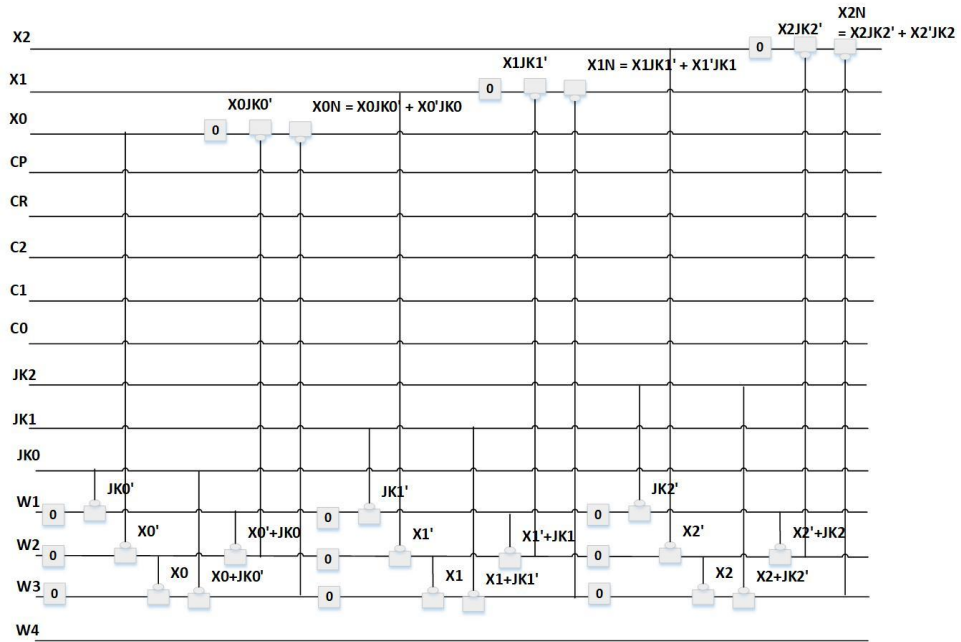


Fig. 86 Calculating the new X for each bit part of the memristor-level circuit of the 3-bit Rate Counter.

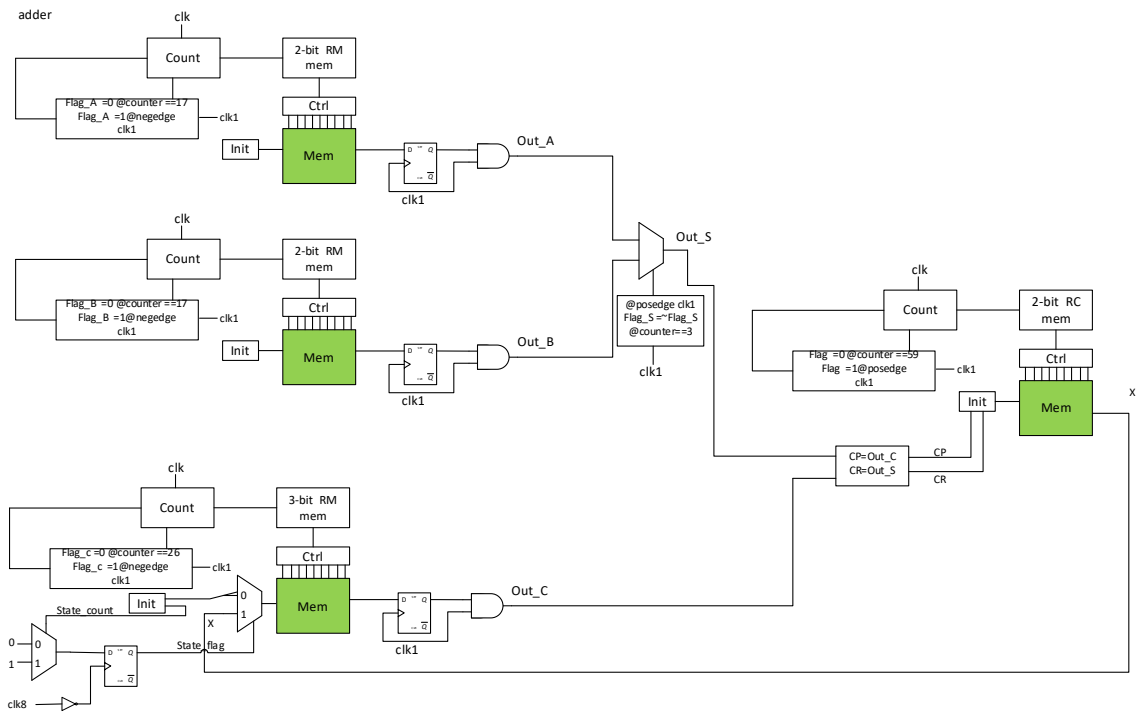


Fig. 87 Block diagram for hybrid memristor circuit for pulse rate adder.

6.4 Design of pulse rate system with memristors FPGA

A cascade of imply gates is used to implement a pulse rate computing system with a memristor-CMOS hybrid circuit, as shown in Fig.87. The green block represents the memristor crossbar, which performs the RM and RC operations. The white blocks are the CMOS circuit, which controls the timing and programming of the memristor FPGA.

Chapter 7 Conclusion

7.1 Related Work and Contributions

7.1.1 Quantum-based algorithm for generalized Ashenhurst-Curtis decomposition.

7.1.1.1 Related work

Early functional decomposition algorithms relied on decomposition charts [7], but these were later replaced by cube representations [8]. This change led to the creation of algorithms based on cube array operations [8]. However, as the size of decomposition charts and cube representations increased rapidly, these algorithms could no longer handle large practical data effectively. More recent solutions for functional decomposition include Binary Decision Diagrams (BDDs) [51] and formulating the problem as a Satisfiability (SAT) solving problem [52], but both have limitations. BDDs can become large when representing Boolean functions, particularly for functions with multiple values or incomplete specifications. On the other hand, the SAT approach works well for large functions with up to 300 input variables but only applicable to completely specified functions which makes it not usable for machine learning. In addition, it cannot handle multi-valued functions.

7.1.1.2 My contribution

To surpass the limitations of BDD and SAT methods in handling large functions with multiple values and incomplete specified functions, a new quantum algorithm for generalized Ashenhurst-Curtis decomposition is proposed. This algorithm leverages the exponential speedup provided by Grover's search algorithm to solve larger functions

effectively. This approach guarantees an exact minimum solution with sufficient number of qubits on a quantum computer. The number of qubits used is proportional to the size of the input function. The decomposition technique that is proposed in this dissertation is versatile, capable of handling both fully and partially specified functions, as well as functions with single and multiple outputs.

A systematic methodology is presented for the design and construction of quantum oracles for finding the minimum-cost decomposition, using partitions to represent the projective minterms and function output. It is for the first time that partition calculus is considered for constructing quantum oracle for Grover's Algorithm. This represents a new direction in the field of quantum oracle design and Quantum Machine Learning.

7.1.2 A new quantum algorithm for vertex graph coloring based on domination to solve the column multiplicity problem.

7.1.2.1 Related work

The column multiplicity in a chosen bound set is vital to the success of functional decomposition. A significant portion of the program's run time is dedicated to column minimization, making it imperative to find a solution that is both fast and effective. Graph coloring for functional decomposition was first introduced by Muzio and Wesselkamper [92] and Perkowski [93], with recent advancements leading to more graph coloring-based approaches [111]. Perkowski introduced a domination-based graph coloring heuristic and demonstrated that exact graph coloring is not necessary for high-quality functional decomposition, particularly in data mining applications [94]. He showed that a new heuristic graph coloring algorithm can yield results of comparable

quality to those produced by an exact graph coloring algorithm in functional decomposition.

7.1.2.2 My contribution

My new quantum graph coloring algorithm can be used for Ashenhurst-Curtis decomposition and many problems in Electronic Design Automation and robotics. Inspired by the domination-based Graph coloring heuristic [94], a hybrid quantum algorithm for solving the Column Multiplicity Problem is developed. The quantum oracle for the domination-based graph coloring was designed using functional quantum circuit blocks that were developed for the Ashenhurst-Curtis decomposition. The hybrid algorithm was implemented using Quipper, a Haskell-based language that supports both classical and quantum programming. The proposed approach provides a promising example for hybrid quantum algorithms in the future.

7.1.3 Implementation of memristive finite state machines

7.1.3.1 Related work

Several studies have proposed frameworks and architectures for system-level design with memristors, as seen in the works of Rahman [74], Tissari [75], and Xie [84]. Rahman and Tissari propose configurable logic architectures akin to conventional FPGAs, while Xie's architecture features a fixed logic unit similar to an ASIC. However, the implementation of memristive finite state machines (FSMs) has received limited attention, as highlighted by a few works such as Ferrandino [112]. Conventional state assignment (SA) techniques, such as KISS [98] and NOVA [99], which use symbolic minimization, are not suitable for memristive FSMs. Heuristic-based SA techniques for multi-level circuits,

such as MUSTANG [100], JEDI [101], and MUSE [102], which aim to maximize the factoring of expressions to reduce the number of literals in the resulting circuit, may not minimize the circuit cost for memristive FSMs.

7.1.3.2 My contribution

A new concept to the design of memristor-based field-programmable gate arrays (FPGAs) has been proposed [74], that I am also the co-author of. Here this approach is extended to memristive finite state machines (FSM). To optimize the cost for the memristive FSM, a state assignment (SA) algorithm was developed based on the selected memristor architecture [74] to minimize the cost of the memristor circuit in various memristor realization structures. This is a completely new approach to the design of memristive state machines.

7.1.4 Development of Memristor-based Components for a Pulse Rate System

7.1.4.1 Related work

The pulse rate system [92] is a computational approach that leverages the frequency of a signal's occurrence, rather than its electrical characteristics, to encode and process information. Taha and Perkowski [91] developed a dynamic solver of algebra and differential equations that utilized pulse rate representation and a feedback loop system based on a classical sequential circuit. The pulse rate computer can perform two operand multiplication, division, and exponential computation.

7.1.4.2 My contribution

I extended the pulse rate design methodology to solve the systems of two arbitrary algebraic and differential equations. A few more operations, such as addition and

integration over time, are added to the previous methodology. I illustrated and simulated this methodology with the example of a new pulse rate computer that can solve systems of simple algebraic and differential equations. Subsequently, the basic components of the pulse rate computer are converted to a cascade of imply-gates, which is realized in memristive FPGAs. My approach represents the first attempt to implement a pulse rate system using memristor circuits, offering reduced circuit area and lower power consumption, albeit at the cost of reduced speed. These implementations have the potential to be ideal solutions for embedded systems that operate in low-power environments and have low computing performance requirements.

7.2 Conclusion

In this research, I created two general approaches to synthesize combinational and sequential circuits in two new technologies: quantum computing and memristors. In the first area, I developed two quantum algorithms based on Grover's algorithm. Utilizing a bottom-up approach, I designed various quantum oracles to tackle functional decomposition problems. My oracle design methodology was applied to the decomposition and graph coloring problems, revealing its potential applications and impact on various optimization problems. I contend that the most significant contribution of this study lies in the emerging field of Quantum Machine Learning, as my method differs from previous quantum learning algorithms. A notable distinction between the applications of logic synthesis and machine learning is that the functions employed in the latter tend to have a larger number of input variables, a limited number of positive or negative samples, and a considerable number of "don't cares." This feature makes my

method more suitable for machine learning than most previous approaches based on decomposition. Furthermore, functions in Machine Learning often have multi-valued inputs, making my proposed decomposition method particularly effective for handling strongly unspecified functions. My methods can be easily extended to incompletely specific multi-value functions. This opens the possibility for the development of a new area of Quantum Machine Learning, specifically functional decomposition-based quantum machine learning. In the second application area, I developed two original approaches to systematically design memristive circuits and specifically memristive finite state machines and sequential pulse rate systems. My algorithm to design and encode memristive automata give better result than several encoding algorithms because it takes into account of optimal memristive circuit synthesis. In another contribution, I developed a memristor-based pulse rate computing system utilizing a set of pulse rate sequential circuit components. The system represents numbers with sequences of pulses, simplifying the circuit required to perform algebraic operations such as multiplication and differential equations, resulting in a reduction of power consumption and circuit size.

References

- [1] Z. Kohavi, *Switching and finite automata theory*. New York: McGraw-Hill, 1978.
- [2] B. Zupan, M. Bohanec, I. Bratko, J. Demsar, *Machine Learning by Function Decomposition*, *Proceeding Machine Learning and Its Applications, Advanced Lectures* pp. 71-101, 2001.
- [3] C. M. Files and M. A. Perkowski, "Multi-valued functional decomposition as a machine learning method," *Proceedings. 1998 28th IEEE International Symposium on Multiple-Valued Logic*, Fukuoka, 1998.
- [4] C. Delobel and R. G. Casey, "Decomposition of a Data Base and the Theory of Boolean Switching Functions," *IBM Journal of Research and Development*, vol. 17, no. 5, pp. 374-386, Sept. 1973.
- [5] J. B. H. Li, "A Function Decomposition Approach for Data Privacy of Controlling Smart Grids," *2016 IEEE Global Communications Conference (GLOBECOM)*, Washington, DC, 2016.
- [6] R. L. Ashenurst, "The Decomposition of Switching Functions," *Proceedings of an International Symposium on the Theory of Switching*, April 2-5, 1957.
- [7] H. A. Curtis, *A New Approach to the Design of Switching Circuit*. Princeton, New Jersey: D. Van Nostrand, 1962.
- [8] J. P. Roth, R.M. Karp: *Minimization over Boolean graphs*, *IBM Journal*, April 1962, pp. 227-238.
- [9] M. A. Perkowski and S. Grygiel, "A survey of literature on function decomposition," *Portland State University, Tech. Rep. Ver. IV*, 1995.

- [10] L. Jozwiak, F. Volf. An efficient method for decomposition of multiple-output Boolean functions and assigned sequential machines, Proceedings of the European Conference on Design Automation, 114-122, 1992.
- [11] T. Lee, T. Ye. A relational approach to functional decomposition of logic circuits, ACM Transactions on Database Systems, vol.36 (2), article No. 13, 2011.
- [12] C. Scholl. Functional decomposition with application to FPGA synthesis. Kluwer Academic Publishers, 2010.
- [13] Y, Lai, M. Pedram and S. B. K. Vrudhula, "BDD Based Decomposition of Logic Functions with Application to FPGA Synthesis," 30th ACM/IEEE Design Automation Conference, 1993, pp. 642-647.
- [14] S. T. Afshord , Y. Pottosin , B. Arasteh. An input variable partitioning algorithm for functional decomposition of a system of Boolean functions based on the tabular method, Discrete Applied Mathematics, vol. 85(20): 208–219, 2015.
- [15] J. L. Hein. Discrete Structures, Logic, and Computability. Jones and Bartlett Publishers, Inc., USA, 1995.
- [16] T. Luba, H. Selvaraj. A General Approach to Boolean Function Decomposition and its Application in FPGA Based Synthesis, VLSI Design, Vol. 3 (3-4):289-300, 1995.
- [17] J. Hartmanis, R. E. Stearns. Algebraic structure theory of sequential machines[M], Prentice-Hall, 1966.
- [18] P. Erdős, R. Rado. A Partition Calculus in Set Theory. In: Gessel I., Rota GC. (eds) Classic Papers in Combinatorics. Modern Birkhäuser Classics. Birkhäuser Boston, 2009.

- [19] D Ellerman, An Introduction to Partition Logic. Logic Journal of the IGPL, 2013.
- [20] L. K. Grover, "A fast quantum mechanical algorithm for database search," Proceedings of the 28th Annual ACM Symposium on Theory of Computing 1996, pp. 212-219
- [21] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," Phys. Rev. Lett., 1997, vol.79(2): pp.325-328.
- [22] M. A. Nielsen and I. L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2000.
- [23] H. Frédéric, J. Hamza, N. Ismaël. Grover's algorithm and the secant varieties, Quantum Information Processing, Vol.15(11): 4391-4413, 2016.
- [24] C. Kaushik, M. Subhamoy. Application of Grover's algorithm to check non-resiliency of a Boolean function, Cryptography and Communications, Vol.8(3): 401-413, 2016.
- [25] P. Kaye. Reversible addition circuit using one ancillary bit with application to quantum computing, vol.2: quant-ph/ 0408173, 2004.
- [26] P. S. Phaneendra, C. Vudadha, V. Sreehari and M. B. Srinivas, "An Optimized Design of Reversible Quantum Comparator," 2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems, Mumbai, 2014, pp. 557-562.
- [27] A. Sarker, M. Shamiul Amin, A. Bose and N. Islam, "An optimized design of binary comparator circuit in quantum computing," 2014 International Conference on Informatics, Electronics & Vision (ICIEV), Dhaka, 2014, pp. 1-5.

- [28] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger and B. Valiron. Quipper: A Scalable Quantum Programming Language, Proceedings of the 34th ACM Conference on Programming Language Design and Implementation (PLDI), pp: 333-342, 2013.
- [29] S. Siddiqui, M. J. Islam, O. Shehab. Five Quantum Algorithms Using Quipper, arXiv:1406.4481v2[quant-ph], 2014.
- [30] The Quipper System, June 2013. URL <http://www.mathstat.dal.ca/~selinger/quipper/doc/>.
- [31] S. Yamashita and M. Nakanishi, "A practical framework to utilize quantum search," 2007 IEEE Congress on Evolutionary Computation, Singapore, 2007, pp. 4086-4093.
- [32] R.V.Meter and C. Horsman, A blueprint for building a quantum computer. Commun. ACM 56, 10 (Oct. 2013),84–93.
- [33] E. Knill, Conventions for quantum pseudocode (1996), Technical Report LAUR-96-2724, Los Alamos National Laboratory.
- [34] H. Selvaraj, T. Luba, M. Nowicka, B. Bignall.: Multiple-Valued Decomposition and its Applications in Data Compression and Technology Mapping, International Conference on Computational Intelligence and Multimedia Applications, Australia, 1997.
- [35] P. Sapiecha, H. Selvaraj, M. Pleban. Decomposition of Boolean Relations and Functions in Logic Synthesis and Data Analysis. In: Ziarko W., Yao Y. (eds) Rough Sets and Current Trends in Computing. RSCTC 2000, 2001.

- [36] A. Bernasconi, R. K. Brayton, V. Ciriani, G. Trucco and T. Villa, "Bi-Decomposition Using Boolean Relations," 2015 Euromicro Conference on Digital System Design, Funchal, 2015, pp. 72-78.
- [37] G. Yang, F. Xie, X. Song, M. Perkowski. Universality of two-qudit ternary reversible gates. *Journal of Physics A Mathematical and General*, The Institute of Physics, 39(2006), 7763-7773, 2006.
- [38] X. Song, G. Yang, M. Perkowski. Algebraic characteristics of reversible gates. *Theory of Computing Systems (Mathematical Systems Theory)*, Springer-Verlag, 39(2), 311-319, 2006.
- [39] G. Yang, X. Song, M. Perkowski and J. Wu. Realizing ternary quantum switching networks without ancilla bits. *Journal of Physics A Mathematical and General*, The Institute of Physics, 38(2005), 9689-9697, 2005.
- [40] P. McGeer, J. Sanghavi, R. Brayton and A. S. Vincentelli, "ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions," 30th ACM/IEEE Design Automation Conference, Dallas, TX, USA, 1993, pp. 618-624.
- [41] N. Cerf, L. Grover, C. Williams, "Nested quantum search and N P complete problems". Jun 1998. 18 pp. *Phys.Rev. A*61, 2000.
- [42] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46:493, 1998.
- [43] G. Chen, S. Fulling, and M. Scully. Grover's algorithm for multiobject search in quantum computing. arXiv e-Print [quant-ph/9909040](https://arxiv.org/abs/quant-ph/9909040), 1999.

- [44] A. Younes, "Strength and weakness in grover's quantum search algorithm ", arXiv:quant-ph/0811.4481v1, November 2008.
- [45] H. Selvaraj, H. Niewiadomski, P. Buciak, M. Pleban, P. Sapiecha, T. Luba, V. Muthukumar, Implementation of Large Neural Networks using Decomposition, 249-255. Las Vegas, NV: University of Nevada, Las Vegas, 2002.
- [46] P. Buciak, T. Luba, H. Niewiadomski, M. Pleban, Sapiecha, P. H. Selvaraj, Decomposition and Argument Reduction of Neural Networks. In: IEEE Sixth International Conference on Neural Networks and Soft Computing (ICNNSC'02), Zakopane, Poland, June, 2002.
- [47] C. M. Files and M. A. Perkowski, "Multi-valued functional decomposition as a machine learning method," Proceedings. 1998 28th IEEE International Symposium on Multiple- Valued Logic (Cat. No.98CB36138), Fukuoka, Japan, 1998, pp. 173-178.
- [48] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, Quantum Machine Learning, Nature 549, 195-202 (2017); arXiv:1611.09347.
- [49] E. P. DeBenedictis, "A Future with Quantum Machine Learning," in Computer, vol. 51, no. 2, pp. 68-71, February 2018.
- [50] S.Gupta, S. Mohanta, M. Chakraborty and S. Ghosh, "Quantum machine learning- using quantum computation in artificial intelligence and deep neural networks: Quantum computation and machine learning in artificial intelligence," 2017 8th Annual Industrial Automation and Electromechanical Engineering Conference (IEMECON), Bangkok, 2017, pp. 268-274.

- [51] Y.T.Lai, M. Pedram and S.B.K. Vrudhula, "BDD based decomposition of logic functions with applications to FPGA synthesis," Proc. 30th ACM/ IEEE DAC, June 1993, pp. 642-7.
- [52] H. P. Lin, J.-H. R. Jiang, and R.-R. Lee. To SAT or not to SAT: Ashenhurst decomposition in a large scale. In Proc. Int'l Conf. Computer-Aided Design, pp. 32-37, 2008.
- [53] V. Bertacco, M. Damiani, The disjunctive decomposition of logic functions, in: ICCAD, International Conference on Computer-Aided Design, 1997, pp. 78–82.
- [54] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *J. Symbolic Logic*, 22(3):250-268, 1957.
- [55] C. C. Lee, J.-H. R. Jiang, C.-Y. Huang, and A. Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In Proc. ICCAD, pp.227-233, 2007.
- [56] T. Sasao, FPGA design by generalized functional decomposition, pp. 233–258. Kluwer Academic Publishers, 1993.
- [57] S.-C. Chang, M. Marek-Sadowska, and T. Hwang, "Technology mapping for TLU FPGA's based on decomposition of binary decision diagrams," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 1226–1235, 1996.
- [58] H. Sawada, S. Yamashita, and A. Nagoya, "Restructuring logic representations with easily detectable simple disjunctive decompositions," in *Proceedings of Design Automation Conference*, pp. 755–759, IEEE, 1998.

- [59] B. Mohammad, D. Homouz, O. A. Rayahi, H. Elgabra and A. S. A. Hosani, "Hybrid Memristor-CMOS memory cell: Modeling and design," *ICM 2011 Proceeding*, Hammamet, 2011, pp. 1-6.
- [60] C. E. Merkel, N. Nagpal, S. Mandalapu and D. Kudithipudi, "Reconfigurable N-level memristor memory design," *The 2011 International Joint Conference on Neural Networks*, San Jose, CA, 2011, pp. 3042-3048.
- [61] A. Emara, M. Ghoneima and M. El-Dessouky, "Differential 1T2M memristor memory cell for single/multi-bit RRAM modules," *2014 6th Computer Science and Electronic Engineering Conference (CEEC)*, Colchester, 2014, pp. 69-72.
- [62] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams, "Memristive switches enable 'stateful' logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.
- [63] P. Kuekes, "Material implication: Digital logic with memristors," presented at the Memristor Memristive Systems Symp., Berkeley, CA, USA, Nov. 2008, vol. 21
- [64] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-based material implication (IMPLY) logic: Design principles and methodologies," *IEEE Trans. Very Large Scale IntegrSyst.*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.
- [65] H. Thimbleby, "Modes, WYSIWYG and the von Neumann bottleneck," *IEE Colloquium on Formal Methods and Human-Computer Interaction: II*, London, 1988, pp.

- [66] A. Chakraborty, A. Dhara and H. Rahaman, "Design of memristor-based up-down counter using material implication logic," 2016 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Jaipur, 2016, pp. 269-274.
- [67] M. Teimoory, A. Amirsoleimani, A. Ahmadi, S. Alirezaee, S. Salimpour and M. Ahmadi, "Memristor-based linear feedback shift register based on material implication logic," 2015 *European Conference on Circuit Theory and Design (ECCTD)*, Trondheim, 2015, pp. 1-4.
- [68] D. Mahajan, M. Musaddiq and E. E. Swartzlander, "Memristor based adders," 2014 *48th Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2014, pp. 1256-1260.
- [69] A. H. Shaloot and A. H. Madian, "Memristor based carry lookahead adder architectures," 2012 *IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Boise, ID, 2012, pp. 298-301.
- [70] K. Bickerstaff and E. E. Swartzlander, "Memristor-based arithmetic," 2010 *Conference Record of the Forty Fourth Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2010, pp. 1173-1177.
- [71] Y. Zhang, Y. Shen, X. Wang and Y. Guo, "A Novel Design for a Memristor-Based or Gate," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 8, pp. 781-785, Aug. 2015.

- [72] A. Zhanbossinov, K. Smagulova and A. P. James, "CMOS-memristor dendrite threshold circuits," *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, Jeju, South Korea, 2016, pp. 131-134.
- [73] L. Guckert and E. E. Swartzlander, "MAD Gates—Memristor Logic Design Using Driver Circuitry," in *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 64, no. 2, pp. 171-175, Feb. 2017.
- [74] K. C. Rahman, D. Hammerstrom, Y. Li, H. Castagnaro and M. A. Perkowski, "Methodology and Design of a Massively Parallel Memristive Stateful IMPLY Logic-Based Reconfigurable Architecture," in *IEEE Transactions on Nanotechnology*, vol. 15, no. 4, pp. 675-686, July 2016.
- [75] J. Tissari, E. Lehtonen, M. Laiho, L. Koskinen and J. Poikonen, "A cellular architecture for memristive stateful logic," *2014 14th International Workshop on Cellular Nanoscale Networks and their Applications (CNNA)*, Notre Dame, IN, 2014, pp. 1-2.
- [76] L. Chua, "Memristor-The missing circuit element," in *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507-519, Sep 1971.
- [77] D. B. Strukov, G. S. Snider, D. R. Stewart and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.
- [78] R. Williams, "How we found the missing memristor," *IEEE Spectrum*, vol. 45, no. 12, pp. 28–35, Dec. 2008.

- [79] A. Raghuvanshi and M. Perkowski, "Logic synthesis and a generalized notation for memristor-realized material implication gates," *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, San Jose, CA, 2014, pp. 470-477.
- [80] T. Bengtsson and A. Martinelli, "A BDD-based fast heuristic algorithm for disjoint decomposition," in *Proceedings of Asia and South Pacific Design Automation Conference, ASP-DAC03*, (Kitakyushu, Japan), January 2003.
- [81] V. M. Schafer, C. J. Ballance, K. Thirumalai, L. J. Stephenson, T. G. Ballance, A. M. Steane, and D. M. Lucas. Fast quantum logic gates with trapped-ion qubits. *Nature*, 2018.
- [82] A. Saha, A. Chongder, S.B Mandal, A.Chakrabarti, (2015). Synthesis of vertex coloring problem using grover's algorithm. In *IEEE International Symposium on Nanoelectronic and Information Systems*.
- [83] C. Calude and E. Calude. *The Road to Quantum Computational Supremacy*. Tech. rep. Dec. 2017. arXiv:1712.01356.
- [84] X. Lei, N. Hoang Anh Du, M. Taouil, S. Hamdioui and K. Bertels, "Fast boolean logic mapped on memristor crossbar," *2015 33rd IEEE International Conference on Computer Design (ICCD)*, New York, NY, 2015
- [85] A. Mehrotra and M. Trick. A column generation approach for graph coloring. *INFORMS Journal on Computing*, 8(4):344–354, 1996.
- [86] M. Adegbindin, Alain Hertz, and M. Bellaïche, "A New Efficient RLF-Like Algorithm for the Vertex Coloring Problem," *Yugoslav Journal of Operations Research*, vol. 26, no. 4, pp. 441-456, 2016.

- [87] P. San Segundo, "A New DSATUR-Based Algorithm for Exact Vertex Coloring," *Computers & Operations Research*, vol. 39, no. 7, pp. 1724-1733, 2012.
- [88] D. Johnson, C. Aragon. "Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning." *Operations research* 39.3 (1991): 378-406.
- [89] A, Muayad, P, Long, and M Perkowski. "Memristor-based volistor gates compute logic with low power consumption." *BioNanoScience* 6.3 (2016): 214-234.
- [90] A. Raghuvanshi and M. Perkowski, "Logic synthesis and a generalized notation for memristor-realized material implication gates," 2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, 2014, pp. 470-477.
- [91] M. M. A. Taha and M. Perkowski, "Realization of Arithmetic Operators Based on Stochastic Number Frequency Signal Representation," 2018 IEEE 48th International Symposium on Multiple-Valued Logic (ISMVL), Linz, Austria, 2018, pp. 215-220.
- [92] J. P. Hayes, "Introduction to stochastic computing and its challenges," 2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 2015, pp. 1-3, doi: 10.1145/2744769.2747932.
- [93] Y. Liu, S. Liu, Y. Wang, F. Lombardi and J. Han, "A Survey of Stochastic Computing Neural Networks for Machine Learning Applications," in *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 7, pp. 2809-2824, July 2021.

- [94] M. Perkowski, R. Malvi & L. Jozwiak. Exact Graph Coloring for Functional Decomposition: Do we Need it?, 1998.
- [95] L. Chua, "Memristor-The missing circuit element," in IEEE Transactions on Circuit Theory, vol. 18, no. 5, pp. 507-519, Sep 1971.
- [96] B. Mohammad, D. Homouz, O. A. Rayahi, H. Elgabra and A. S. A. Hosani, "Hybrid Memristor-CMOS memory cell: Modeling and design," ICM 2011 Proceeding, Hammamet, 2011, pp. 1-6.
- [97] C. E. Merkel, N. Nagpal, S. Mandalapu and D. Kudithipudi, "Reconfigurable N-level memristor memory design," The 2011 International Joint Conference on Neural Networks, San Jose, CA, 2011, pp. 3042-3048.
- [98] J. Hartmanis, "On the State Assignment Problem for Sequential Machines, I", IRE Trans. Electr. Comp., Vol. EC-10, p. 157-165, June 1961.
- [99] T. Villa and A. Sangiovanni-Vincentelli, "NOVA: state assignment of finite state machines for optimal two-level logic implementation," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 9, no. 9, pp. 905-924, Sept. 1990.
- [100] S. Devadas, Hi-Keung Ma, A. R. Newton and A. Sangiovanni-Vincentelli, "MUSTANG: state assignment of finite state machines targeting multilevel logic implementations," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 7, no. 12, pp. 1290-1300, Dec. 1988.

- [101]B. N. V. M. Gupta, H. Narayanan and M. P. Desai, "A state assignment scheme targeting performance and area," Proceedings Twelfth International Conference on VLSI Design. (Cat. No.PR00013), Goa, India, 1999, pp. 378-383.
- [102]X. Du, G. Hachtel, B. Lin and A. R. Newton, "MUSE: a multilevel symbolic encoding algorithm for state assignment," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 10, no. 1, pp. 28-38, Jan. 1991.
- [103]T. Kam, T. Villa, R.K Brayton & Sangiovanni-Vincentelli, A. L. Synthesis of finite state machines: functional optimization. Springer Science & Business Media, 2013.
- [104]L. Benini, G. De Micheli and F. Vermeulen, "Finite-state machine partitioning for low power," 1998 IEEE International Symposium on Circuits and Systems (ISCAS), Monterey, CA, USA, 1998, pp. 5-8.
- [105]M. Hindi, &R.V Yampolskiy, Genetic algorithm applied to the graph coloring problem. In Proc. 23rd Midwest Artificial Intelligence and Cognitive Science Conf (pp. 61-66). 2012.
- [106]D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning," Operations Research, vol. 39, no. 3, pp. 378-406, 1991.
- [107]A. Hertz and D. D. Werra, "Using Tabu Search Techniques for Graph Coloring," Computing, vol. 39, no. 4, pp. 345-351, 1987.
- [108] D. Bertsimas and J. Tsitsiklis, "Simulated Annealing," Statistical Science, vol. 8, no. 1, pp. 10-15, 1993.

- [109]M. Markaki, I. Kassotakis and A. Vasilakos, "An adaptive genetic algorithm for channel sharing in high-speed networks," IEEE GLOBECOM 1998, Sydney, NSW, Australia, 1998, pp. 2658-2663 vol.5.
- [110]R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-Valued Minimization for PLA Optimization," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 6, no. 5, pp. 727-750, September 1987.
- [111]P. Sapiecha, M. Perkowski, and T. Luba, "Decomposition of Information Systems Based on Graph Coloring Heuristics," Symposium on Modeling, Analysis and Simulation, CESA'96 !MACS Multiconference. Lille -France, July 9-12.
- [112]U. Ferrandino, M. Traiola, M. Barbareschi, A. Mazzeo, P. Fiser and A. Bosio, "Synthesis of Finite State Machines on Memristor Crossbars," 2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Budapest, Hungary, 2018, pp. 107-112,