Journal of Circuits, Systems, and Computers
Vol. 15, No. 5 (2006) 719–738
© World Scientific Publishing Company

World Scientific
www.worldscientific.com

AN ALGORITHM FOR MINIMUM SPACE QUANTUM BOOLEAN CIRCUITS CONSTRUCTION

I-MING TSAI* and SY-YEN KUO†

Department of Electrical Engineering,
National Taiwan University,
No.1, Sec. 4, Roosevelt Road, Taipei, Taiwan, 106
**imtsai@cht.com.tw*
†*sykuo@cc.ee.ntu.edu.tw*

Revised 28 May 2006

Implementing a quantum computer at the circuit level has emerged as an important field of research recently. An important topic of building a general-purpose quantum computer is to implement classical Boolean logic using quantum gates and devices. Since the Toffoli gate is universal in classical Boolean logic, any classical combinational circuit can be implemented by the straightforward replacement algorithm with auxiliary qubits as intermediate storage. However, this inefficient implementation causes a large number of auxiliary qubits to be used. In this paper, a systematic procedure is proposed to derive a minimum space quantum circuit for a given classical combinational logic. We first formulate the problem of transforming an m-to-n bit classical Boolean logic into a t-bit unitary quantum operation. The eligible solution set is then constructed such that a solution can be found simply by selecting any member from this set. Finally, we show that the algorithm is optimal in terms of the space consumption.

Keywords: Computer circuit design; quantum circuits.

1. Introduction

Since the theoretical models of quantum computer were introduced in the early 1980s,[1–3] a great deal of research efforts has been focused in the field of *Quantum Information Science*. Recently, quantum computing has expanded rapidly due to the discovery of Shor's prime factorization and Grover's fast database search algorithm.[4,5] To implement a quantum algorithm, it is necessary that the algorithm can be realized using elementary quantum gates. Not long after Deutsch proposed his theoretical model of quantum computers, he demonstrated that a three-bit quantum gate is universal and capable of realizing any unitary operation.[6] Independently, DiVincenzo and Barenco showed that two-bit gates are sufficient to implement any unitary operation.[7,8] Furthermore, Yao pointed out that any function computable in polynomial time by a quantum Turing machine has a polynomial-size

quantum circuit.[9] All these results make experimental implementation of quantum circuits more practical.

Unlike traditional circuits that are built on top of Boolean logic, a quantum circuit is actually a unitary matrix involving superposition and phase manipulation. To distinguish the difference between these two concepts, traditional circuits involving binary logic functions (such as **AND**, **OR**) are called *classical* circuits, as a contrast to *quantum* circuits that perform a unitary operation in a complex vector space. The problem of implementing a unitary matrix into a sequence of quantum gates has been studied in many different ways.[10–13] Previous study shows that any classical combinational circuit can be implemented by the straightforward replacement algorithm with auxiliary qubits as intermediate storage. However, this inefficient implementation causes a large number of auxiliary qubits to be used. In this paper, a systematic procedure is proposed to derive a minimum space quantum circuit for a given classical combinational logic. We first formulate the problem of transforming an m-to-n bit classical Boolean logic into a t-bit unitary quantum operation. The eligible solution set is then constructed such that a solution can be found simply by selecting any member from this set. Finally, we show that the algorithm is optimal in terms of the space.

The rest of this paper is organized as follows: Section 2 introduces basic notations and preliminaries. Section 3 formulates the problem. Section 4 describes the construction process and space consumption issues. Finally, conclusions are given in Sec. 5.

2. Preliminaries

A two-level quantum system can be represented using a basis consisting of two eigenstates, denoted by $|0\rangle$ and $|1\rangle$, respectively. These states can be either spin states of a particle ($|0\rangle$ for spin-up and $|1\rangle$ for spin-down) or energy levels in an atom ($|0\rangle$ for ground state and $|1\rangle$ for excited state). These two states can be used to simulate the classical binary logic.

A classical binary logic value must be either **ON** (1) or **OFF** (0), but not both at the same time. However, a bit in a quantum system can be any linear combination of these two states, so we have the quantum state of a two-level system $|\psi\rangle$ as

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle\,, \tag{1}$$

where c_0, c_1 are complex numbers and $|c_0|^2 + |c_1|^2 = 1$. In column matrices, this is written as

$$|\psi\rangle = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}\,. \tag{2}$$

The state shown above exhibits a unique phenomenon in quantum mechanics, which is called *superposition*. When a particle is in such a superposed state, it has a part corresponding to $|0\rangle$ and a part corresponding to $|1\rangle$ at the same time. When you measure the system, it will be projected to either $|0\rangle$ or $|1\rangle$. The overall

probability for each state is given by the absolute square of its amplitude. Taking nuclear spin as an example, the quantum state describing a spin state could be

$$|\psi\rangle = \sqrt{0.2}|0\rangle + \sqrt{0.8}|1\rangle = \begin{pmatrix} \sqrt{0.2} \\ \sqrt{0.8} \end{pmatrix} . \tag{3}$$

Upon a measurement, $|c_0|^2 = 0.2$ and $|c_1|^2 = 0.8$ represent the probabilities of obtaining $|0\rangle$ and $|1\rangle$, respectively. This means we have 20% of the chance that the result is $|0\rangle$ (spin-up) and 80% of the chance that the result is $|1\rangle$ (spin-down). Obviously, the sum of $|c_0|^2$ and $|c_1|^2$ must be 1 to satisfy the probability rule. To distinguish the above system from the classical binary logic, a bit in such a two-level quantum system is referred to as a quantum bit, or *qubit*.

Two or more qubits can also form a quantum system jointly. A two-qubit system is spanned by the basis of the tensor product of their own spaces. Hence, the joint state of qubit a and qubit b is spanned by $|00\rangle_{ab}$, $|01\rangle_{ab}$, $|10\rangle_{ab}$, and $|11\rangle_{ab}$, i.e.,

$$|\phi\rangle_{ab} = c_0|00\rangle_{ab} + c_1|01\rangle_{ab} + c_2|10\rangle_{ab} + c_3|11\rangle_{ab} \, , \tag{4}$$

where c_0, c_1, c_2, c_3 are all complex numbers and $|c_0|^2 + |c_1|^2 + |c_2|^0 + |c_3|^2 = 1$. For example, a legal two-qubit quantum system could be

$$|\phi\rangle = \sqrt{0.1}|00\rangle + \sqrt{0.2}|01\rangle + \sqrt{0.3}|10\rangle + \sqrt{0.4}|11\rangle = \begin{pmatrix} \sqrt{0.1} \\ \sqrt{0.2} \\ \sqrt{0.3} \\ \sqrt{0.4} \end{pmatrix} . \tag{5}$$

The notations described above can be generalized to multiple-qubit systems. For example, in a three-qubit system, the space is spanned by a basis consisting of eight elements ($|000\rangle_{abc}$, $|001\rangle_{abc}, \ldots, |111\rangle_{abc}$).

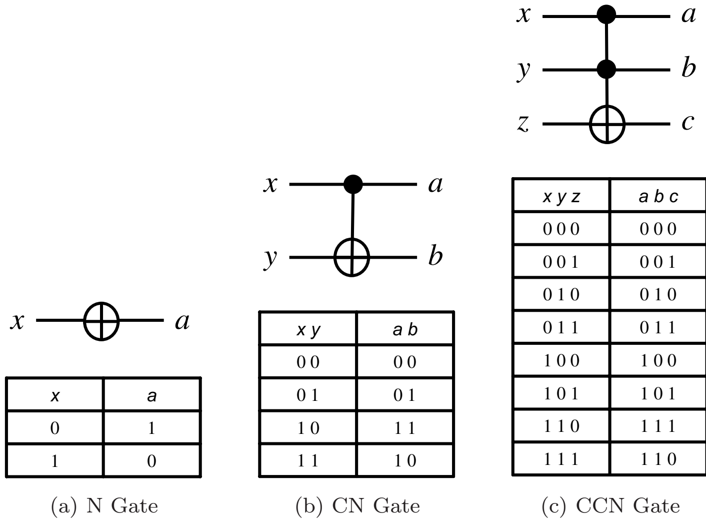
A quantum system can be manipulated in many different ways, called *quantum gates*. A quantum gate can be represented in the form of a matrix operation. For example, a quantum "*Not*" (**N**) gate applied on a single qubit can be represented by multiplying a 2×2 matrix

$$N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \tag{6}$$

which changes the quantum state from $|1\rangle$ to $|0\rangle$ and from $|0\rangle$ to $|1\rangle$, as

$$N \cdot \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_0 \end{pmatrix} . \tag{7}$$

The symbol of an **N** gate is shown in Fig. 1(a). Note that the horizontal line connecting the input and the output is not a physical wire as in classical circuits. It represents a qubit under time evolution.



(a) N Gate (b) CN Gate (c) CCN Gate

Fig. 1. The symbol and bit-wise operation for (a) **N**, (b) **CN**, and (c) **CCN** gate.

Similarly, a two-bit gate can be represented by a 4×4 matrix. For example, a "*Control-Not*" (**CN**) gate is represented by

$$CN = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} . \tag{8}$$

As shown in Fig. 1(b), a **CN** gate consists of one *control* bit x, which does not change its value, and a *target* bit y, which changes its value only if $x = 1$. Assuming the first bit is the control bit, the gate can be written as $CN(|x, y\rangle) = |x, x \oplus y\rangle$, where "$\oplus$" denotes exclusive-or. In matrix form, a **CN** gate changes the probability amplitude of a quantum system as follows:

$$CN \cdot \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ c_3 \\ c_2 \end{pmatrix} . \tag{9}$$

It is noteworthy that, although the function of this quantum gate can be described using binary logic, it has to be realized in nanometer scale to take advantages of superposition and phase manipulation. Taking a **CN** gate on two nuclear spins in a Nuclear Magnetic Resonance (NMR) environment as an example, the conditional logic can be realized by the *coupling* of the two nuclear spins, so the

target qubit is influenced by the control qubit in the way we desired (inverted if and only if the control qubit is 1) while the control qubit is not changed.

An example of a three-bit gate is the *Control-Control-Not* (**CCN** or *Toffoli*) gate. It consists of two control bits, x and y, which do not change their values, and a target bit z, which changes its value only if $x = y = 1$. The bit-wise operation can be written as $CCN(|x, y, z\rangle) = |x, y, (x \cdot y) \oplus z\rangle$. The symbol of a **CCN** gate and its bit-wise operation are summarized in Fig. 1(c).

A generalization of the three-bit Toffoli (**CCN**) gate is the n-bit Toffoli (**T**) gate. The behavior of a **T** gate can be described using three parameters, S (set), R (reset), and I (invert), with

$$\begin{aligned} &(1)\ S, R, I \in \{0,1\}^n\,, \\ &(2)\ \Delta(I, \{0\}^n) = 1\,, \\ &(3)\ S \cdot R = R \cdot I = S \cdot I = \{0\}^n\,, \end{aligned} \tag{10}$$

where $\Delta(x, y)$ denotes the Hamming distance between x and y and "$\cdot$" (dot) stands for bit-wise **AND** operation. The function of a **T** gate is similar to that of a three-bit Toffoli gate. All input bits are left unchanged while the target bit is inverted conditionally. In the notation shown above, S and R are binary digits that mark the positions of the control bits. The bits that are set in S specify the control bits which have to be 1's to activate the logic. Similarly, the bits that are set in R specify the control bits which have to be 0's to activate the logic. I represents the (one and only one) target bit to be inverted when the conditions of S and R are satisfied. Those bits that are not specified in either S, R, or I are *do not care* bits. Throughout this paper, the functionality of a **T** gate will be written as $\mathbf{T(S, R, I)}$. Whenever necessary, an optional bit order can be specified as subscript to denote the order of the control and target bits. Using this notation, the **CCN** gate in Fig. 1(c) can be written as $\mathbf{T}(110, 000, 001)_{xyz}$, and the **CN** gate in Fig. 1(b) is written as $\mathbf{T}(10, 00, 01)_{xy}$.

The symbol of a **T** gate can be generalized from a **CCN** gate. As shown in Fig. 2, black dots and white circles denote the positions of S and R bits, respectively. For example, gate 1 denotes $\mathbf{T}(1100, 0001, 0010)_{abcd}$ and gate 5 represents $\mathbf{T}(0001, 0100, 1000)_{abcd}$. Note that, the square (or even powers) of a **T** gate (e.g., gates 3 and 4) turns out to be an identity. A *do not care* bit is generated when two neighbor **T** gates (e.g., gates 1 and 2) differ in only one bit between their control parts. These are similar to classical Boolean reduction rules, which allow us to optimize the circuits.

Further generalization of the quantum gates described above involves *rotation* and *phase shift*. They control the phase difference and relative contributions of the eigenstates to the whole state. Notice that, to satisfy the probability rule, any quantum gate must be unitary in their matrix form U, i.e.,

$$UU^\dagger = I\,, \tag{11}$$

where $U^\dagger$ is the conjugate transpose of U.

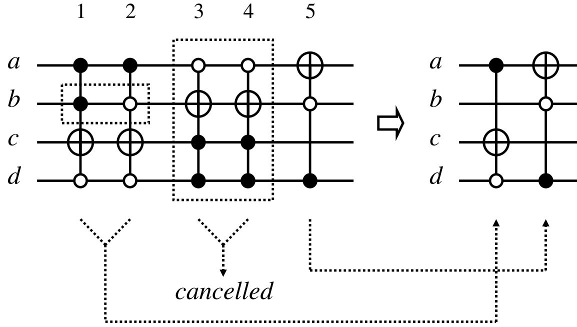


Fig. 2. Examples of **T** gates and circuit simplification.

Just like **AND** and **NOT** form a universal set for classical Boolean circuits, the **CCN** gate is also universal in quantum Boolean circuits. If we set $x = y = 1$, then $c = \bar{z}$, thus the **CCN** gate simulates a classical **NOT** gate. If we set $z = 0$, then $c = x \cdot y$, thus the **CCN** gate simulates a classical **AND** gate. However, it is not enough to perform quantum Boolean logic by using only **NOT** and **AND** gates. Another important quantum Boolean logic function is **FANOUT**, which takes one bit as input and gives two copies of the same bit value as output. In the classical world, this can be achieved simply by a metallic contact. But according to the law of quantum mechanics, it is a nontrivial function and must be performed explicitly. One way to do this is to set $x = 1$ and $z = 0$ in a **CCN** gate, then we get $b = c = y$. However, to save the resources, usually the **NOT** and **FANOUT** functions are not implemented using **CCN** gates. By sending the input into a quantum **N** gate, we can invert the quantum state and hence simulate a classical **NOT** operation. By setting $y = 0$ in a **CN** gate, we have $a = b = x$ and get a **FANOUT** operation. These allow us to implement **NOT** and **FANOUT** using only one- and two-bit quantum gates.

In general, any truth table can be implemented using quantum gates. Taking an m-to-n bit circuit as an example, a truth table consists of two parts, an α table ($2^m \times m$ in size) for input, and a β table ($2^m \times n$ in size) for output [see Fig. 6(a) as an example]. Each row in the α table contains an m-bit input pattern, while the same row of the β table holds the corresponding n-bit output. Throughout this paper, we will use the notation $A[i][*]$ to denote the ith row of a table, and $A[i][m:n]$ to denote the positions in the ith row with column index from m to n. Similar notations are used to denote a column block. In addition, for a table we will follow the convention that the row indices are arranged in ascending order from top to bottom, and the column indices are arranged in ascending order from right to left. With these notations, given a truth table like Fig. 6(a), the output for $\alpha[i][*]$ is indicated by $\beta[i][*]$.

Similar to the classical case, a *Quantum Transformation Table* is used to describe a t-bit quantum Boolean logic. A quantum transformation table consists of two

parts, a ψ table ($2^t \times t$ in size) for input, and a ϕ table of the same size for output. Each column represents a qubit. Each row of the ψ table ($\psi[i][*]$) contains a t-bit input pattern, while the same row of the ϕ table ($\phi[i][*]$) holds the corresponding t-bit output. Because a quantum operation is a reversible unitary transformation, the 2^t rows in the ϕ table are simply a permutation of the input patterns. Given a classical truth table, we will show how to build the corresponding quantum transformation table in Sec. 4.

Another important property regarding a quantum Boolean operation is *permutation*. Since the time evolution of any quantum transformation is a unitary and logically reversible process, any quantum Boolean logic can be represented using a permutation. A permutation is a one-to-one and onto mapping from a finite order set onto itself. A typical permutation P is represented using the symbol

$$P = \begin{pmatrix} a & b & c & d & e & f \\ d & e & c & a & f & b \end{pmatrix} . \tag{12}$$

This permutation changes $a \to d$, $d \to a$, $b \to e$, $e \to f$, and $f \to b$, with state c remaining unchanged. A permutation can also be expressed as disjoint *cycles*. A cycle includes its members in a list like

$$C = (e_1, e_2, \ldots, e_{n-1}, e_n) . \tag{13}$$

The order of the elements describes the permutation. For example, in Eq. (13), the cycle takes $e_1 \to e_2$, $e_2 \to e_3, \ldots, e_{n-1} \to e_n$, and finally $e_n \to e_1$. The number of elements in a cycle is called *length*. A cycle with length 1 is called a *trivial* cycle, which can be ignored as it does not change anything. A cycle of length 2 is called a *transposition*. Using this notation, the same permutation P shown in Eq. (12) can be written as

$$P = (a, d)(c)(b, e, f) = (a, d)(b, e, f) . \tag{14}$$

In the language of permutation, a quantum Boolean logic gate can be expressed as cycles. For example, a **CN** gate is indicated by $P_{CN} = (10, 11)$, performing the mapping $10 \to 11$ and $11 \to 10$, leaving all other states unchanged. Similarly, a three-bit **CCN** gate is indicated by $P_{CCN} = (110, 111)$. A **T** gate without a *do not care* bit denotes an adjacent state transposition (i.e., transposition between two states with Hamming distance 1), while a **T** gate with k do not care bits is actually a group of 2^k adjacent state transpositions.

3. Problem Formulation

As described previously, a classical combinational circuit can be transformed into its quantum version by replacing the classical gates with their corresponding quantum counterparts. An example of a classical half adder, which will be used throughout this paper, is shown in Fig. 3.

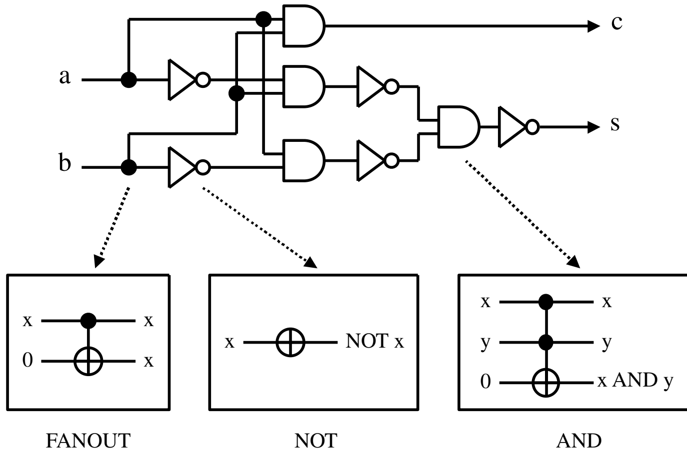


Fig. 3. Direct replacement of classical gates in a half adder.

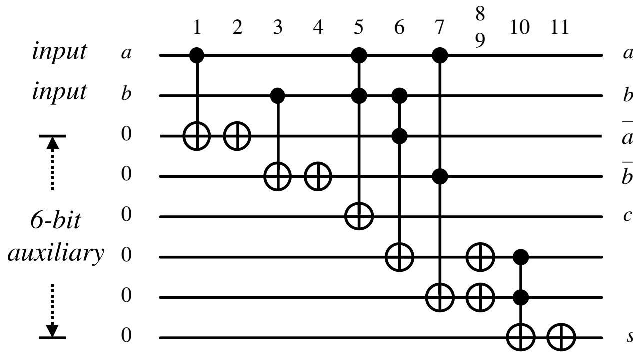


Fig. 4. Straightforward implementation of a quantum half adder.

As shown in the figure, the output *sum* and *carry* are

$$s = \overline{a} \cdot b + a \cdot \overline{b}\,, \tag{15}$$

$$c = a \cdot b\,. \tag{16}$$

The straightforward quantum version of the same circuit is shown in Fig. 4. Two **CN** gates (gates 1 and 3) and two **N** gates (gates 2 and 4) are used to generate $\bar{a}$ and $\bar{b}$. One **CCN** gate (gate 5) is used to generate $a \cdot b$. Gates 6 and 7 provide the function of two **AND** gates, gates 8 through 11 implement the **OR** function in Eq. (15). Note that the **OR** function is implemented using DeMorgan's law,

$$p + q = \overline{\overline{p} \cdot \overline{q}}\,, \tag{17}$$

so it is substituted using three **N** gates and one **CN** gate.

One disadvantage of the straightforward replacement is that both **FANOUT** and **AND** operations need auxiliary qubits. As we can see in Fig. 4, for a classical half adder, eight qubits and 11 quantum gates are used in the straightforward implementation, including six auxiliary qubits. In general, auxiliary bits are necessary to perform reversible logic, but, as we will see later, the straightforward substitution is inefficient in terms of both space (i.e., the number of qubits) and time (i.e., roughly the number of gates).

In the following sections, we will describe an algorithm that can be used to convert a given classical m-to-n bit combinational logic into its quantum version with minimum space (i.e., the number of qubits being used). The problem is formulated as follows:

Problem 1. Given a classical m-to-n bit combinational logic $C\colon A(\{0,1\}^m) \rightarrow B(\{0,1\}^n)$ and an integer p ($0 \leq p \leq m$), construct a t-bit quantum operation $Q\colon \Psi(\{0,1\}^t) \rightarrow \Psi(\{0,1\}^t)$ with the smallest t, such that for each classical mapping from $\alpha = \alpha_0\alpha_1\cdots\alpha_{m-1} \in A$ ($\alpha_i \in \{0,1\}$) to $\beta = C(\alpha) = \beta_0\beta_1\cdots\beta_{n-1}$ ($\beta_i \in \{0,1\}$), there exist two states $\psi = \psi_0\psi_1\cdots\psi_{m-1}\cdots\psi_{t-1} \in \Psi$ and $\phi = \phi_0\phi_1\cdots\phi_{t-1} \in \Psi$ satisfying

(1) $\psi_i = \alpha_i$, for $i = 0, 1, \ldots, m-1$,
(2) $\psi_i = 0$, for $i = m, m+1, \ldots, t-1$,
(3) $Q(\psi) = \phi$,
(4) $\phi_i = \alpha_i$, for $i = 0, 1, \ldots, p-1$,
(5) $\phi_i = \beta_{i-p}$, for $i = p, p+1, \ldots, p+n-1$.

Each of the criteria is shown graphically in Fig. 5. For each input (α_i) and output (β_i) pair in the classical circuits, the quantum Boolean circuits (permutation) shall map the same input ($\psi_i = \alpha_i$), together with $t-m$ auxiliary qubits, to its corresponding output ($\phi_i = \beta_i$). Initially, all auxiliary qubits have to be reset to 0's by the reset circuitry. The input state of all qubits will be overwritten by the output,

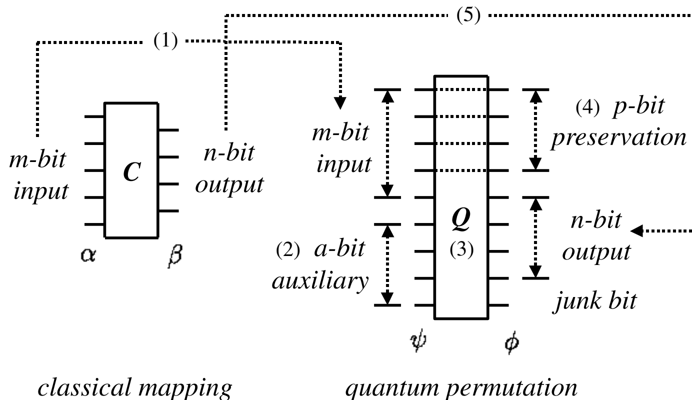
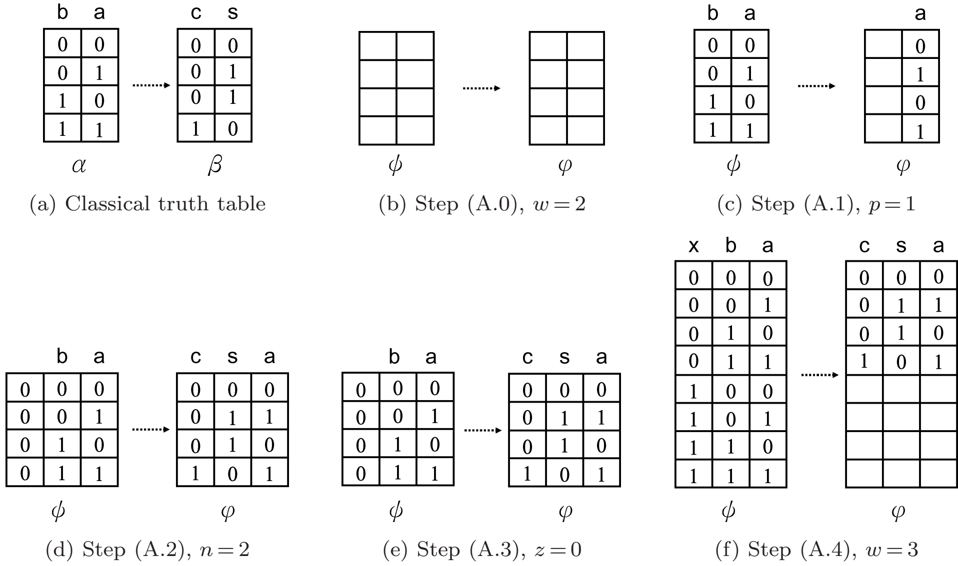


Fig. 5. Problem formulation.



(a) Classical truth table (b) Step (A.0), $w = 2$ (c) Step (A.1), $p = 1$

(d) Step (A.2), $n = 2$ (e) Step (A.3), $z = 0$ (f) Step (A.4), $w = 3$

Fig. 6. Building the quantum constraint table for a half adder.

except p preserved qubits. Preserved qubits are the input qubits that have to remain unchanged and are intended to be used as the input to other circuits. The construction process and space consumption issues are described in the following sections.

4. Quantum Boolean Circuits Construction

4.1. *Quantum Boolean constraint table*

To build the quantum transformation table for a given classical logic, the constraints must be clearly identified first. The steps to identify the constraints based on the classical circuits are shown below.

Input:

(1) An m-to-n bit classical truth table. As shown in Fig. 6(a), an input in the α table is mapped to the corresponding entry in the β table.

(2) An integer p, denoting the number of qubits that are to be preserved. Without loss of generality, assume qubits 0 to $p - 1$ ($0 < p \leq m$) are the qubits to be preserved. In case p is 0, no input qubit needs to be preserved.

Output: A quantum constraint table.

Procedures:

(A.0) *Initialization*

Let $w = \max(m, n)$ be the current table width, and prepare two empty tables, ψ and ϕ, such that both are of size $2^w \times w$. This is shown in Fig. 6(b). Throughout this procedure, the size of both tables will be dynamically expanded by adding columns and rows from low index to high index.

(A.1) *Preserve the input qubits*
For each row i ($0 \leq i \leq 2^m - 1$), copy $\alpha[i][0 : m-1]$ to $\psi[i][0 : m-1]$. If no input qubit needs to be preserved ($p = 0$), go to Step (A.2). Otherwise ($p > 0$), copy the preserved qubits from $\alpha[i][0 : p-1]$ to $\phi[i][0 : p-1]$. An example of preserving the input qubit a ($p = 1$) is shown in Fig. 6(c).

(A.2) *Assign the output qubits*
Since qubits 0 to $p-1$ have been used to preserve the input qubits, assign qubits p to $p+n-1$ as the output qubits. In case $p+n > w$, expand the width of both tables by adding $p+n-w$ columns, set $w = p+n$, and fill in the new columns in the ψ table as all 0's. Otherwise, no column needs to be added. For each row i ($0 \leq i \leq 2^m - 1$), copy $\beta[i][0 : n-1]$ to $\phi[i][p : p+n-1]$. The result of assigning qubit 2 and 3 as sum and carry is shown in Fig. 6(d).

(A.3) *Distinguish each output state*
Find the largest number k and $i_1, i_2, \ldots, i_k$ such that $\phi[i_1][*] = \phi[i_2][*] = \cdots = \phi[i_k][*]$; set $z = \lceil \log_2 k \rceil$. If $p+n+z > w$, expand the width of both tables by adding $p+n+z-w$ columns, set $w = p+n+z$, and fill in the new columns in the ψ table as all 0's. Otherwise, no column needs to be added. For each row $\phi[i][*]$ ($i \in \{i_1, i_2, \ldots, i_k\}$), assign a different binary pattern to the newly added columns to make each row different. In the half adder example, it happens that the patterns in the ϕ table are all different, so $z = 0$ and no extra column is added, as shown in Fig. 6(e).

(A.4) *Complete the constraint table*
If new columns have been added in Step (A.2) or (A.3), expand both ψ and ϕ tables to be 2^w rows in length. For the ψ table, follow the sequence and fill in each row with a different binary pattern in ascending order. For the ϕ table, fill in the new entries with blanks (*null*). The final quantum constraint table is shown in Fig. 6(f).

Based on the constraints derived from the classical logic, the quantum transformation table is now partially constructed. In the ψ table, $\psi[i][*]$ are different bit patterns in ascending order, representing different input patterns. In the corresponding row of the ϕ table, $\phi[i][*]$ hold information for the output. There are three possibilities for the patterns in $\phi[i][*]$:

(1) A fully specified qubit pattern. This completely defines the output qubit pattern for the input $\psi[i][*]$.
(2) A partially specified qubit pattern. In this case, some of the qubits are defined, while others can be arbitrary binary values. Usually this happens when new columns are added in Step (A.3), or in a circuit with $m \gg n$.
(3) A nonspecified qubit pattern. In this case, any binary pattern is eligible. This usually happens in the lower part of the table, when the corresponding auxiliary qubits are not 0's.

As stated previously, the time evolution of any quantum transformation is a unitary and logically reversible process; thus any quantum Boolean logic can be represented using a permutation. The permutation can be completed simply by filling in the blanks in the constraint table to make it a one-to-one and onto mapping. This is done in the following section.

4.2. *Quantum Boolean constraint digraph*

A quantum constraint table can be represented using a quantum constraint digraph. A quantum constraint digraph has 2^t vertices ($t = 3$ in this example), corresponding to each of the 2^t rows in the constraint table. Each directed link from $v_{\rm s}$ to $v_{\rm d}$ represents a source–destination mapping from $v_{\rm s}$ to $v_{\rm d}$ in the quantum constraint table. To build a complete quantum constraint digraph containing partially specified patterns, we introduce the following definition.

Definition 1. Let $A, B = \{0, 1, null\}^n$ be two n-digit strings and A_i, B_i $(0 \le i \le n-1)$ denote the ith digit of the two strings.

(1) Two digits A_i and B_i are compatible, denoted by $A_i \cong B_i$, if and only if either $A_i = B_i$ or one of them is *null*.
(2) String A is compatible to B, denoted by $A \cong B$, if and only if $A_i \cong B_i$ for each i.

According to this definition, a fully specified pattern in the ϕ table has only one compatible pattern, since every bit is fully specified. On the other hand, a partially specified (and nonspecified) pattern has multiple compatible patterns. In this case, a source node in the ψ table can be directed to more than one destination node. More specifically, two types of links can be defined, depending on the pattern of $v_{\rm d}$. A source–destination pair $(v_{\rm s}, v_{\rm d})$ generates a *hard link* if and only if $v_{\rm d}$ is fully specified in $\phi[i][*]$. Hard links define the one-to-one mapping between $v_{\rm s}$ and $v_{\rm d}$. No other links can be generated from the source and no other links can be directed to the destination. On the other hand, if $v_{\rm d}$ is not fully specified in $\phi[i][*]$, then multiple links can be generated from the source. In this case, for each node $v_{\rm m} \cong \phi[i][*]$, one link can be generated from $v_{\rm s}$ to $v_{\rm m}$. The notation $v_{\rm m} \cong \phi[i][*]$ is used to denote all the states that are compatible to $\phi[i][*]$. In general, if u $(u < t)$ qubits are specified in $\phi[i][*]$ then 2^{t-u} possible links can be generated from the source. These are called *soft links.* Notice that, it is possible for a soft link to have the same destination as a hard link. In this case, the soft link must be removed unconditionally to avoid the conflict. The reason that conflict links are not eligible is that there is no alternative destination for a hard link; hence it gets higher priority. For soft links, there are always other destinations for the source node, so they get lower priority. This is why they are called "soft" links.

In summary, a quantum constraint digraph is constructed by adding all hard and soft links but excluding the conflict links in a digraph with 2^t vertices. Each of

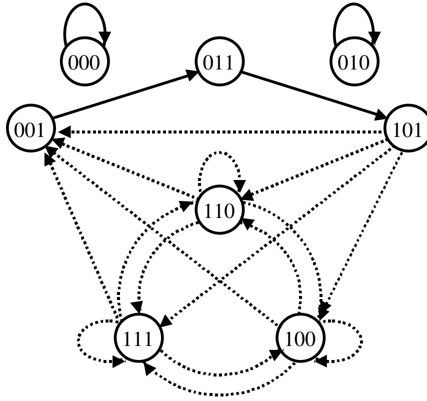


Fig. 7. Constraint digraph with hard and soft links.

the vertices represents a quantum state $|\psi[i][*]\rangle$. The quantum constraint digraph for Fig. 6(f) is shown in Fig. 7. In this figure, solid arrows represent hard links; these include $|001\rangle \to |011\rangle$, $|011\rangle \to |101\rangle$, $|000\rangle \to |000\rangle$, and $|010\rangle \to |010\rangle$. All other dash arrows denote soft links.

Since a permutation consists of disjoint cycles covering all the vertices, the problem of building a permutation table is transformed into a problem of finding disjoint cycles that cover all the hard links and nodes in the constraint digraph. This problem is formulated as follows:

Problem 2. Given a digraph $G = (V, E = E_{\rm h} \cup E_{\rm s})$, where $E_{\rm h}$ denotes hard links and $E_{\rm s}$ denotes soft links, find a family of sets $S = \{S_i \mid S_i = \{v_0^i, v_1^i, \ldots, v_{n-1}^i\}, v_j^i \in V\}$ satisfying $\bigcup S_i = V$ and $\bigcap S_i = \emptyset$, such that the corresponding family of cycles $C = \{C_i | C_i = (v_0^i, v_1^i, \ldots, v_{n-1}^i), v_j^i \in V\}$ covers the hard links, i.e., $\forall e \in E_{\rm h}, e \in \bigcup C_i$.

Because the quantum constraint table is built in such a way that permutations are embedded in the digraph, the task of finding disjoint covering cycles becomes straightforward. It is essentially the same thing as filling in the $t - u$ blank qubits in the ϕ table and selecting some of the soft links. This is described in the next section.

4.3. *Finding a solution*

In Step (A.3) of the constraint table building process, we have eliminated the possibility of getting entries with the same value in the ϕ table. This made it possible for building a permutation by filling in the blanks (i.e., choosing a soft link). However, usually there are multiple permutations satisfying the constraints. The following algorithm provides a simple way to pick a permutation from the solution set.

Input: A constraint digraph $G = (V, E = E_{\mathrm{h}} \cup E_{\mathrm{s}})$.
Output: A permutation (of V) covering the existing links $e \in E_{\mathrm{h}}$.
Procedures:

(B.1) Arbitrarily select a node v_i and let $v_{\mathrm{s}} = v_i$.

(B.2) If v_{s} is the source of a hard link [i.e., $(v_{\mathrm{s}}, v_{\mathrm{d}}) \in E_{\mathrm{h}}$], mark the hard link as selected. Otherwise, arbitrarily mark a soft link [i.e., $(v_{\mathrm{s}}, v_{\mathrm{d}}) \in E_{\mathrm{s}}$] as selected; remove all other soft links with either v_{s} as the source or v_{d} as the destination. Set $v_{\mathrm{s}} = v_{\mathrm{d}}$.

(B.3) If $v_{\mathrm{s}} \neq v_i$, go to Step (B.2). Otherwise, a cycle is found; record the cycle and delete all the nodes in that cycle.

(B.4) If not all nodes are included in a cycle, go to Step (B.1). Otherwise, complete the permutation table according to the cycles.

This algorithm guarantees that a disjoint cycle covering can be found in a straightforward way. The reason is that a mapping is a permutation if and only if it is one-to-one and onto. This means, regardless of the mapping, a permutation from ψ to ϕ exists if and only if both ψ and ϕ contain the same set of nonduplicating entries. This property holds throughout the whole cycle finding process for the following reasons.

(1) Recall that in Steps (A.0) through (A.4), two tables with the same number of entries (i.e., 2^t) are built. The possibility of duplicated patterns is eliminated by distinguishing all entries in Step (A.3).

(2) Moreover, a cycle can be represented in a mapping table using a set of source–destination arrows. Notice that if the links of a cycle are drawn between ψ and ϕ, it is essentially a nonduplicating set of entries with an equal number of elements in both tables. Deleting a cycle of length n from a permutation of length p leaves the same number (i.e., $p - n$) of nonduplicating entries in both tables. Thus, by definition, a permutation can still be found.

(3) The cycle deleting process makes the number of entries a strictly decreasing function until it eventually becomes 0. A disjoint cycle covering will then be found.

Obviously, the minimum effort construction is to connect the "tail" to the "head" of each consecutive hard link sequence and then loop every single node back to itself. Using this approach, a circuit construction together with its corresponding permutation tables is shown in Fig. 8. In this example, soft links $(101, 001)$, $(110, 110)$, and $(111, 111)$, are selected; all other soft links are removed.

Since there are many degrees of freedom in assigning the bit patterns in the quantum constraint table, it is possible to construct a disjoint cycle covering in many different ways. Actually, a traversal algorithm which demonstrates how to list all such permutations can be easily constructed. However, all these alternative solutions have the same space consumption, i.e., all the solutions use the same number of qubits to construct the given function. As an example, a more sophisticated construction for the same quantum constraint table is shown in Fig. 9.

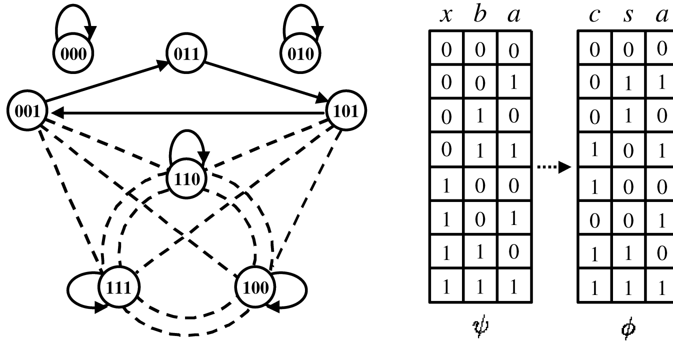


Fig. 8. An example of disjoint cycle covering.

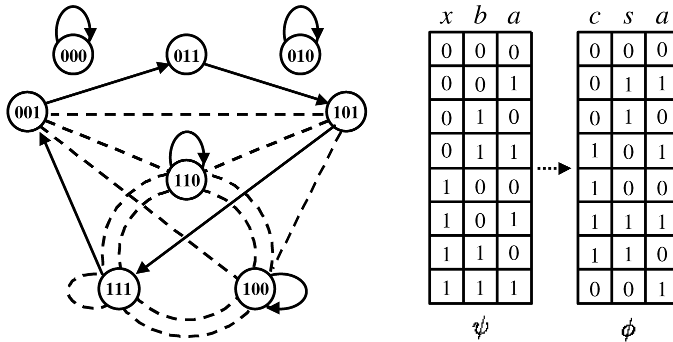


Fig. 9. Another example of disjoint cycle covering.

4.4. *Circuit implementation*

A permutation can be represented using a unitary matrix with elements of either 0 or 1 [e.g., Eqs. (6) and (8)]. The problem of implementing a unitary matrix into a sequence of one- or two-bit elementary quantum gates has been studied in many different ways.[14] For the purpose of illustration, a new intuitive construction of permutation is described as follows:

(1) A permutation consists of one or multiple disjoint cycles. Since disjoint cycles commute, each cycle in the permutation can be implemented individually.

(2) Given a general cycle $C = (p_0, p_1, p_2, \ldots, p_{n-1})$, it can be constructed using $n-1$ transpositions. For example, following the convention that a sequence of transpositions is performed from right to left, we have

$$C = (p_0, p_1)(p_1, p_2) \cdots (p_{n-3}, p_{n-2})(p_{n-2}, p_{n-1})\,. \tag{18}$$

Notice that the notation of a cycle can be represented in a cyclic fashion and hence can be decomposed into different transposition sequences.

(3) Given any two general states p and q with $\Delta(p, q) = d \neq 0$, the transposition $U = (p, q)$ can be decomposed into $2d - 1$ adjacent state transpositions. This

can be done by finding a list of adjacent states, $s_1, s_2, \ldots, s_{d-1}$, such that for $1 \leq i \leq d-2$, $\Delta(p, s_1) = \Delta(s_i, s_{i+1}) = \Delta(s_{d-1}, q) = 1$. For the list $p, s_1, s_2, \ldots, s_{d-1}, q$, perform the following adjacent state transposition sequence

$$(p, s_1)(s_1, s_2)\cdots(s_{d-2}, s_{d-1})(s_{d-1}, q)(s_{d-2}, s_{d-1})\cdots(s_1, s_2)(p, s_1). \tag{19}$$

Notice that this sequence, starting from (p, s_1), can be done by another symmetrically sequence starting from q

$$(s_{d-1}, q)(s_{d-2}, s_{d-1})\cdots(s_1, s_2)(p, s_1)(s_1, s_2)\cdots(s_{d-2}, s_{d-1})(s_{d-1}, q). \tag{20}$$

(4) Given any two states p and q with $\Delta(p, q) = 1$, the transposition $U = (p, q)$ can be implemented using a $\mathbf{T(S, R, I)}$ gate with

$$S = p \cdot q\,, \quad R = \bar{p} \cdot \bar{q}\,, \quad I = p \oplus q\,. \tag{21}$$

(5) A $\mathbf{T(S, R, I)}$ gate can be further decomposed into one-bit rotation and two-bit control-U gates.[14]

This completes our quantum Boolean circuit construction.

Following the steps described above, the circuit implementation of Fig. 9 can be derived as follows:

(1) Ignoring all trivial cycles, the permutation can be represented as one cycle:

$$C = (101, 111, 001, 011)\,. \tag{22}$$

(2) The cycle can be realized using three transpositions:

$$C = (101, 111)(111, 001)(001, 011)\,. \tag{23}$$

(3) By Eq. (20), the nonadjacent state transposition $(111, 001)$ can be further decomposed into adjacent transpositions with the intermediate state "011". As a result,

$$(111, 001) = (011, 001)(111, 011)(011, 001)\,. \tag{24}$$

Substitute Eq. (24) into Eq. (23); the permutation turns to be a series of adjacent state transpositions:

$$C = (101, 111)(011, 001)(111, 011)(011, 001)(001, 011)\,. \tag{25}$$

(4) Implement the adjacent state transpositions using $\mathbf{T(S, R, I)}$ gates as follows:

$$(001, 011) \Rightarrow \mathbf{T}(001, 100, 010)_{xba}\,, \tag{26}$$

$$(011, 001) \Rightarrow \mathbf{T}(001, 100, 010)_{xba}\,, \tag{27}$$

$$(111, 011) \Rightarrow \mathbf{T}(011, 000, 100)_{xba}\,, \tag{28}$$

$$(011, 001) \Rightarrow \mathbf{T}(001, 100, 010)_{xba}\,, \tag{29}$$

$$(101, 111) \Rightarrow \mathbf{T}(101, 000, 010)_{xba}\,. \tag{30}$$

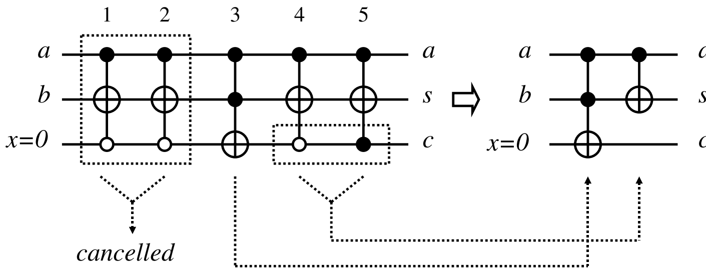


Fig. 10. Circuit implementation of Fig. 9.

Following the convention that the gates are performed from left to right, the result is shown in Fig. 10. Three qubits are used to construct the circuits, including one auxiliary qubit x which is set as $|0\rangle$ initially. Notice that an optional simplification process has been applied in the circuits.

4.5. *Experimental results and complexity analysis*

To verify the proposed algorithm, we wrote a C program under Windows 2000 to implement the core feature of this algorithm. The program takes less than 100 k bytes of memory and includes three internal modules: (1) truth table interface, (2) permutation construction, and (3) gate generation process. Although this is only a preliminary version, it still demonstrates the feasibility of the algorithm with excellent performance. A variety of combinational circuits have been tested; some of the experimental results are

(1) A two-input two-output half adder with one preserved bit was implemented. This is the example we used previously. Without gate optimization, the result showed a series of five **T** gates:
$\mathbf{T}(100, 001, 010)\mathbf{T}(100, 001, 010)\mathbf{T}(110, 000, 010)\mathbf{T}(100, 001, 010)\mathbf{T}(101, 000, 010)$.
(2) A two-input one-output multiplexer with one selection bit was implemented. All three-input bits were preserved. The result showed a series of four **T** gates:
$\mathbf{T}(0110, 1000, 0001)\mathbf{T}(1000, 0110, 0001)\mathbf{T}(1100, 0010, 0001)\mathbf{T}(1110, 0000, 0001)$.
(3) A three-bit majority gate was implemented. The output logic value of a majority gate is the logic value of the majority of the input bits. All input bits were required to be preserved. The result showed a series of four **T** gates:
$\mathbf{T}(0110, 1000, 0001)\mathbf{T}(1010, 0100, 0001)\mathbf{T}(1100, 0010, 0001)\mathbf{T}(1110, 0000, 0001)$.

Obviously, different cycle covering gives different results. In our current implementation, if more than one cycle covering is possible, we randomly select one cycle covering to implement the quantum gates and no circuit optimization is performed. The problem of how to choose a cycle covering so that circuit optimization gives a good result (reducing the total number of gates) is left for further study.

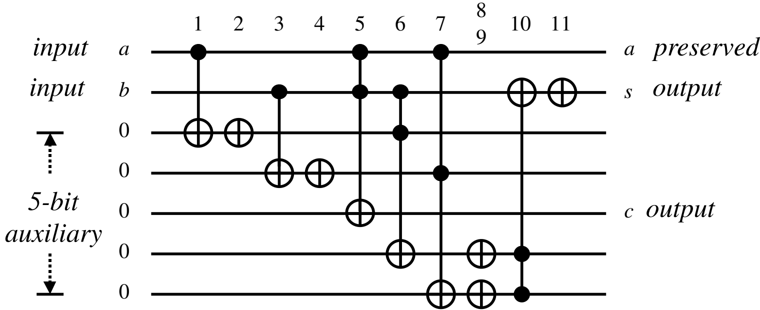


Fig. 11. Overwrite input b gives a slightly compact circuit.

Although this preliminary implementation is limited in size, in principle it can be generalized to deal with large truth tables. Assuming the size of the input (α table plus β table) is $s \times m$ in total, the space complexity (memory consumption) of the program is $s \times m$ for classical truth table. The memory consumption for the quantum transformation table depends on the number of output (n) and preserved (p) bits. In case of $p + n \leq m$, the memory consumption is $2 \times s \times m$. Otherwise, the memory consumption is $2^{(p+n-m+1)} \times s \times m$. However, since the output bits can be implemented individually, in general these output bits can be partitioned into small groups and implemented individually. As a result, even in the case of $p + n > m$, the memory consumption can be kept reasonable according to the computing resources. On the other hand, since the algorithm involves only simple "check and fill" processes, the time complexity is linearly proportional to the space consumption and hence can also be easily kept reasonable.

It is noteworthy that reducing the total number of qubits (space) is an important issue because this makes it easier to maintain the coherence. Consider the same problem (i.e., preserve input a and overwrite input b) using a direct replacement implementation. A slightly more compact result than Fig. 4 is shown in Fig. 11. Although some of the gates can be performed simultaneously, seven qubits have to be used to construct the circuits, including five auxiliary qubits. As we can see, the construction of quantum Boolean circuits using the straightforward algorithm is expensive in space, because some of the auxiliary qubits have to be used to store temporary information. The number of auxiliary qubits needed depends on the complexity of the circuit. Generally speaking, if the circuit is represented using the form of *sum of products*, then more minterms imply more auxiliary qubits.

However, using the proposed algorithm, only a minimum number of qubits are used to achieve the desired function. To calculate the minimum number of qubits to construct the permutation, we have the following observations.

(1) Since a quantum circuit is a state evolution on a set of qubits, to implement a boolean logic with m input and n output bits, the number of qubits must be at least $\max(m, n)$.

(2) The total number of output bits include at least p preserved bits and n output bits. In addition, for a unitary quantum evolution, the quantum transformation table between ψ and ϕ needs to be one-to-one and onto. This implies that the ϕ table shall not contain duplicated entries. Entries with the same value have to be distinguished by at least z extra qubits, as we have done in Step (A.3). So, the total number of qubits must be at least $p + n + z$.

In summary, $t = \max(\max(m, n), p + n + z) = \max(m, p + n + z)$ is the minimum number of qubits to extend the classical mapping C to be a permutation. Since a permutation is unitary and can be implemented using elementary quantum gates without extra qubits, the target boolean function can be realized using t qubits.

5. Conclusions

In this paper, a systematic procedure is proposed to derive a minimum space quantum circuit for a given classical combinational logic. This algorithm has many applications in the field of circuit design automation. For example, the algorithm can be applied to construct the *oracle* in a quantum search process.[5,15] In the quantum search algorithm, an oracle is used to distinguish the target(s) from other elements. Since, in most cases, the target(s) can be identified using classical Boolean expressions, this algorithm can be used to convert any given classical Boolean expressions into a quantum oracle and speedup the search process. Furthermore, since the conventional device architecture will eventually reach its physical limits and we have to take advantage of quantum physics at nanometer scale and continue the computer hardware evolution, this algorithm provides a smooth migration to the next generation circuit design.

References

1. P. Benioff, The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines, *J. Stat. Phys.* **22** (1980) 563–591.
2. R. Feynman, Simulating physics with computers, *Int. J. Theor. Phys.* **21** (1982) 467–488.
3. D. Deutsch, Quantum theory, the Church–Turing principle and the universal quantum computer, *Proc. Roy. Soc. Lond. A* **400** (1985) 97–117.
4. P. Shor, Algorithms for quantum computation: Discrete logarithms and factoring, *Proc. 35th Annual IEEE Symp. Foundations of Computer Science* (1994), pp. 124–134.
5. L. Grover, A fast quantum mechanical algorithm for database search, *Proc. 28th Annual ACM Symp. Theory of Computing* (1996), pp. 212–219.
6. D. Deutsch, Quantum computational networks, *Proc. Roy. Soc. Lond. A* **425** (1989) 73–90.
7. D. DiVincenzo, Two-bit gates are universal for quantum computation, *Phys. Rev. A* **51** (1995) 1015–1022.
8. A. Barenco, A universal two-bit gate for quantum computation, *Proc. Roy. Soc. Lond. A* **449** (1995) 679–683.

9. A. Yao, Quantum circuit complexity, *Proc. 34th Annual IEEE Symp. Foundation of Computer Science* (1993), pp. 352–361.
10. I. M. Tsai and S. Y. Kuo, An algorithm for quantum Boolean circuit construction, *Proc. 2001 IEEE Conf. Nanotechnology* (2001), pp. 111–116.
11. D. Maslov and G. Dueck, Reversible cascades with minimal garbage, *IEEE Trans. Computer-Aided Design* **23** (2004) 1497–1509.
12. L. Storme, A. De Vos and G. Jacobs, Group theoretical aspects of reversible logic gates, *J. Universal Comput. Sci.* **5** (1999) 307–321.
13. V. Shende, A. Prasad, I. Markov and J. Hayes, Synthesis of reversible logic circuits, *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.* **22** (2003) 710–722.
14. A. Barenco, C. Bennett, R. Cleve, D. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin and H. Weinfurter, Elementary gates for quantum computation, *Phys. Rev. A* **52** (1995) 3457–3467.
15. I. M. Tsai, S. Y. Kuo and D. Wei, Quantum Boolean circuit approach for searching an unordered database, *Proc. 2002 IEEE Conf. Nanotechnology* (2002), pp. 315–318.

Copyright of Journal of Circuits, Systems & Computers is the property of World Scientific Publishing Company and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.