BUILDING AN EFFECTIVE GENERAL-PURPOSE QUANTUM SIMULATOR FOR THE DESIGN AND ANALYSIS OF QUANTUM CIRCUITS

by

Anh Quoc NGUYEN

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirement for the degree of Doctor of Philosophy, The City University of New York

2006

UMI Number: 3231998

Copyright 2006 by
Nguyen, Anh Quoc

All rights reserved.

UMI

UMI Microform 3231998
Copyright 2006 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2006

ANH QUOC NGUYEN

All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirements for the degree of Doctor of Philosophy.

Prof. Michael Anshel
City College of New York – CUNY

____________ ____
Date Chair of Examining Committee

Prof. Ted Brown

____________ ____
Date Executive Officer

Dr. Michael Brenner (Mitre Corporation)

Dr. Xiangdong Li (NYC College of Technology)

Prof. Stephen Lucci (City College of New York)

Prof. James Cox (Brooklyn College)
Supervisory Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

BUILDING AN EFFECTIVE GENERAL-PURPOSE QUANTUM SIMULATOR

FOR THE DESIGN AND ANALYSIS OF QUANTUM CIRCUITS

by

Anh Quoc NGUYEN

Adviser: Professor Michael Anshel

Recently quantum computing has achieved significant results both in theoretical and experimental areas and quantum algorithms have drawn the attention of many scientists. At the present there exist several experiments that realized some parts of certain important quantum algorithms. While those experiments work with only up to seven qubits, quantum simulators allow scientists to design algorithms with many more qubits. The existing simulators have one or more of the following restrictions: either they require powerful computers and additional computation packages such as Matlab or Mathematica or they don't provide an easy method for interacting between the classical and quantum parts of an algorithm. More importantly, none of them can simulate a generic 64-qubit circuit. It is important at this stage of quantum computing development to investigate technological improvements in simulating quantum circuit design.

In this thesis we developed a specialized data structure, QSV, for quantum simulators along with new method for calculation of the new state vector after application of a quantum gate in a quantum simulator. We also developed a technique that speeds up the simulation time with our QSV vectors. With those results we built a quantum simulator

that overcomes many shortcomings in other simulators and it has many advanced features.

Our simulator can simulate many 64-qubit circuits and it can directly simulate multiple qubit gates as well as allows users to define gates with up to 10 qubits. The simulation time is much less in comparison with other simulators we tested. The simulator defines specifications so that users can easily design and test quantum circuits. With the simulator we also developed some new quantum circuits and verified many designs and constructions.

Acknowledgements

I deeply appreciate my thesis advisor, Professor Michael Anshel for his guidance, encouragement and support in my research.

I would like to express my gratitude to Doctor Michael Brenner at Mitre Corporation for the proofreading of the thesis and for the discussion to improve it.

I would also like to thank Doctor Joseph Emerson for the discussion that helps me to complete section 4.4 of the dissertation.

Finally, I would like to express special thanks to my wife for her patience, encouragement, discussion and support in all these hard years.

Content

Chapter 1 .... - 1 -

Basic concepts of quantum computation .... - 1 -

1.1. Qubits and evolution of a quantum system .... - 1 -

1.1.1 Qubit .... - 1 -

1.1.2 Multiple qubits .... - 2 -

1.1.3 Evolution of a quantum system .... - 4 -

1.2. Quantum gates and quantum computation .... - 6 -

1.2.1. Quantum gates and graphical representation .... - 6 -

1.2.2. Entanglement, pure and mixed states .... - 9 -

1.2.3. Quantum measurement .... - 10 -

1.3. Quantum parallelism and quantum algorithm .... - 12 -

1.3.1 Quantum parallelism .... - 12 -

1.3.2. Deutsh – Jozsa algorithm .... - 14 -

1.3.3 Grover search algorithm .... - 16 -

1.3.4 Quantum Fourier transform and Shor's quantum factoring algorithm.... - 19 -

1.4. Implementations of quantum computers .... - 23 -

Chapter 2 .... - 26 -

Simulation of quantum computers on a classical computer .... - 26 -

2.1 Quantum simulators – assisting tools for quantum research .... - 26 -

2.2 Important aspects of general purpose quantum simulators .... - 28 -

Chapter 3 .... - 30 -

Building Effective General-Purpose Quantum Simulator .... - 30 -

3.1 Representing a quantum state vector in a classical computer .... - 30 -

3.2 Application of a gate .... - 34 -

3.3 Solution for building an effective quantum simulator .... - 35 -

3.3.1 Gate application technique .... - 35 -

3.3.2 Special data structure for simulation with swapping technique: *QSV* ..... - 36 -

3.3.3 Algorithm to perform swapping two qubits with *QSV* .... - 38 -

3.3.4 Improve simulation time by processing only non-empty sub-vectors ..... - 40 -

3.4 Calculating average simulation time for *qsim* .... - 43 -

3.5 Simulating measurement operations .... - 44 -

3.5.1 Measurement with output only .... - 44 -

3.5.2. Measurement with output and feedback .... - 47 -

3.6. Main features of *qsim* .... - 48 -

Chapter 4 .... - 51 -

Designing and Analyzing Quantum Circuits with *qsim* .... - 51 -

4.1. Concept of designing quantum circuits with classical registers .... - 51 -

4.2. Optimal Reversible Quantum Circuit for Multiplication .... - 55 -

4.2.1 Addition using Quantum Fourier Transform .... - 56 -

4.2.2 Quantum circuit for multiplication using addition in QFT state .... - 57 -

4.3. Preparing arbitrary superposition for a quantum system .... - 60 -

4.3.1. Preparing a superposition for 1-qubit system .... - 60 -

4.3.2 Preparing arbitrary superposition for n-qubit .... - 62 -

4.4. Quantum circuit to generate random state for computation .... - 65 -

4.5 Solovay-Kitaev algorithm and quantum circuit for factoring using only three gate types: Hadamard, T, and CNOT ... - 70 -

4.6 Semi-classical quantum circuit for factorization ... - 76 -

4.7. Future research ... - 81 -

Conclusion ... - 82 -

Appendix A ... - 84 -

Qsim specifications ... - 84 -

A.1 The circuit file ... - 84 -

A.2 Input for a circuit ... - 87 -

A.3 Output for a circuit ... - 88 -

A.4 Built-in gates in *qsim* ... - 90 -

A.4.1 Gates without parameters ... - 90 -

A.4.2 Gates with a special parameter ... - 92 -

A.5 All *qsim* directives ... - 94 -

A.6 Command-line options for *qsim* ... - 96 -

Appendix B ... - 98 -

Sample circuit for the quantum simulator ... - 98 -

Appendix C ... - 100 -

Comparison with other quantum simulators ... - 100 -

Bibliography ... - 104 -

Chapter 1

Basic concepts of quantum computation

1.1. Qubits and evolution of a quantum system

1.1.1 Qubit

A quantum computer is a physical system that functions under the rules of quantum mechanics. The basic storage unit in such a computer is the quantum bit, or qubit, that has two states: zero and one; they are denoted by $|0\rangle$ and $|1\rangle$ respectively. States $|0\rangle$ and $|1\rangle$ are called *basis computational states*. Unlike classical bit that can be in either state 0 or 1 qubit can be in both states at the same time. Such a state is called a *superposition*. The general state of a qubit can be written as:

$$|\psi\rangle = a|0\rangle + b|1\rangle, \tag{1.1}$$

where a and b are complex numbers satisfying condition:

$$|a|^2 + |b|^2 = 1 \tag{1.2}$$

With condition (1.2) the equation (1.1) can be expressed as:

$$|\psi\rangle = e^{i\gamma}\left[\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right], \tag{1.3}$$

where γ, θ, φ are real numbers. The factor $e^{i\gamma}$ is called the *global phase* and can be ignored because it has no observable effect, or one cannot measures it, and (1.3) becomes

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \tag{1.4}$$

The equation (1.4) can visualized as a point on sphere, called Block sphere, as shown in figure 1.1.

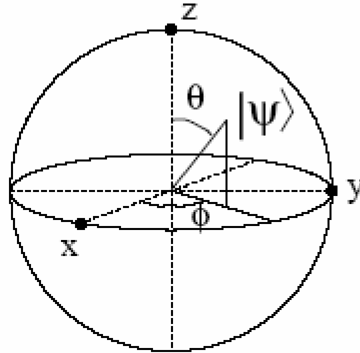


Figure 1.1: Block sphere representation of a state $|\psi\rangle$ of a qubit.

1.1.2 Multiple qubits

Suppose we have a quantum system with two qubits. The state space for two qubits can be constructed from their state spaces for each of them by using the *tensor product* or *Kronecker product.* Let the state of each qubit is $|\psi_1\rangle = a_1|0\rangle + b_1|1\rangle$ and $|\psi_2\rangle = a_2|0\rangle + b_2|1\rangle$ then the state of the whole system is:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle = (a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) \tag{1.5}$$

$$|\psi\rangle = a_1 a_2|0\rangle \otimes |0\rangle + a_1 b_2|0\rangle \otimes |1\rangle + b_1 a_2|1\rangle \otimes |0\rangle + b_1 b_2|1\rangle \otimes |1\rangle, \tag{1.6}$$

where $\otimes$ denotes tensor product. We can denote state $|0\rangle \otimes |0\rangle$ as $|00\rangle$, state $|0\rangle \otimes |1\rangle$ as $|01\rangle$ and so forth and (1.6) becomes:

$$|\psi\rangle = a_1 a_2|00\rangle + a_1 b_2|01\rangle + b_1 a_2|10\rangle + b_1 b_2|11\rangle \tag{1.7}$$

As we see from (1.7) the state of two-qubit system is represented through four computational basis states with four complex coefficients. For convenience we can express (1.7) as:

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle + c_2|2\rangle + c_3|3\rangle \tag{1.8}$$

The state space that is constructed from two sub-spaces with tensor product is called a *Hilbert space*. The dimension of the result space equals the product of dimensions of sub-spaces. If we have a system with n qubits then its state space has dimension $N = 2^n$.

The equation (1.8) can be expanded to represent the state of n qubits:

$$|\psi\rangle = c_0|0\rangle + c_1|1\rangle + ... + c_{N-1}|N-1\rangle, \tag{1.9}$$

where the coefficients must satisfy the normalization condition:

$$|c_0|^2 + |c_1|^2 + ... + |c_{N-1}|^2 = 1 \tag{1.10}$$

One of the quantum mechanics postulates states that:

> *Associated with any isolated (closed) physical system is a complex vector space with inner product, or Hilbert space, and the state of the system is completely described by its state vector, which is a unit vector in the complex vector space.*

An element of an N-dimensional Hilbert vector space is a *column vector*, or simply vector, of N components, which are complex numbers. The inner product of two vectors $|v\rangle, |w\rangle$ in such vector space is defined as:

$$(|v\rangle, |w\rangle) = \sum_{i=0}^{N-1} v_i^* w_i, \tag{1.10a}$$

where v_i, w_i are components of vectors $|v\rangle, |w\rangle$ respectively and the * denotes the complex conjugation. For a vector $|v\rangle$ in an N-dimensional Hilbert vector space its

transposed and complex-conjugated vector, $\langle v|$, is called its *dual vector*. The dual vector $\langle v|$ is a *row vector*. With the dual vector notation the inner product of two vectors $|v\rangle, |w\rangle$ can be expressed as:

$$(|v\rangle, |w\rangle) = \langle v|.|w\rangle, \tag{1.10b}$$

where the . denotes the regular dot product. If A is an operator, sometimes it is more convenient to express $(|v\rangle, A|w\rangle)$ as $\langle v|A|w\rangle$:

$$(|v\rangle, A|w\rangle) = \langle v|.(A|w\rangle) = \langle v|A|w\rangle \tag{1.10c}$$

We also use $|w\rangle\langle v|$ notation to denote the *outer product* of two vectors $|w\rangle, |v\rangle$ and it is the result of multiplying the column vector $|w\rangle$ by the row vector $\langle v|$, or in other words the outer product of two vectors $|w\rangle, |v\rangle$ is a matrix A where $A_{ij} = w_i^* v_j$.

1.1.3 Evolution of a quantum system

According to quantum mechanics postulates the change of the state of a closed quantum system, or *its evolution*, is described by the Schrödinger equation:

$$i\hbar \frac{d|\psi(t)\rangle}{dt} = H|\psi(t)\rangle, \tag{1.11}$$

where H is a Hermitian operator and $\hbar$ is a physical constant known as Plank's constant.

In linear algebra for an arbitrary linear operator A in a Hilbert space V there exists a unique operator, $A^\dagger$, called *adjoint* or *Hermitian conjugate* of the operator A, such that for all vectors $|v\rangle, |w\rangle \in V$,

$$(|v\rangle, A|w\rangle) = (A^\dagger|v\rangle, |w\rangle) \tag{1.11a}$$

An operator A is *Hermitian* if its adjoint is also A. An operator B is called *unitary* if $BB^\dagger = B^\dagger B = I$. The operator H that describes the evolution of a closed quantum system is called the *Hamiltonian* of the system.

If H is time-independent then equation (1.11) has the solution:

$$\left|\psi(t_2 - t_1)\right\rangle = \exp\left[\frac{-iH(t_2 - t_1)}{\hbar}\right]\left|\psi(t_1)\right\rangle \tag{1.12}$$

If we denote

$$U(t_2 - t_1) = \exp\left[\frac{-iH(t_2 - t_1)}{\hbar}\right] \tag{1.13}$$

then (1.12) becomes:

$$\left|\psi(t_2 - t_1)\right\rangle = U(t_2 - t_1)\left|\psi(t_1)\right\rangle \tag{1.14}$$

If H is time-dependent then the solution to equation (1.11) can be approximated by a sequence of time-independent operators in the form (1.13). From the fact that H is Hermitian, $H = H^\dagger$, it is easy to see that U is a unitary operator, because:

$$\begin{aligned} UU^\dagger &= \exp\left[\frac{-i\,H\,(t_2 - t_1)}{\hbar}\right]\exp\left[\frac{i\,H^\dagger\,(t_2 - t_1)}{\hbar}\right] = \exp\left[\frac{-i\,H\,(t_2 - t_1)}{\hbar}\right]\exp\left[\frac{i\,H\,(t_2 - t_1)}{\hbar}\right] = I \\ U^\dagger U &= \exp\left[\frac{i\,H^\dagger\,(t_2 - t_1)}{\hbar}\right]\exp\left[\frac{-i\,H\,(t_2 - t_1)}{\hbar}\right] = \exp\left[\frac{i\,H\,(t_2 - t_1)}{\hbar}\right]\exp\left[\frac{-i\,H\,(t_2 - t_1)}{\hbar}\right] = I \end{aligned} \tag{1.15}$$

This means that the evolution of a closed quantum system is described by a unitary transformation or in other words the closed quantum system obeys *unitary evolution*. The unitary evolution implies that it is reversible and one can apply another operator to the system to get back its previous state.

1.2. Quantum gates and quantum computation

1.2.1. Quantum gates and graphical representation

The state of a quantum system changes when we apply a unitary operator to it. The operator performs some function on the state of the quantum system. In analogy to classical computation *the operator is called a quantum gate*, and the process of changing the state of the quantum system by applying the operator is called the *gate's application*. The state of the quantum system can be used to encode information, or more precisely *quantum information*. We can use a sequence of unitary operators to change the state of a quantum system in a controlled way to perform a computation. Such a process is called *quantum computation*. One of the characteristics of quantum computation is that it is *completely reversibl*e because it is performed with unitary operators.

Consider the quantum *NOT* gate for a one-qubit system. Such a system has two computational basis states: $|0\rangle$ and $|1\rangle$. Similar to the classical *NOT* gate the quantum *NOT* gate should convert the state $|0\rangle$ to the state $|1\rangle$ and vice versa. But a qubit can also be in a superposition: $a|0\rangle + b|1\rangle$ and it requires a solution for such state. It turns out that the quantum *NOT* gate acts linearly and it converts the state $a|0\rangle + b|1\rangle$ into $b|0\rangle + a|1\rangle$.

If we use the vector notation $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ to represent state $|0\rangle$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ to represent state $|1\rangle$ then the state $a|0\rangle + b|1\rangle$ is represented as $\begin{bmatrix} a \\ b \end{bmatrix}$. The quantum *NOT* gate can be represented as 2 by 2 matrix:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.16}$$

and the action of the gate on arbitrary state is simply the multiplication of the matrix to the state vector, because $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix}$.

Linear algebra theory states that any linear operator can be represented as a matrix and a matrix can be regarded as a linear operator. The matrix representation for operators is more suitable for simulating quantum computation in a classical computer. This representation also allows us to verify that quantum computation is easily reversible by applying the adjoint operator $U^\dagger$ to the resulting state $U\vec{v}$: $U^\dagger(U\vec{v}) = (U^\dagger U)\vec{v} = I\vec{v} = \vec{v}$.

One of the most popular quantum gates is the *Hadamard* gate. Its matrix is:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.17}$$

The *Hadamard* gate is often used to create a superposition for a qubit. Superpositions allow quantum computers to perform calculation in parallel as it will be introduced in section 1.3. The *Hadamard* gate is also used in the circuit that creates the entanglement of qubits as described in the section 1.2.2. It is used in the *Grover iterations* as well (see 1.3.3).

When the *Hadamard* gate is applied to a qubit in the basis state $|0\rangle$ the result state is an equal superposition: $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. If the initial state of the qubit is $|1\rangle$ then the *Hadamard* gate converts it to $|\varphi\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Applying the *Hadamard* gate again to the same qubit in such states will return it to the pure state.

It is useful to represent quantum gates graphically. A qubit is represented as a line with the direction of time goes from left to right. A quantum gate can be represented as a box, a circle, or some other graphical symbol. The following figure gives three valid representations of quantum *NOT* gate:

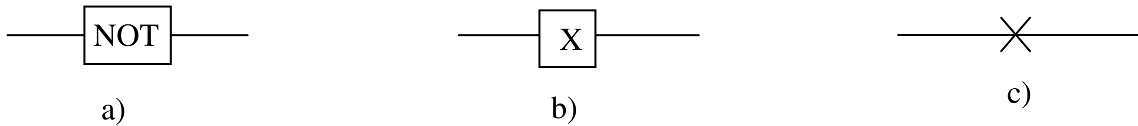
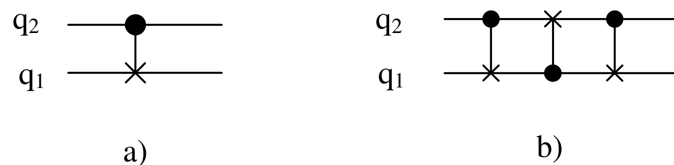


Figure 1.2: Three different representations of quantum *NOT* gate.

In this thesis we prefer the last representation.

If a quantum gate acts on two qubits its graphical representation has two qubit-lines connect to it. Figure 1.3a represent two-qubit controlled not (*CNOT)* gate, which flips the first qubit if the second qubit is in state $|1\rangle$. If the second qubit is in state $|0\rangle$ then the state of the first qubit remain unchanged. The state of the second qubit remains unchanged in both cases; therefore it is called the *control qubit,* and the first qubit is called the *target qubit*.



a) b)

Figure 1.3: a) Quantum *CNOT* gate. b) Quantum circuit that performs *SWAP* operation.

When we put together many qubits and quantum gates to perform some computation the system is called a *quantum circuit*. Figure 1.3b represents a quantum circuit and it's easy to verify that it swaps the states of qubits.

1.2.2. Entanglement, pure and mixed states

When we have a system with two qubits and their states are $|\psi_1\rangle$ and $|\psi_2\rangle$ the state of the system is $|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$. Such state is called *separable* because it's fully defined from separate states of individual qubits. But some states for two-qubit system, such as the state $|\Phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, cannot be expressed as the product of two one-qubit states. Such a state is called *non-separable* or *entangled*. State $|\Phi\rangle$ can be gotten by the circuit in figure 1.4, where *H* is the *Hadamard* gate and the input of the circuit is the basis state $|00\rangle$:

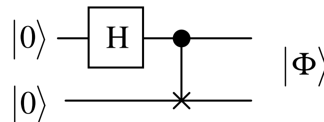


Figure 1.4: Quantum circuit to create entanglement.

When we apply the other three basis states $|01\rangle, |10\rangle, |11\rangle$ to the input of the circuit in figure 1.4 we get different entangled states. Altogether they form four states known as Bell states, which turn out to be *maximally entangled,* and they play an important part for *quantum teleportation*.

When the state of a quantum system is known and it can be represented in format (1.9) the state is called a *pure state*. Sometimes we just know that a quantum system can be in one of the states $|\psi_i\rangle$ with probability p_i then such system is said to be in a *mixed state*. The state of such a system is described by a *density operator*:

$$\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|, \tag{1.18}$$

where $|\psi_i\rangle\langle\psi_i|$ denotes the *outer product* of vector $|\psi_i\rangle$ and it's dual vector $\langle\psi_i|$. The probabilities p_i are non-negative and $\sum_i p_i = 1$. Sometimes the density operator is called the *density matrix*. Every pure state $|\psi\rangle$ has a density matrix as $|\psi\rangle\langle\psi|$. It's easy to see that the trace of the density matrix $|\psi\rangle\langle\psi|$ for a pure state $|\psi\rangle$ is 1:

$$Tr(|\psi\rangle\langle\psi|) = \sum_{i=0}^{N-1} c_i^* c_i = \sum_{i=0}^{N-1} |c_i|^2 = 1, \tag{1.18a}$$

where c_i are components of the vector.

From (1.18a) it follows that the trace of the density matrix ρ for a mixed state is also 1:

$$Tr(\rho) = \sum_i p_i Tr(|\psi_i\rangle\langle\psi_i|) = \sum_i p_i = 1 \tag{1.19}$$

1.2.3. Quantum measurement

In the process of quantum computation a quantum system doesn't interact with the outside world. At some point of the computing process we need to obtain the information inside the system. That is where the quantum measurement takes place. In the process of measurement the quantum system is no longer closed therefore it need not obey unitary evolution. The quantum measurement is described by a collection $\{M_m\}$ of *measurement operators*. The index m is the index of the outcomes. If the state of the quantum system is immediately before the measurement then the outcome m occurs with probability:

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle, \tag{1.20}$$

and the state after the measurement is:

$$\frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}} \tag{1.21}$$

The measurement operators satisfy the *completeness relation*:

$$\sum_m M_m^\dagger M_m = I\ , \tag{1.30}$$

which leads to the fact that the probabilities sum to one:

$$\langle\psi|\sum_m M_m^\dagger M_m|\psi\rangle = \sum_m \langle\psi|M_m^\dagger M_m|\psi\rangle = \sum_m p(m) = \langle\psi|I|\psi\rangle = \langle\psi|\psi\rangle = 1 \tag{1.31}$$

An important case of measurement is the measurement in orthonormal computational basis states. For a one qubit system the measurement is defined by two operators $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$. Those operators are Hermitian and $M_0^2 = M_0$, $M_1^2 = M_1$. If we measure the state $|\psi\rangle = a|0\rangle + b|1\rangle$ the probability of obtaining the outcome 0 is

$$p(0) = \langle\psi|M_0^\dagger M_0|\psi\rangle = \langle\psi|M_0^2|\psi\rangle = \langle\psi|M_0|\psi\rangle = |a|^2 \tag{1.32}$$

Similarly the probability of obtaining outcome 1 is $p(1) = |b|^2$. We can extend this case for a system of *n* qubits. Measuring the state in the equation (1.9) where $|0\rangle, |1\rangle, .., |N-1\rangle$ are orthonormal gives outcome *m* with the probability $p(m) = |c_m|^2$ and the state after measurement is $\frac{c_m}{\sqrt{|c_m|^2}}|m\rangle$. The term $\frac{c_m}{\sqrt{|c_m|^2}}$ has modulus one and it plays the role of global phase; therefore it can be dropped.

1.3. Quantum parallelism and quantum algorithm

1.3.1 Quantum parallelism

It is convenient to introduce a definition of *quantum registers* as they are usually used in literature. *A quantum register is a group of qubits that serve for a common purpose*. For example if we want to use the index of a computational basis state to represent a number then in order to represent number *9* we need *4* qubits, because *9* has binary representation *1001*. So these *4* qubits can be considered as a quantum register of four qubits. The definition of the quantum register is a loose definition, because in some cases one qubit can be used as part of one register for a part of a circuit and later be used as part of another register. This often happens to qubits in scratch registers.

As we mentioned earlier one can put together many quantum gates into a circuit to perform a computation. It can be proved that any classical Boolean function can be implemented via appropriate quantum gates. Because quantum computation is reversible; for some reversible Boolean function f there may exist a direct implementation that converts input state $|\psi\rangle$ into state $f(|\psi\rangle)$ as described in figure 1.5a, but in general we have to use a construction that is described in figure 1.5b. In the figure U_f means the implementation of the function *f*.

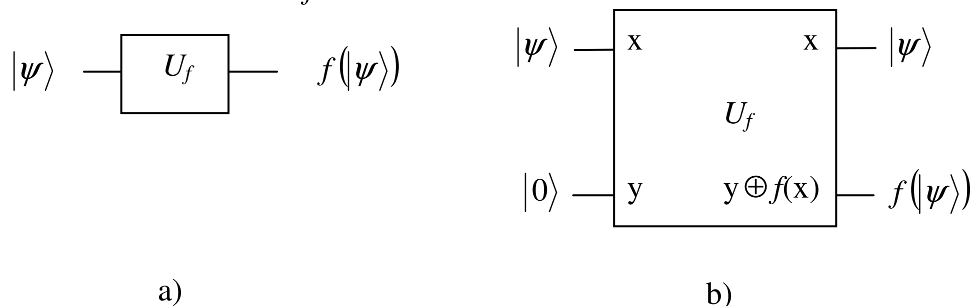


Figure 1.5: a) Direct implementation of function *f*.

b) Implementation of function *f* using additional (ancillary) qubits.

Suppose we have a circuit that calculates a function *f*. If the input to the circuit is the state $|0\rangle$ then the output state will be $f(|0\rangle)$. If the input to the circuit is the state $|1\rangle$ then the output state will be $f(|1\rangle)$. Because all quantum gates are linear operators the function *f* is a linear function. If we prepare the input state as a superposition $a|0\rangle + b|1\rangle$ then the output state will be $a*f(|0\rangle) + b*f(|1\rangle)$ where the asterisk means complex number multiplication. It is important to notice here that the circuit runs only once but it computes results for two computational states! This is *quantum parallelism*.

If we have a circuit with *n* qubits for input and *n* qubits for output to compute a function *g*, and we prepare the input state as

$$|\psi\rangle = \frac{1}{\sqrt{N}}\left(|0\rangle + |1\rangle + ... + |N-1\rangle\right) \tag{1.33}$$

then the output will be

$$g(|\psi\rangle) = \frac{1}{\sqrt{N}}\left[g(|0\rangle) + g(|1\rangle) + ... + g(|N-1\rangle)\right] \tag{1.34}$$

i.e. we have the result of the computation for all basis states in the output. Recall that $N=2^n$, the quantum circuit performs 2^n computations in just one run. The computation power of quantum computer in this case is exponential in comparison with a classical computer. But the result of the parallelism as described in the equation (1.34) is not immediately usable, because when we measure the output we just get the result for one computational basis state, say $g(|m\rangle)$, with the probability $\frac{1}{\sqrt{N}}$, and the state of the output collapses to $g(|m\rangle)$. However for some problems the state as described in the equation (1.34) can be passed to a specially designed circuit so that the information one is

interested in can be effectively extracted. That is where *quantum algorithm* comes into the place. Certain problems that are considered intractable for classical computers can be effectively solved on a quantum computer, thus are tractable with quantum computing. There is no known algorithm for factoring large integers on classical computers in polynomial time, but quantum circuits can effectively solve the problem as described in section 1.3.4.

1.3.2. Deutsh – Jozsa algorithm

In the previous section we describe a logical gate U_f that is capable of parallel computing. We don't have to know the details of the circuit; in fact we can consider it as a "black box", or an "oracle", that is capable of transforming input to output. Every application of the logic gate is treated as one query to the oracle.

Suppose we have to solve the following problem: Given a binary function $f(x) \to \{0,1\}$ with domain $\{0,1\}$, determine if *f(x)* is a constant function? In classical computing, using an oracle we have to do two queries with different values of x, and then compare the results to answer the question. Deutsch proposed an algorithm that requires only one query to the oracle to obtain the answer.

The circuit that implement Deutsch's algorithm is shown in figure 1.6.

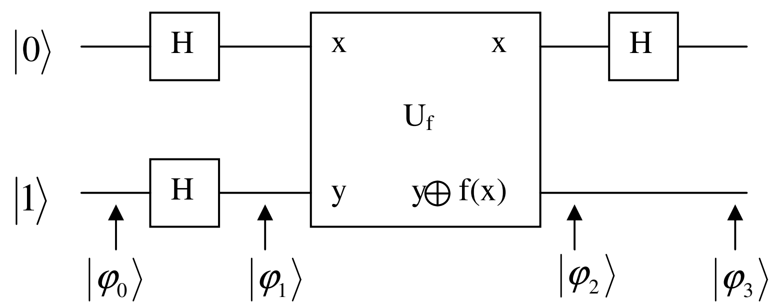


Figure 1.6. Quantum circuit for Deutch algorithm.

$$\left|\varphi_0\right\rangle = \left|01\right\rangle \tag{1.35}$$

$$\left|\varphi_1\right\rangle = \left[\frac{\left|0\right\rangle + \left|1\right\rangle}{\sqrt{2}}\right]\left[\frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}}\right] = \frac{1}{2}\left[\left|0\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right) + \left|1\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right)\right] \tag{1.36}$$

We notice that applying U_f to the state $\left|x\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right)$ gives the result $(-1)^{f(x)}\left|x\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right)$.

There are two possibilities for function f(x):

a) $f(0) = f(1)$. This implies $(-1)^{f(0)} = (-1)^{f(1)}$

b) $f(0) \neq f(1)$. This implies $-(-1)^{f(0)} = (-1)^{f(1)}$

For the case a)

$$\left|\varphi_2\right\rangle = U_f\left|\varphi_1\right\rangle = \frac{1}{2}\left[(-1)^{f(0)}\left|0\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right) + (-1)^{f(0)}\left|1\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right)\right] = (-1)^{f(0)}\frac{\left|0\right\rangle + \left|1\right\rangle}{\sqrt{2}}\frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}} \tag{1.37}$$

Applying the *Hadamard* gate to the first qubit in state $\left|\varphi_2\right\rangle$ gives us:

$$\left|\varphi_3\right\rangle = (-1)^{f(0)}\left|0\right\rangle\frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}}, \tag{1.38}$$

or the first qubit is in state $\left|0\right\rangle$ if $f(0) = f(1)$.

For the case b)

$$\left|\varphi_2\right\rangle = U_f\left|\varphi_1\right\rangle = \frac{1}{2}\left[(-1)^{f(0)}\left|0\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right) - (-1)^{f(0)}\left|1\right\rangle\left(\left|0\right\rangle - \left|1\right\rangle\right)\right] = (-1)^{f(0)}\frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}}\frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}} \tag{1.38a}$$

Applying the *Hadamard* gate to the first qubit in state $\left|\varphi_2\right\rangle$ gives us:

$$\left|\varphi_3\right\rangle = (-1)^{f(0)}\left|1\right\rangle\frac{\left|0\right\rangle - \left|1\right\rangle}{\sqrt{2}}, \tag{1.39}$$

or the first qubit is in state $\left|1\right\rangle$ if $f(0) \neq f(1)$.

If we measure the first qubit in state $|\varphi_3\rangle$ then the outcome will tell us whether the function *f(x)* is constant with certainty. Therefore using a quantum circuit we solve the problem with only one query.

1.3.3 Grover search algorithm

Consider a search for an item in an unstructured database. An example of such search is finding a phone number using a regular phone book, which lists names in alphabetical order. If the phone book has *N* names then the average number of tries is *N/2*. In 1996 Lov Grover invented a quantum search algorithm which greatly speeds up such type of search [Gro96]. The search problem can be mathematically defined with a function *f(x)* that returns $f(x) = 1$ for one value $x=x_0$ and returns $f(x) = 0$ for all other *x*. The value x_0 is called the *search-value*. As in Deuth-Jozsa algorithm the function *f(x)* is called oracle and each application of *f* is a query to the oracle. The oracle uses one qubit, $|y\rangle$, to return the result of the query and the input to the oracle is a quantum register $|x\rangle$. The qubit $|y\rangle$ is called the oracle qubit and the quantum register $|x\rangle$ is called search register. Each query performs a transformation:

$$U_f|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle \tag{1.40}$$

If we prepare the oracle qubit in the state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ then

$$U_f|x_0\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \rightarrow |x_0\rangle\frac{|1\rangle-|0\rangle}{\sqrt{2}} = (-1)^{f(x_0)}|x_0\rangle\frac{|0\rangle-|1\rangle}{\sqrt{2}} \tag{1.41}$$

And the action of the oracle can be expressed as:

$$U_f\left|x\right\rangle\frac{\left|0\right\rangle-\left|1\right\rangle}{\sqrt{2}}\rightarrow(-1)^{f(x)}\left|x\right\rangle\frac{\left|0\right\rangle-\left|1\right\rangle}{\sqrt{2}} \tag{1.42}$$

Notice that the state of oracle qubit is unchanged, therefore we say that the oracle marks the solution to the search problem. The Grover quantum search algorithm for a search space of size $N = 2^n$ is defined as the followings:

1. Initialize the search register to the state $\left|0\right\rangle$.
2. Apply the Hadamard transform to all qubits of the search register to bring it to the state $\frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}\left|x\right\rangle$.
3. Repeatedly perform a quantum subroutine, known as *Grover iteration*, *m* times, where *m* is the greatest integer less than $\frac{\pi\sqrt{N}}{4}$.
4. Measure the search register, and the probability of obtaining the search value is high.

The Grover iteration is performed in four steps:

a) Apply the oracle U_f.

b) Apply the Hadamard transform for all qubits in search register.

c) Perform a conditional phase shift for all terms except for $\left|0\right\rangle$, i.e. $\left|x\right\rangle\rightarrow-\left|x\right\rangle : x\neq 0$.

d) Apply the Hadamard for all qubits in search register.

Steps *b*, *c* and *d* together are called inversion about the average, because their combined effect is to invert the amplitude of each term about the average of all 2^n terms. Figure 1.16a gives a graphical explanation of the algorithm work for $N=8$.

It is important to notice that the Grover iteration requires the quantum implementation of the oracle, which seems to know the solution of the search problem already. The fact is the oracle just *recognizes* the solution and it is possible to do so *without knowing the exact solution* to the search problem.

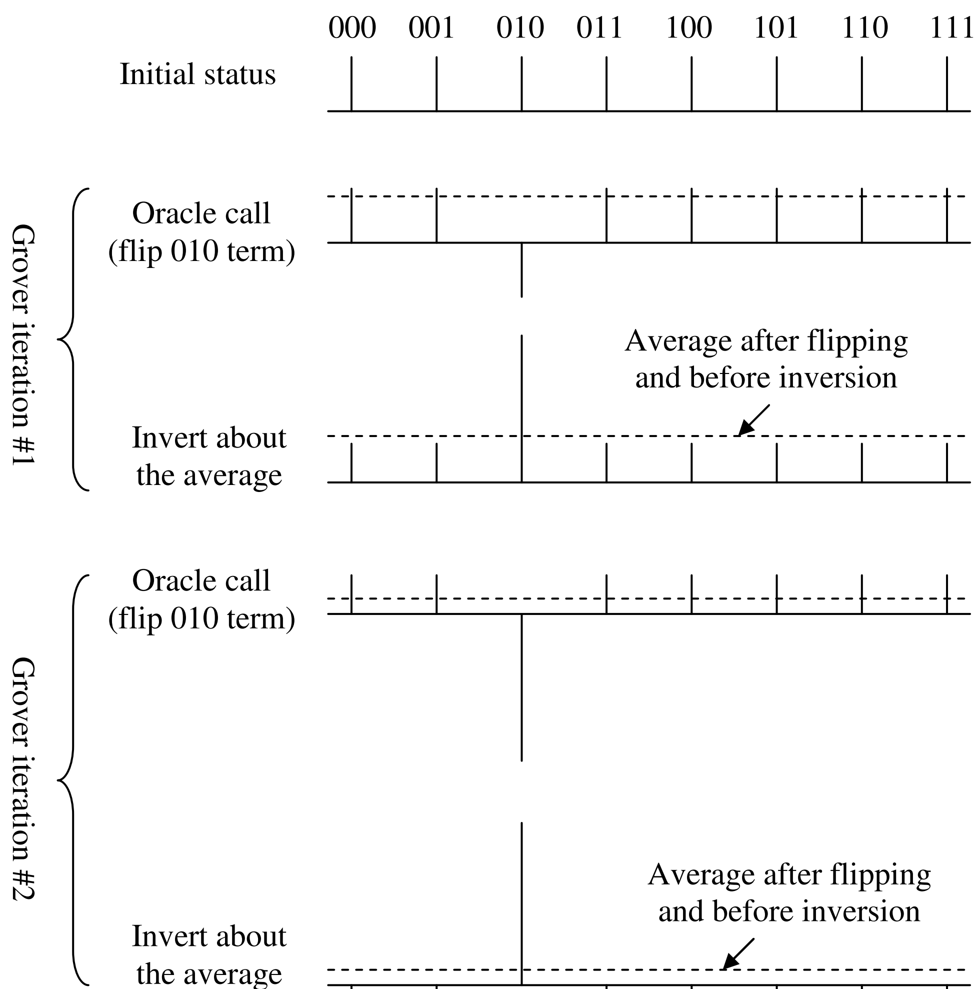


Figure 1.16a: Illustration of probability amplitudes in Grover algorithm for *N=8* (*3* qubits) and the search item is $|010\rangle$.

In the figure 1.16a the required number of iterations is *2*. The probability for the search item reaches 0.945 after two iterations.

1.3.4 Quantum Fourier transform and Shor's quantum factoring algorithm

In section 1.3.2 we described the Deutsch-Jozsa algorithm, which uses a Hadamard gate to transform input qubits into special states, make a query to the oracle, then uses another Hadamard gate to transform the result into the state where one can extract the useful information. Hadamard transform is a special case of Fourier transform. *Quantum Fourier Transform* (*QFT*) is rooted from digital Fourier transform (*DFT*), which converts vector $\vec{x}$ into vector $\vec{y}$ using the following formula:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi ijk/N} x_j \ , \tag{1.43}$$

If we know the vector $\vec{y}$ we can obtain the vector $\vec{x}$ using an inverse *DFT*, or DFT^{-1}

$$x_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{-2\pi ijk/N} y_j \ , \tag{1.44}$$

Quantum Fourier transform is defined on orthonormal basis $|0\rangle, |1\rangle, .., |N-1\rangle$ as:

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi ijk/N} |k\rangle, \tag{1.45}$$

If we use the binary representation for j as $j_0 j_1..j_{n-1}$ and adopt the binary fraction representation $0.j_0 j_1..j_{n-1} = \frac{j_0}{2} + \frac{j_1}{2^2} + .. + \frac{j_{n-1}}{2^n}$ then the quantum Fourier transform can be expressed as:

$$|j_0 j_1..j_{n-1}\rangle \rightarrow \frac{\left(|0\rangle + e^{2\pi i 0.j_{n-1}}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.j_{n-1}j_{n-2}}|1\rangle\right)...\left(|0\rangle + e^{2\pi i 0.j_{n-1}j_{n-2}..j_0}|1\rangle\right)}{\sqrt{N}} \tag{1.45a}$$

The Quantum Fourier transform can be effectively implemented using the design by Coppersmith [Cop94], as in figure 1.7. This implementation literally follows the formula (1.45a). The *conditional rotate gates*, which are represented as circles with a value inside them and a control qubit, pick up the phase for $|1\rangle$ component for appropriate qubits and contribute to the final result.

Quantum Fourier transforms can be used to solve many problem, such as to estimate the phase for an eigen state of a quantum transform, or to find the order of a constant *a* with respect to a modulo *N*. Using quantum Fourier transforms Shor developed an efficient algorithm for factoring a composite number [Sho94].

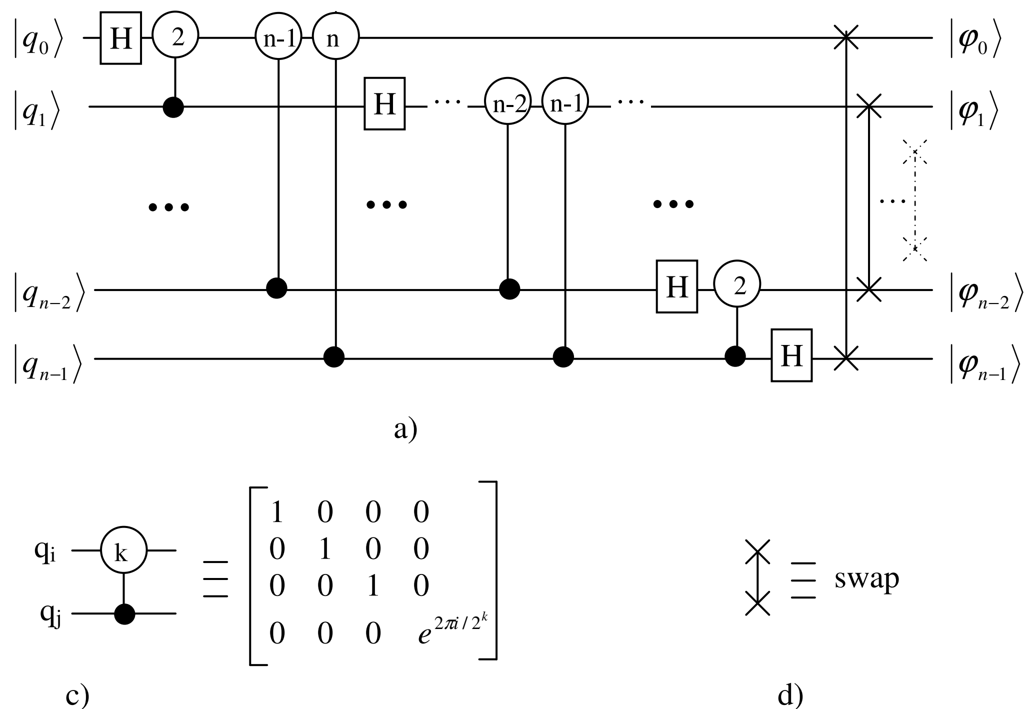


Figure 1.7. Circuit for Quantum Fourier Transform.

a) The circuit. b) Conditional rotate gate. c) Swap gate.

The circuit that implements Shor algorithm is described schematically as in the figure 1.8:

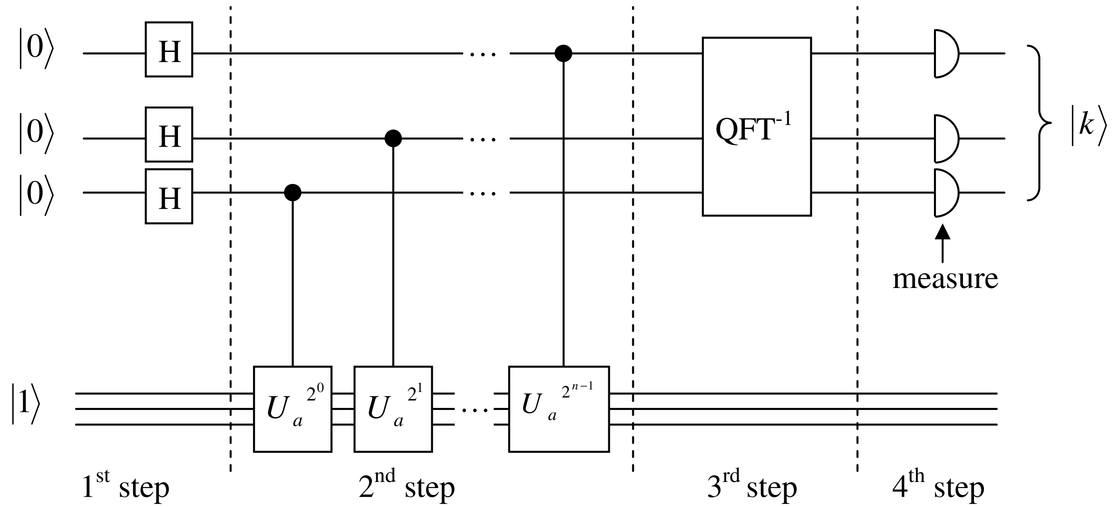


Figure 1.8. Shor quantum circuit for factoring.

The upper *m* qubits form a quantum register for the variable. The lower *n* qubits form a quantum register for the result. The size of the result register is defined by the number *N*; $n = log(N)$. The size of the variable register is at least double of the size of the result register to ensure a high probability of success. The variable register is initialized to the state $|0\rangle$ and the result register is initialized to the state $|1\rangle$. The block U_a^k can be understood as the repetition of U_a *k* times and U_a performs the following:

$$U_a|j\rangle|x\rangle = \begin{cases} |0\rangle|x\rangle, |j\rangle = |0\rangle \\ |1\rangle|ax \bmod N\rangle, |j\rangle = |1\rangle \end{cases}, \tag{1.46}$$

but one can use another implementation that performs direct calculation of $a^k x$ mod N as the following:

$$U_a^k|j\rangle|x\rangle = \begin{cases} |0\rangle|x\rangle, |j\rangle = |0\rangle \\ |1\rangle|a^k x \bmod N\rangle, |j\rangle = |1\rangle \end{cases} \tag{1.47}$$

The circuit works as the following. The first step prepares the variable register with all possible basis state to perform calculation in parallel. The second step calculates the function a^k *mod N*, which is periodic, and the result is stored in the result register. This is the most difficult part of the circuit. It was proved that $|1\rangle$ is the superposition of all eigenstates of the operator U_a; therefore the state of the result register can be treated as unchanged but the phase shift get transported into the variable register. Some articles call this construction as the *"phase kick-back"*. The third step performs inverse QFT to the variable register. Because this register gets the phase shift from the second step and the function a^k *mod N* is periodic the inverse QFT process effectively sets the register with values that contain information about the period of the function a^k *mod N*. The last step measures the variable register and gets a value that can be used to derive the period with high probability. Quantum computation ends at this step, but additional computation is required to complete the algorithm.

The index obtained after the measurement is used to compute the order of the constant *a* with respect to the modulo *N*, and the order of *a* is used to calculate the factor of *N*. This post-quantum calculation is performed classically.

The first step in the circuit uses *n* gates. The second step uses *n* blocks, each can be implemented using $O(n^2)$ gates, that requires total $O(n^3)$ gates. The third step has *n* stages, see figure 1.7a, each stage has at most *n* gates that results in $O(n^2)$ gates. The post experiment calculation can be done with running time $O(n^3)$ using continued fraction algorithm. The total running time for Shor's quantum factoring algorithm is $O(n^3)$. All known classical algorithms for factoring require exponential running time with respect to

the size n of the problem; therefore quantum factoring algorithm provides exponential speed up.

1.4. Implementations of quantum computers

In the article “The Physical Implementation of Quantum Computation”, [DiV00], Di Vincenzo describes important rules that any quantum system for computation should follow:

1. *A scalable physical system with well characterized qubits.*
2. *The ability to initialize the state of qubits to a simple basis state, such as* $|00...0\rangle$
3. *Long relevant decoherence times, much longer than gate operation time.*
4. *A “universal” set of quantum gates.*
5. *A qubit-specific measurement capability.*

Quantum systems for computation can be implemented using different techniques. There are many proposed plans and some research groups performed different experiments with one- or two-qubit gates.

Cirac and Zoller [CZ95] shows that cold ions in a radio frequency (RF) trap can serve as a quantum computer. Many ions are held within a linear array of traps to form a quantum register. Laser pulses focused on an ion can be used to perform a single qubit operation. A two-qubit operator can be implemented in this scheme as well. Monroe et al. [MMK+95] demonstrated the implementation of the CNOT gate using this technique. Along with single qubit gates the achievement shows that the technique can be used to build a quantum computer. Later, Molmer and Sorenson [MS99] proposed a technique

for entangling *n* ions with a wide laser beam. Recently Chiaverini et al. [CBL+05] report the implementation of the semi-classical quantum Fourier transform in a system of three beryllium ion qubits confined in a segmented multi-zone trap.

Another technique for quantum computer is using cavity quantum electrodynamic (QED) for trapping atoms. Pellizzani et al. analyze the effects of decoherence on a realistic model of a quantum computer based on cavity QED in [PGCZ95]. Scully and Zubairy [SZ02] present a scheme for the implementation of the discrete quantum Fourier transform using cavity quantum electrodynamics. In the proposed scheme a series of atoms with atomic coherence carries the input state passes through a series of cavities and classical field. The result of those transformations is that the state in the cavities is the quantum Fourier transform of the input state.

The technique that achieves the most experimental results in quantum computing is the one that uses liquid state nuclear magnetic resonance (NMR). Cory, Fahm and Havel [CFH97] present a new computational model, which uses macroscopic ensembles of quantum spins. Each molecule acts as an independent quantum computer (NMR computers). Qubits in such quantum computer are the spins of atomic nuclei. All molecules are placed in a static magnetic field. The spins in every molecule can be directly manipulated via radio-frequency electromagnetic pulses. Two qubit quantum gates like a CNOT gate can be implemented using the proposed schemes in [CPH98] or [GC97]. The NMR computers can be electronically programmed. Vandersypen et al. report the realization of quantum order finding algorithm with NMR computers of five atoms in [VSB+00]. The order of a number a with respect to a modulus N is the smallest positive integer x such that $a^x \; mod \; N = 1$ where a and N are coprime. In 2001 they

reported the realization of Shor's quantum factoring algorithm with NMR computers of seven atoms [VSB+01].

Chapter 2

Simulation of quantum computers on a classical computer

2.1 Quantum simulators – assisting tools for quantum research

Implementation of a quantum system for computation requires many expensive devices and special conditions. As an alternative many types of software were developed to assist research in quantum computation.

Raedt and Michielsen give an overview of methods for simulating quantum computers with the review of many quantum simulation software and packages in [RM04]. They divide quantum simulators into five groups. Below we outline the main features for those groups.

A. Programming language and libraries for quantum computers: they are not a complete product for us to design and test quantum circuits.

B. Quantum compilers, which take input as a unitary transformation and translate it into sequence of one- and two-qubit gates that performs the same action. As author of [BBC95] proved any multiple-qubit gate can be realized as a sequence of one- and two-qubit gates, quantum compilers help us to generate such sequence. With current technology the implementation of multiple-qubit gate is difficult, yet these tools can be useful in laboratory.

C. Quantum circuit simulators: this group contains all complete applications allowing simulations of quantum circuits on the gate level. A gate-level quantum

simulator is considered general-purpose if it can simulate any valid quantum operator. At the same time it should not accept invalid, or non-unitary, operators because an isolated quantum system follows unitary evolution. Some quantum simulators such as [NMI03] restrict the size of gates to be simulated to two qubits, with one qubit being a control bit. In [VRMH03] the authors built their quantum simulator with new data structure that allows them to utilize many existing libraries from binary decision-making research. Their simulator is optimized in storage for repeating structures. They reported that for Grover's search problem their simulator outperforms all others. The main drawback of this simulator is that it cannot simulate Quantum Fourier Transform. In [RAK+04] the authors inform about web-based service that extends the concept of simulator for parallel computer to networked computers to overcome the limit of 20 qubits for stand alone PC. The simulator can handle circuits up to 30 qubits. For a restricted gate set it can investigate circuits with up to 60 qubits.

D. Software that uses time-dependent Hamiltonians to implement unitary transformations of the qubits. The expression of such Hamiltonians contains coefficients that depend on time. This type of software emulates the ideal quantum computers with details of physical implementations of quantum computer hardware.

E. Pedagogical software packages.

The group C draws great attention from scientists, researchers and students. Existing applications and packages have many shortcomings, such as: they require powerful computers, require an additional computation package such as Matlab or Mathematica or

they don't provide an easy method for interacting between classical and quantum parts of an algorithm. The graphical user interface in those applications poses a difficulty for creating complicated circuits with a thousand gates. We designed an effective quantum simulator to overcome those shortcomings and provide many more features.

2.2 Important aspects of general purpose quantum simulators

In order to simulate a quantum system for computation a simulator must simulate the state of the quantum system, quantum operators, and evolution of the system quantum operators along with the quantum measurement. As we mentioned earlier the state of a quantum system with n qubits is fully defined by a complex vector of 2^n elements. An array of 2^n complex numbers can be used to store the state vector. Any quantum operator, or quantum gate, is a linear operator and it can be represented as a matrix. If we consider the quantum system as single entity then any quantum operator for the system is represented as a 2^n by 2^n matrix. The evolution of the quantum system under a quantum operator can be simulated by multiplying a matrix by the state vector. If the state vector of the quantum system before the evolution is $\vec{v}$ and the quantum operator is described by the matrix A then the new state vector after the evolution is $A\vec{v}$. Simulation of the quantum measurement will be described later.

For multiple qubit system it is possible to apply a quantum operator for separate qubits; therefore the dimension of the matrix depends on the operator and one must specify which qubits the operator acts on. For example we can apply a *NOT* gate to the first qubit in a quantum system. In general the number of qubits, n, in the quantum system for computation is much larger than the number of qubits, m, that are involved in quantum

gates. The matrix for an *m*-qubits gate has dimension 2^m by 2^m and one cannot directly multiply such matrix to the state vector with dimension 2^n. Different simulators resolve this issue in different ways. We shall come back to this issue in the section 3.2.

For convenience I use the following definitions in this thesis:

Definition 1: The *size* of a quantum gate is the number of involved qubits in the quantum operator. By analogy with classical gates each involved qubit is called a pin; each pin of a gate is connected to a qubit in the quantum system

Definition 2: A c*ircuit* is an ordered set of qubits along with ordered list of gates.

Since the qubits have order, we can use indices to label them. We also use indices to label pins of the gate. In this thesis the indices for qubits in the system as well as the indices for pins in a gate start with *0*. The number of gates in a circuit is called *circuit depth*.

Definition 3: *Computation* is a process of computing the new state of the system, from the initial state to the final state, by application of gates in the order specified in the circuit.

The input to computation is encoded in the initial state of the system. The result of computation is encoded in its output state or in a result of a measurement. It's important to notice that the circuit must accept all possible input states and produce appropriate output states, but the computation succeeds, i.e. the output satisfies the purpose of computation, only for certain input states. We could not find a simulator that allows users to determine if a computation succeeds; the simulator we developed has such feature.

Chapter 3

Building An Effective General-Purpose Quantum Simulator

Our goal is building a general-purpose quantum simulator that effectively simulates many generic 64-qubit circuits with many features that help researchers and scientists to design and analyze quantum circuits using small computers.

3.1 Representing a quantum state vector in a classical computer

The most difficult issue for all quantum simulators is the space required to represent the state vector, which is exponential of the number of qubits in the system. The computation precision also plays some role. The complex number that is used to represent the probability amplitude for each basis computational state has two floating-point numbers, one for real and one for imaginary parts. The single precision floating-point number cannot provide high accuracy. If we use single precision variable to store value $x = \left(1 / \sqrt{199} \right)^2$ then adding *x 199* times gives us $1 + 1.67 * 10^{-6}$, or the error is $1.67 * 10^{-6}$.

For a state vector with 2^{20} components the error would be intolerant. Therefore for serious designs and computations we have to use floating point numbers with double precision.

With double precision floating-point numbers the above sum is $1 + 2.9 * 10^{-15}$ and the double precision is the most accurate representation in all regular computers. For

calculations with higher accuracy one must use some special library and it will consume too much of the memory in the computers. We consider the accuracy for double precision floating-point numbers is sufficient and it provides good balance between precision and memory usage. A complex number with double precision requires *16* bytes. In a 32-bit operating system the addressable space is 2^{32} bytes (*4* GB) one cannot fully store state vector for *28* qubits. Many existing quantum simulators store the probability amplitudes in a contiguous memory array and the index of elements in the array is the index of computational state vectors; for small computers they can simulate up to 20-qubit generic circuits. This straight forward approach cannot simulate a circuit with *28* qubits. Some simulators that run in parallel computing systems report that they can simulate circuits with up-to *30* or *31* qubits. *In order to simulate a 64-qubit circuit we have to use different approach.*

We notice that if all qubits are in an *independent superposition* then the state vector for n qubits has 2^n components. This rarely happens for many circuits. The definition of the independent superposition will be introduced shortly. A qubit in a state of a quantum system is in a superposition if its value can be either 0 or 1 when one measures the state. In order to determine that one must prepare the same state multiple times and measure the qubit every time.

Consider two-qubit quantum systems. Suppose the state of such a system is represented in binary coefficients as $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |01\rangle)$ then the qubit whose value is represented in the left position is not in superposition while the other qubit is. The final state of the

quantum system in the figure 1.4 is $|\Phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ where both qubits are in its superposition but the state vector has only two components. This is due to the entanglement, or we can say that there is a dependency between the two qubits. A superposition of a qubit that is gotten with a single-qubit gate is *independent*.

If a superposition of a qubit is gotten only after an application of a two-qubit gate and the other involved qubit must be in superposition then their superposition is *dependent*. A similar concept can be applied for the gates with more than two qubits. In general for a system with n qubits that are in a dependent superposition the system state vector doesn't have 2^n components. In contrast if the n qubits are in an independent superposition then the state vector always has 2^n components.

In many popular quantum circuits the superposition of qubits is dependent. For example the circuit for the modular exponentiation function described in [4] with k-qubit operand uses $7k+1$ qubits, but only $2k$ qubits are used for variable of the function. When the circuit is used in Shor's factoring algorithm those $2k$ qubits are put in the independent superposition at the beginning and maximum number of components in the state vector never exceeds 2^{2k}.

The quantum circuit for addition in the *QFT* state, [Dra00], uses $2n$ qubits with only n qubits are in independent superposition and the maximum number of components in the state vector never exceeds 2^n.

The fact that a quantum computation must begin with a computational basis state for which the state vector has only one component and end with a measure of relevant qubits, which collapses superposition of those qubits, thus reduces the number of components in

the state vector, also supports the observation that quantum state vector is sparse most of the time.

However some implementations of the Grover circuit for quantum searching use *n* qubits and all of them are in an independent superposition and the state vector has 2^n components most of the time. Our model still works with this type of circuits, but it could not provide the speedup it has with other types of circuits.

Figure 3 represents a quantum state vector with many empty components as white squares.

Index 0 1 2 3 4 5 6 7 8 9 . . . N
Amplitude □ □ ■ □ □ ■ □ □ □ ■ □ □ □ □ □ □

Figure 3.1. A Quantum State Vector with three non-empty elements
The top line is the indices of computational basis states

The fact that the state vector of a quantum system is sparse most of the time suggests us to store only non-empty elements in a data structure, which can be called *sparse vector*, to save the memory so that we can simulate more qubits. In this paper we use term *'index m'* to specify *'the computational basis state with index m'* and the probability amplitudes are called *'amplitudes'*. Each element of a sparse vector must contain information of both *the index* and *the amplitude* in order to reconstruct the original state vector when it is required. The quantum state vector in figure 1 can be stored in a sparse vector as $<2,a_2>,<5,a_5>,<9,a_9>$ where a_2, a_5, a_9 are complex numbers. The actual number of elements stored in the sparse vector is called *vector size*.

The idea of using a sparse vector to store the vector state of quantum system is not new: Matlab package uses sparse matrices to describe quantum gates, but it has to expand the

sparse matrices into full format for evaluation of the gates, as described in [Hsu02]. The main requirement for a sparse vector is the ability to get the amplitude for any index. If the index is not found among its actual elements, the amplitude for the index is 0. I denote the action of getting amplitude for index *m* as *GetAmplitude(m)*.

3.2 Application of a gate

The second difficult for a quantum simulator is calculating the new state after application of an *m*-qubit gate. The straight-forward method to apply an *m*-qubit gate on an *n*-qubits system is extending gate's matrix from size 2^m by 2^m into 2^n by 2^n matrix and then multiplying the new matrix to the state vector. This requires too much memory and current personal computers don't have enough memory to simulate a 20-qubits system with that method. Therefore all simulators try to avoid using of 2^n by 2^n matrix to calculate the new state vector after the gate action. Different authors describe their method of gate's application slightly differently but the technique can be specified as the following:

a. *m*-qubit gate has $2^m x2^m$ matrix and it can multiply with vectors of dimension 2^m.

b. *m*-qubit gate can be apply to any *m* qubits. If those qubits are not the lowest qubits, or they are not in the order specified by the gate's matrix, we can pick 2^m components from appropriate positions in the system state vector to form a temporary vector, multiply gate's matrix by new vector, then put the components of the result vector back into the positions where they were picked up.

c. Repeat step b) 2^{n-m} times to cover all components of the system state vector.

We use a specialized data structure with different method to apply a quantum gate to the state vector that is explained in the section 3.3.1.

3.3 Solution for building an effective quantum simulator

I develop a data structure with a new gate application technique that allows us to effectively simulate many 64-qubit circuits. As I mentioned earlier, using a sparse vector will save the memory for the system state vector. One can combine the pair <*index, amplitude*> into a data structure, or a node, and put nodes into one of many standard searchable data structures, such as *AVL* tree, red-black tree, or standard template library (*STL*) map … to form a sparse vector. A node is identified by its index. But these standard data structures are not ready to improve simulation speed, as I shall prove later.

3.3.1 Gate application technique

To apply an m-qubit gate on an n-qubit system where $n > m$ we use a technique that is different from the one described in section 3.2. We observe that if m involved qubits are the lowest qubits and they are in the suitable order for the gate, the components of the temporary vectors are adjacent components in the system state vector, and all we have to do is split the system state vector into sub-vectors and multiply the gate's matrix with each of them to get the result. The new state vector for the system is the concatenation of resulting sub-vectors. Our gate application is described as the following:

> *In order to apply an m-qubit gate to m qubits that are not in the lowest positions in a circuit or they are in different order than the pin arrangement in the gate we artificially swap those qubits to the lowest positions and in the required order,*

apply the gate to m lowest qubits by direct multiplication of the gate's matrix to the state sub-vectors of size 2^m to form new states sub-vectors, combine them into a temporary state vector and then swap those qubits in the temporary state vector back to their original positions.

This swapping happens only in simulator, and it is not using a real SWAP quantum gate. The main advantage of this technique is that it retrieves and stores components of the system state vector with incremental indices in many actions. *The swapping qubits back and forth accesses the system state vector with non-incremental indices*, as we'll show later, but we found a method to do that with incremental indices. The splitting of the state vector into state sub-vectors, the combining of the state sub-vectors into the state vector and the matrix multiplications are performed with incremental indices.

3.3.2 Special data structure for simulation with swapping technique: *QSV*

We developed a special data structure: *quantum sparse vector* (*QSV*), which is a sparse vector with a *constant sequential access time*. Recall that a sparse vector stores only non-empty elements; in this case it means only the amplitudes that are different from zero will be stored. We use terms *non-empty* and *non-zero* interchangeably. The sequence of accesses is sequential if the indices of elements appeared in the sequence strictly increase. Figure 3.2 shows such a sequence.

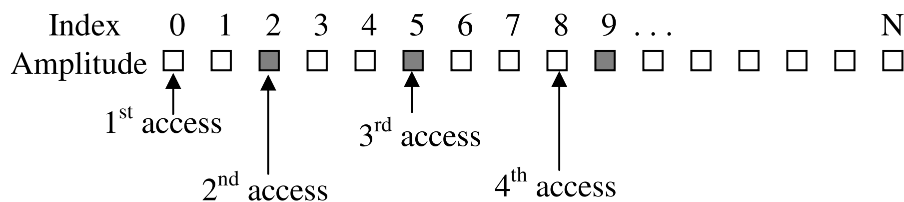


Figure 3.2: Sample of sequential access to a quantum sparse vector.

With the notation described in sub-section 3.1, the sequence in figure 3.2 is described as: *GetAmplitude(0), GetAmplitude(2), GetAmplitude(5), GetAmplitude(8).* The sequence *GetAmplitude(1), GetAmplitude(3), GetAmplitude(2)* is not sequential. We implement a function to perform the dot product of two *QSV* vectors, *DotProduct()*, therefore the matrix of an m-qubit gate can be implemented as an array of 2^m *QSV* vectors; each vector represents a row in the matrix; and the multiplication of the gate's matrix to a temporary vector is a sequence of 2^m dot products. The *DotProduct()* function for two *QSV* vectors with sizes u and v where $u \le v$ uses at most $u+v$ memory accesses and it performs at most u multiplications for only non-empty elements. The measure operator with re-normalization can be directly implemented with *QSV* vectors.

As we mentioned above, the only non-sequential action with system state vector in our simulator is the swapping of qubits back and forth before and after matrix multiplications. Let assume that we want to swap two qubits j and k with $j<k$. In such operation the index i of all elements in the *QSV* state vector is changed, the values of jth and k^{th} bits in binary representation of i are swapped. Let binary representation of index i is $i_1 i_2..i_j..i_k..i_n$. For indices with $i_j=0$ and $i_k=1$ their values after swapping will be larger. On other hand for those indices with $i_j=1$ and $i_k=0$ their values after swapping will be smaller. Therefore two indices u and v where $u < v$ may become $u > v$ after swapping. This means that while we can sequentially get components from the old *QSV* state vector and calculate new indices for them we cannot sequentially store them, one-by-one, to new *QSV* state vector in the direct manner. We developed a method to do this with exactly $2t$ read and $2t$ write operations for the *QSV* state vector with t elements, which will be explained in the next section.

We implement our quantum sparse vector with a single link list and many supporting functions. The data stored in the list has two items: *index* and *amplitude*, as described in sub section 3.1. *The sequential access time for both non-empty and empty elements of the system state vector is constant*. Each *QSV* vector has a position pointer that points to the current stored element in the list. One important function is the function to get the next stored element in a *QSV* vector, *GetNextElement(),* that returns both the index and the amplitude for the element and the position pointer is updated to point to the next element in the *QSV* vector. If the pointer is at the beginning of the *QSV* vector *GetNextElement()* returns its first element. If the last element in the *QSV* vector is already gotten, or the *QSV* vector is empty, *GetNextElement()* returns a special empty element; otherwise it returns a non-empty element. *GetNextElement()* is used frequently in many operations such as performing dot product of two *QSV* vectors, or measuring the state of group of qubits, or constructing temporary vectors for matrix multiplications. In process of simulating a gate we get and discard components from the current *QSV* vector and insert only non-zero amplitudes after calculation into the new QSV vector. Both *QSV* vectors are accessed sequentially.

3.3.3 Algorithm to perform swapping two qubits with *QSV*

Let consider the swapping two qubit j and k with $j<k$ example for a *QSV* vector q as mentioned above. Let $(j,k)_i = i_j i_k$ where i_j and i_k are value of j-th and k-th bits in the binary representation for the i; $(j,k)_i$ has 4 possible values: $00, 01, 10, 11$. If the value for $(j,k)_i$ is 00 or 11 then the index i after swapping remains unchanged. If two indices u and v with $u<v$ and $(j,k)_u = (j,k)_v = 01$ (or $(j,k)_u = (j,k)_v = 10$) then their value u' and v' after swapping are different but they satisfy the same relation $u'<v'$. We know that in a *QSV*

vector all indices are sorted. Therefore if we get all elements from *q* one by one their indices will be increasing. When we calculate new indices for swapping we store new elements into three different *QSV* vectors *q1, q2, q3* for three different values of the <i_j, i_k> pair: *01*, *10* and the rest (*00* or *11*). Because we retrieve elements from q with increasing indices, the indices for every vector *q1, q2,* and *q3* are also increasing; that satisfies the access rule for a *QSV* vector. After all elements from q are processed we will combine elements from *q1, q2, q3* into new *QSV* vector *q'* using a method that is similar to three-way merge-sort. The complete swapping algorithm is:

1. Splitting phase.
   a. Set current position to the beginning for *QSV* vector *q*.
   b. Get current element from *q*, let *i* be its index and *a* be its amplitude. Set current position to next element in *q*.
   c. Calculate new index *i'* from *i* by swapping values for *j*-th and *k*-th bits in the binary representation.
   d. If $((j,k)_i = 00)$ or $((j,k)_i = 11)$ insert the element (*i', a*) into *q1*; if $((j,k)_i = 01)$ insert the element (*i', a*) into *q2*; otherwise insert the element (*i', a*) into *q3*.
   e. Repeat steps b), c) and d) until all elements in *q* are processed.
2. Combining phase.
   a. Set current position to the beginning for *QSV* vectors *q1, q2, q3*.
   b. Get three elements at the current position from *q1, q2, q3*. After getting an element the current position is moved to next element in the *QSV* vector.
   c. Compare three (or less, see notes below) indices and pick the element with smallest index to insert into *q'*.

d. If an element is inserted into *q'* its place will be replaced by the next element from the same *QSV* vector and the current position for the vector is updated.

e. Repeat step c), d) until all elements in *QSV* vectors *q1, q2, q3* are processed.

Notes: The care must be taken to cover cases where there are only two *QSV* vectors for *q1, q2, q3*, or elements from one *QSV* vector are completely processed before others. If there is only one *QSV* vector for *q1, q2, q3* the combining phase is dropped, the *QSV* vector is the result.

In this algorithm all elements in *q* get one read and one write access in each phase. The read and write operations are usually combined into a full memory access; thus the algorithm uses at most $2*t$ full memory accesses where *t* is the size of *QSV* vector *q*.

3.3.4 Improve simulation time by processing only non-empty sub-vectors

Any standard structure for sparse vectors saves the memory, but the *QSV* helps us to save simulation time as well. In order to calculate new state vector for *n*-qubit system after application of an *m*-qubit gate the technique described in the section 3.2 uses 2^{n-m} matrix multiplications and each of them uses 2^m accesses to the state vector. The total number of accesses to the state vector is 2^n. With *GetNextElement()* function described in sub section 3.2 we can identify and perform the matrix multiplication only for non-empty sub-vectors in the system state vector. This significantly reduces simulation time. Recall that our simulator uses swapping qubits and direct matrix multiplications on state sub-vectors technique to calculate the new state vector. For an *m*-qubit gate the state sub-vector size is 2^m and the total number of state sub-vectors is 2^{n-m}. An element with index *idx* in the *QSV* state vector is mapped into an index *j* for the *QSV* state sub-vector and an

inner index *iidx* inside the *QSV* state sub-vector. The following figure describes the mapping between *idx, j* and *iidx* for *m=2*.

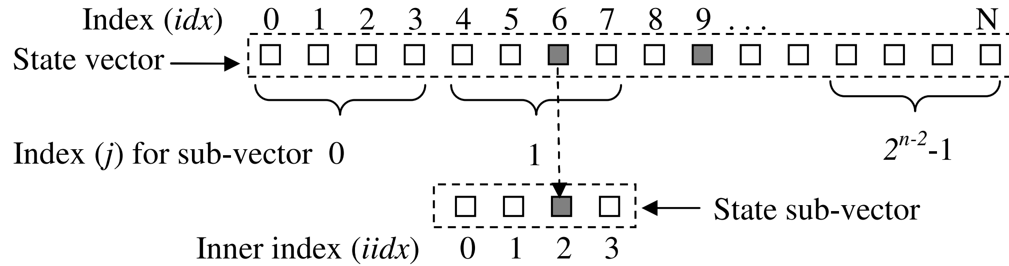


Figure 3.3: the mapping between *idx, j* and *iidx* for *m=2*. For *idx=6* the values for *j* and *iidx* are *1* and *2*, respectively. Knowing *j* and *iidx* one can calculate *idx*.

As we explained earlier the state vector is not always full therefore the number of state sub-vectors that have non-zero elements, or non-empty state sub-vectors, are much less than 2^{n-m}. The index for the state sub-vectors runs from *0* to 2^{n-m}-1. Let *q* be the *QSV* state vector after operating qubits are swapped into the lowest positions in the required order and *C* be the matrix for the gate in the array of *QSV* vectors format. Let *q.GetNextElement()* be the action of calling the *GetNextElement()* function for the *QSV* vector *q* to get an element. The algorithm to calculate the new state vector with multiplications only with non-empty state sub-vectors is described below:

1. Set position pointer for *QSV* vector *q* to the beginning. Create new *QSV* vector *se*, which is a state sub-vector. Create new *QSV* vector *q'*, which is new state vector after matrix multiplications.
2. Let *e = q.GetNextElement().*
3. If *e* is empty then stop; the *QSV* vector *q'* is the result.

4. Let *idx* be the index and *amp* be the amplitude for *e*. Calculate the index *j* for the state sub-vector that contains index *idx*. Calculate the inner index *iidx* for the element *e* in the state sub-vector. Delete old elements in *se* and reset its pointer to the beginning. Insert an element with index *iidx* and amplitude *amp* into *QSV* vector *se*.
5. Let *e2 = q.GetNextElement().* If *e2* is empty element then go to step8.
6. Let *idx2* be the index and *amp2* be the amplitude for *e2*.
7. Calculate the index *j2* for the state sub-vector that contains index *idx2*. If *j2 = j* then calculate the inner index *iidx2* for the element *e2* in the state sub-vector, insert an element with index *iidx2* and amplitude *amp2* into the *QSV* vector *se*, and go back to step 5); otherwise go to step below.
8. Multiply the gate's matrix *C* to the *QSV* state vector *se* to form new *QSV* vector *se'*.
9. If *se'* is empty go to step12 otherwise set position pointer for *se'* to the beginning and go to step below.
10. Set *e3 = se'.GetNextElement()* .
11. If e3 is non-empty let *iidx3* be its index and *amp3* be its amplitude, use *iidx3* and *j* (index for the *QSV* state sub-vector *se*) to calculate new index *idx3* for the element in new state vector *q*', insert an element with index *idx3* and amplitude *amp3* to *QSV* vector *q'*; otherwise delete *se'* and go to step below.
12. Set *e = e2* and go back to step3.

With the above technique and data structure we implemented a general purpose quantum simulator and called it qsim.

3.4 Calculating average simulation time for *qsim*

In this section we estimate the simulation time for our simulator. Although our simulator can directly simulate a gate with *10* qubits, we restrict the estimation to circuits with popular elementary gates, which have size of *1*, *2* and *3*; therefore the average gate size is *2*. The estimation for circuits with highly variable gate sizes would not be accurate, because the matrices for them have different actual elements. For *2*-qubit gates the simulator has to swap twice or less. As we proved earlier the swapping of the qubits takes linear time. Let the average size of the state vector of a circuit be *k*, then swapping its qubits back and fourth for a gate takes *2*(2*k)*a* time, where *a* is the access time of a component in the *QSV* state. Now consider the normal matrix multiplication $C\vec{u} = \vec{v}$:

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ \cdots & \cdots & \cdots & \cdots \\ c_{m1} & c_{m2} & \cdots & c_{mm} \end{bmatrix} * \begin{bmatrix} u_1 \\ u_2 \\ \cdots \\ u_m \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ v_m \end{bmatrix} \tag{3.1}$$

we see that $v_j = \sum_i c_{ji} u_i$ and the element u_i appears in all multiplications of the form $c_{ji} u_i$, or in other words the element u_i of the vector $\vec{u}$ is multiplied by all elements from *i-th* column of the matrix *C*. In our simulator each element of the state vector belongs to only one state sub-vector and each state sub-vector is multiplied only once in a gate application. Therefore in a gate application an element of a state vector is multiplied by at most *4* elements in a column of the gate's matrix. The number of actual multiplications is less than *4* if the column has less than *4* non-zero elements, because the *DotProduct()* function only multiplies non-zero elements. Thus the multiplication time for a gate is at most *4k* (*m* + *a*) where *m* is the time for multiplication of two complex numbers. The

time for calculating new state vectors for a gate application is at most $4k*a + 4k(m + a)$, or $4k(2a + m)$. m and a are constants therefore the simulation time is of order $k*d$ where d is the circuit's depth. In reality it takes more time than that because of the overhead of memory allocation and de-allocation for elements in the *QSV* vectors. *It is important to emphasize here that the simulation time is linear with respect to state vector size*. The average state vector for many circuits with n qubits is far less than 2^n. The Shor quantum circuit for factorization using construction described in [VBA95] uses $7k+1$ qubits, but the largest state vector size is 2^{2k} and not 2^{7k+1}. With improved construction the circuit uses $4n+2$ qubits and the largest state vector size remains 2^{2k}. As we mentioned in section 3.1 the quantum circuit for addition in the *QFT* state uses $2n$ qubits and the maximum number of components in the state vector never exceeds 2^n.

Memory is the only factor that limits the number of qubits in a circuit that our simulator can simulate. With the occupied memory for the operating system and the simulator itself, our simulator is able to simulate many circuits with *64* qubits with the state vector size up to 2^{24}. The simulator has a great advantage for circuits with many qubits and small actual size of the system state vectors.

3.5 Simulating measurement operations

3.5.1 Measurement with output only

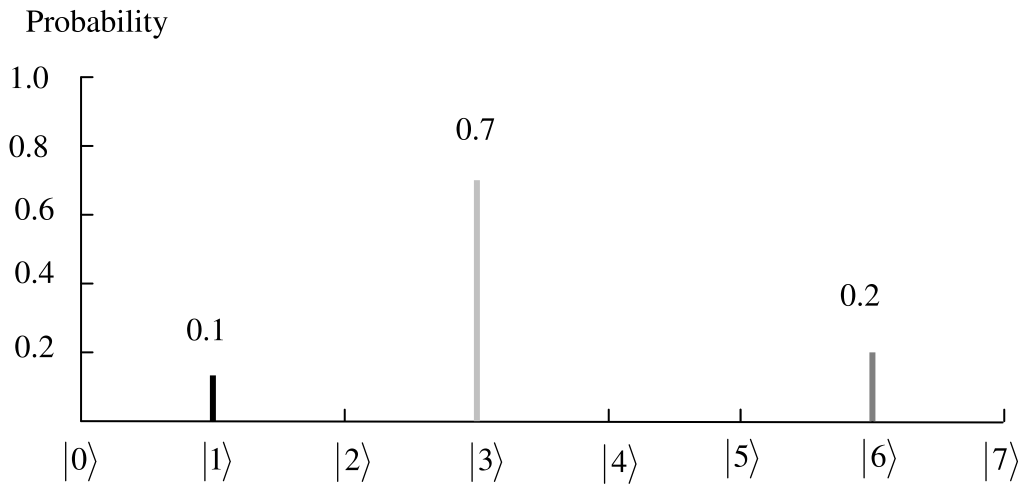
Recall that quantum measurement is explained in the section 1.2.3, and it shows that measuring the state in the equation (1.9) where $|0\rangle, |1\rangle, .., |N-1\rangle$ are orthonormal gives outcome m for the computational basis state $|m\rangle$ with the probability $p(m) = |c_m|^2$ and the state after measurement is $|m\rangle$ with some global phase. Our simulator assumes that the

computational basis states are orthonormal so that the measurement can be simulated. With such a condition the process of measurement in our simulator is performed as the following:

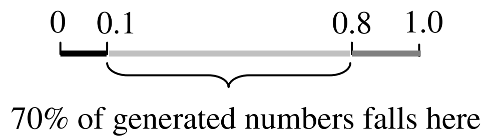
1. A random number r, $0 \le r < 1$, is generated.
2. Starting with the first element in the *QSV* state vector, the probabilities are added until they are greater than the random number. Because the probabilities sum to one this step always ends.
3. The index for the last added probability is the result of the measurement.
4. The state vector collapses to the outcome state and it gets re-normalized.

The above process is sometimes called *full measurement* and it can be easily understood with the following example. Assume we measure a quantum system with *3* qubits in the state $|\psi\rangle = \sqrt{0.1}|1\rangle + \sqrt{0.7}|3\rangle + \sqrt{0.2}|6\rangle$, so the probabilities for the states $|1\rangle$, $|3\rangle$ and $|6\rangle$ are 10%, 70% and 20% respectively. If we lineup intervals that represent those probabilities one after another we'll have an interval of unit length. Figure 3.4 represents such lineup graphically. As we can see from figure 3.4b the state $|3\rangle$ gets 70% of the generated numbers.

When we just want to measure the state vector to obtain some basis state and the outcome doesn't have the effect to the circuit structure after the point where the measurement is performed we call the measure "*output only*". Most of the simulators we know provide this kind of measurement only. In our simulator the result of the measurement is written to a file so that other programs can read it for their calculation. The measurement is often implemented as the *MEASURE* gate. Some quantum circuits use measurements and the



a) Probability distribution for computational states



70% of generated numbers falls here

b) Probability distribution for random number generator

Figure 3.4: Graphical explanation for measurement process in our simulator

outcomes are used to control the behavior of some gates after the *MEASURE* gate. We call such a measurement "*output with feedback*" and it will be described in next sub-section.

It is possible for a quantum system with multiple qubits to measure the state for only some qubits. In our simulator such measurement is allowed for qubits that have adjacent indices so that one can specify starting and ending indices, *startBit* and *endBit,* for them and it is performed as the following:

1. Perform steps 1), 2) and 3) as described in the full measurement.

2. Extract the value for the bit positions [*startBit*, *endBit*] from the result. Let v be the value.
3. Iterate the *QSV* state vector and accumulate probabilities for all elements with indices for which value for the bits in the group [*startBit*, *endBit*] is v. Let p be the accumulated probabilities.
4. Iterate the *QSV* state vector again and remove all elements with indices for which value for the bits in the group [*startBit*, *endBit*] is not v and the amplitudes for the remaining elements are adjusted by the factor $\frac{1}{\sqrt{p}}$. The result of the measurement, v, is written to an output file as in the case of the full measurement.

We can measure the state of a single qubit by specifying its index for both *startBit* and *endBit* for the measurement.

3.5.2. Measurement with output and feedback

There are two quantum circuits that use measurement with output and feedback that are described in [PP00] and [Bea02]. We shall analyze them with more details in section 4.6. Using measurements with feedback significantly reduces the number of qubit usage for the *QFT* and thus extends the scope for experiments with few qubits. None of the quantum simulators we know of provide measurements with feedback.

In our simulator the measurement with output and feedback is implemented via *INTERNAL_MEASURE* gate. The parameters for such gates include *startBit*, *endBit* and *storageIndex*. The *storageIndex* is the index of an internal storage where the result will be stored for later calculation in the simulating process. The result from the measurement is

written to an output file as in regular measurements. In addition to that the value is stored in an internal storage that will be used via a special directive that will be covered later.

Qsim has many built-in gates so that users can build many useful circuits, such as Shor's factoring, Grover's search, Draper's addition or semi-classical circuit for factoring. It also has many directives to help designers to debug their circuit, to study the circuit behavior or to produce reports or educational materials.

3.6. Main features of *qsim*

1. It can simulate many 64-qubit circuits and it runs very fast.
2. It accepts a circuit file from the command-line, or it prompts for the file if one is not specified. The circuit file is a text file with certain rules that are fully specified in the Appendix A. The main advantage of the text circuit file is that it can be generated by some script that is more error-proof than the circuit that is created by hand using a graphical interface. All gates and directives to the simulator are enclosed within a {} pair. A comment begins with # and continues until the end of line.
3. It implements many popular gates such as *NOT, CNOT, TOFFOLI, HADAMARD, FREDKIN…*
4. It has special gates {CTRL_ROTATE:n} and {INV_CTRL_ROTATE:n} that are frequently used in quantum Fourier transform (*QFT*) and inverse *QFT*.
5. It allows user to define custom gates with a unitary matrix. A gate for m qubits requires 2^{2m} complex coefficients for the matrix. The maximum number of pins for a user-defined gate is 10, which requires more than 1 millions coefficients for its matrix, and one should be very careful in preparing such matrix. The simulator will

check if the matrix is unitary before it accepts the gate. I could allow more qubits for such gates, but the matrix would be too huge, and it is not correspondent with current technology.

6. It has a {MEASURE} gate to measure a group of adjacent qubits, each measurement writes the result in a separate file for further processing.
7. The input state of the system can be specified in three ways: as value 0 or 1 for individual qubits, an appropriate value for a group of qubits, or as a superposition state with unit total probabilities. The first two formats, {INPUT *bit; value*}, {INPUT *start-bit, end-bit; value*}, are supported by the requirement that states a quantum computing system must be able to initialize qubits in a computational basis state as described in [DiV00]. The last format, {INITIAL_STATE *index, amplitude, index, amplitudes*, …}, is valid because we proved that any quantum system that can implement an arbitrary one qubit gate along with the *CNOT* gate can be put in an arbitrary superposition.
8. It has {OUTPUT *bit; value*}, {OUTPUT *start-bit, end-bit; value*} directives to specify final state of computation with expected value for each qubit or qubit groups. This allows the simulator to check if the circuit works as designed at the end of simulation.
9. It has many command-line options. One of the options specifies the input data file for a circuit that contains the initial and the final states in one line for each pair, so that we can check the correctness of a circuit with the full range of input states automatically.

10. It has {RANDOM qubit} and {TWO_BODY_INTERACTION qubit1, qubit2} gates to implement pseudo-random quantum circuit that creates a random state for group of qubits in as described in [EWS+03].
11. It has a {PARTIAL_TRACE qubit} directive to get the partial trace for a qubit, which is very useful for analyzing the randomness and entanglement for quantum states that are created by a gates sequence.
12. It has {INTERNAL_MEASURE} and {INTERNAL_CONTROL} gates that allow designers to create and test semi-classical Shor factoring circuits that use much fewer qubits than pure quantum circuits for factoring.
13. It has many useful directives such as {DUMP}, {HISTOGRAM}, {ENTROPY} … to help designers to get useful information from quantum circuits for analysis. Full list of available gates and directives is described in Appendix A.

Chapter 4

Designing and Analyzing Quantum Circuits with *qsim*

In quantum circuit design the circuits are not optimized in the first several attempts. Scientists often try to implement circuits with plenty of qubits to verify their functionality and then optimize them to get less qubits and gates. Our quantum simulator is the perfect tool for that purpose. In [VBA95] V. Vedral, A. Barenco, and A. Ekert developed basic quantum circuits for addition, modular addition, controlled modular multiplication and modular exponentiation that are sufficient to implement Shor's algorithm. With our simulator we built basic quantum circuits after the construction that is described in [VBA95]. Standard circuit for the modular exponentiation in the article uses $7k+1$ qubits to implement circuit for k-qubit operand. Using {INPUT} and {OUTPUT} directives along with *-i* command-line options as specified in the Appendix A we are able to verify the correctness of the circuit. It also proved that our simulator works well.

4.1. Concept of designing quantum circuits with classical registers

In [VBA95] authors specified that one can reduce the number of qubit usage in the modular exponentiation circuit by using classical registers to keep the values for the constant a and the modulus N. We found that there is very little literature for such implementation. In our opinions it's more convenient to name this concept as "design circuits with hard-coded values" as we shall explain shortly.

To better describe our approach we extend some definitions:

a. *A composite quantum gate* is a small circuit that uses one or more quantum registers to perform a function or a task. Sometimes we drop the adjective “quantum” and just say a composite gate.

b. A composite gate doesn’t need an explicit unitary matrix for its action. The purpose of the gate is specified by some text; usually its purpose is previously defined in the same article.

c. A composite gate has a name and it can be represented by a rectangle, or block, with many connected qubits or quantum registers. Each qubit or quantum register may or may not have a name in the layout of the block. If a register doesn’t have a name then its position decides its purpose.

d. If the context allows we will drop the term “composite” and just call the gate with its name.

e. A gate or circuit is *universal* if it performs the same function or task for all valid input states that are applied to its inputs. We will give the samples of universal composite gates shortly.

Using the concept of composite gates makes the description of the circuits easier. A circuit is built from gates to perform some computation. If the circuit is used in another circuit as a building block it becomes a composite gate with a name. With such definition at any time there is only one circuit and other blocks are gates.

In figure 4.1a shows the expanded version of the quantum circuit for addition as described in [VBA95]. The figure 4.1b represents the block version of the circuit and it can be called as composite gate for addition, or *ADDER* gate. The *ADDER* gate is

universal, its second register always has value *a+b* on output for any valid values *a* and *b* on inputs. It is important that the graphical represent of a composite gate shows all quantum registers and qubits it uses. This will reduce the confusion in the designing process, otherwise the designer has to always remember that there is a hidden qubit or register in the representation. When a composite gate is used in a circuit all of its inputs and outputs must be connected, except for the ones at the beginning or at the and of the whole circuit.

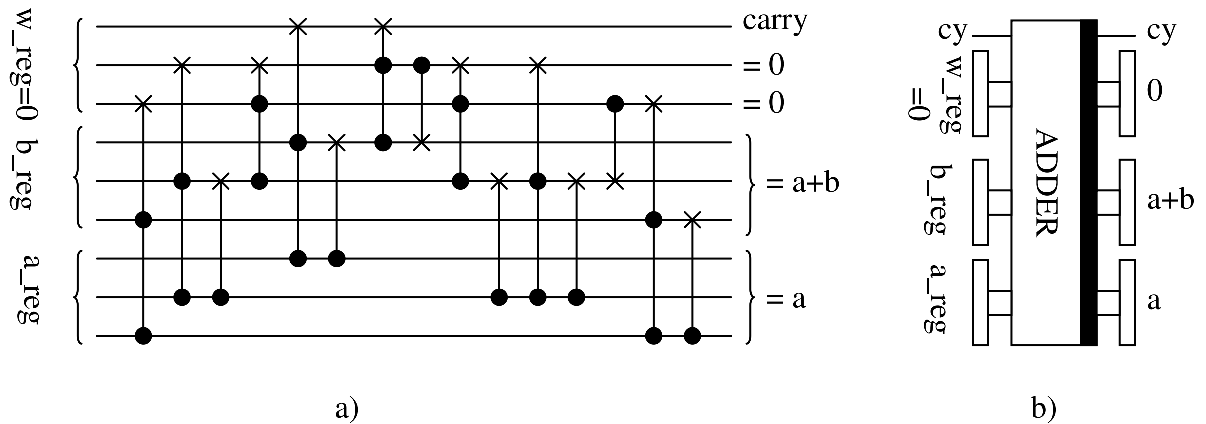


Figure 4.1 a) Circuit for addition two numbers using a working register

b) The graphical representation of the composite gate for the circuit.

If the *ADDER* gate we just described above is used repeatedly but in all of its appearances the register *reg_a* is used to keep the same number, say *5*, then we can hard-code the number *5* into the composite gate itself. The new composite gate adds number *5* to the content of the register *reg_b* and we call it *ADD_5*.

We notice that in figure 4.1a each qubits in register *reg_a* is used as control bit in some gates. The number *5* has binary representation as *101*; first and third qubits have value *1*,

and the second qubit has value *0*. Therefore all gates that have second qubit from register *reg_a* as a control bit will not make any action. On the other hand all gates that have either first or third qubit from register *reg_a* as a control bit will depend only on other control bits. Thus we can drop all gates that connect to the second qubit from register *reg_a*, and replace all gates that connect to first or third qubit from register *reg_a* by simpler gates with less or zero control bits, e.g. *TOFFOLI* gates are replaced by *CNOT* gates and *CNOT* gates are replaced by *NOT* gates. The circuit for *ADD_5* is represented in figure 4.2 and its block representation is figure 4.2b. Let emphasize here that there is no register, both quantum and classical, is used to keep the value *5*. The value *5* is hard-coded inside the gate.

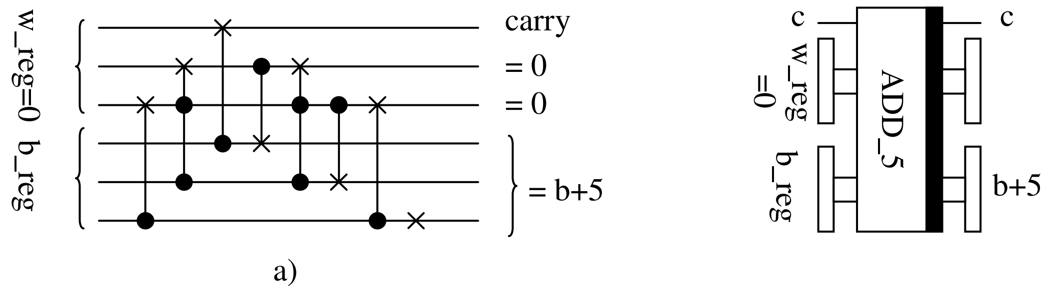


Figure 4.2 Quantum circuit to add number *5* to a register a) and its block representation b)

With hard-coded version for *ADDER* and *ADDER_MODULO* we are able to implement the improved version for modular exponentiation circuit that uses *5k+1* qubits. We use the circuit as composite gate to build a quantum circuit for Shor factoring algorithm with *5k+1* qubits and successfully simulate it with our simulator. Reducing the number of qubits increases the complexity of gates. The hard-coded modular exponentiation

quantum composite gate uses generalized *TOFFOLI* gates with *3* or *4* control qubits along with regular *TOFFOLI* gates.

4.2. Optimal Reversible Quantum Circuit for Multiplication

In this section we describe the design of new quantum circuit for multiplication that is optimal in term of qubit usage.

Quantum circuits for addition, modular addition, controlled modular multiplication and modular exponentiation are described in [VBA95]. There is no effective quantum circuit design in current literature for generic multiplication operation and we try to implement that one.

In general product of two n-bit numbers, a and b, is a 2n-bit number. If we denote the binary form of a as $a_0a_1 \ldots a_{n-1}$ then the product $a*b$ can be represented as:

$$a * b = a_0 * (2^0 * b) + a_1 * (2^1 * b) + \ldots + a_{n-1} * (2^{n-1} * b)$$

In classical circuit design each term $a_j*(2^j*b)$ is called partial product and the multiplication circuit can be built from addition circuits. The product register of size 2n bits is set to 0 at beginning. At j^{th} step partial product $a_j*(2^j*d)$ is added to the product. According to rules for quantum circuit designs that are described in [3a] the circuits must be reversible and have no loop. From that point of view the multiplication circuit for computing the product of two n-qubit numbers requires at least $4n$ qubits, because we have to preserve both input values and store the product in a separate register. It's easy to see that the plain adder circuit that is described in article [VBA95] is not optimal in term of qubit usage, because it uses $n-1$ extra qubits for temporary carries, therefore we look for another circuit for addition.

4.2.1 Addition using Quantum Fourier Transform

In article [Dra00] T. Draper developed a circuit for addition that utilizes a property of Quantum Fourier Transform (*QFT*) to avoid using extra qubits for temporary carries. The circuit to perform *QFT* is described in figure 1.7. It is interesting to notice that in [Dra00] author denotes the state of first qubit in *n*-qubit register after QFT_n by $|\varphi_{n-1}(a)\rangle$, and the state of nth qubit in *n*-qubit register after QFT_n by $|\varphi_0(a)\rangle$; this creates some confusion in applying the circuit for a particular design. We reorganize gates so that we can use the notation $|\varphi_0(a)\rangle$ as state for the first qubit after QFT_n.

Figure 4.4 represents the modified version. This circuit can be considered as a composite gate, *Fourier Addition* or *FA*. The *FA* circuit has a very interesting feature: *its gates are completely independent, and they can appear in any order*. The *FA* gate is reversible; we can un-compute the result register to get back value *b*. The inverse *FA* gate, or subtraction, is not the one with reversing gates' order in addition gate, because it will produce same result. Rather, the inverse *FA* gate uses controlled rotation gates with opposite, or negative, phase.

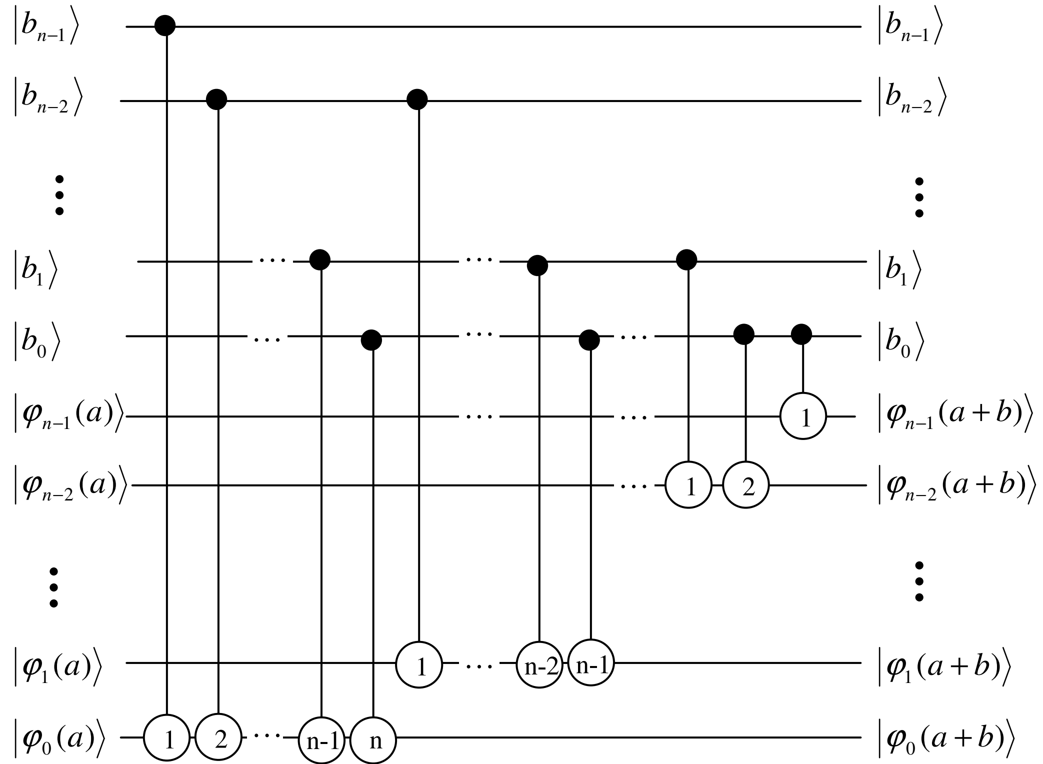


Figure 4.4. Modified circuit for addition in QFT state, or Fourier Addition (FA)

4.2.2 Quantum circuit for multiplication using addition in QFT state

Since *FA* gate requires exactly *2n* qubits for addition of two *n*-qubit numbers, we can use them to built optimal quantum circuit for multiplication. The original *FA* gate can be modified to become controlled *FA* gate, by adding a control bit to every controlled rotation gate in figure 4.4. But such controlled *FA* gates are not ready to use in multiplication circuit that follows the classical circuit design. The result from multiplication of two n-qubit numbers has *2n* qubits and regular controlled *FA* gate that operates with QFT_{2n} states would require two registers of size *2n* qubits along with a control qubit and it adds two *2n*-qubit numbers. In multiplication circuit one of the numbers is the product, which has *2n* bits, but the other number is partial product, which

has only n meaningful bits and other n bits are *0*. Direct application of *FA* gate uses n extra qubits just to hold *0* values in them.

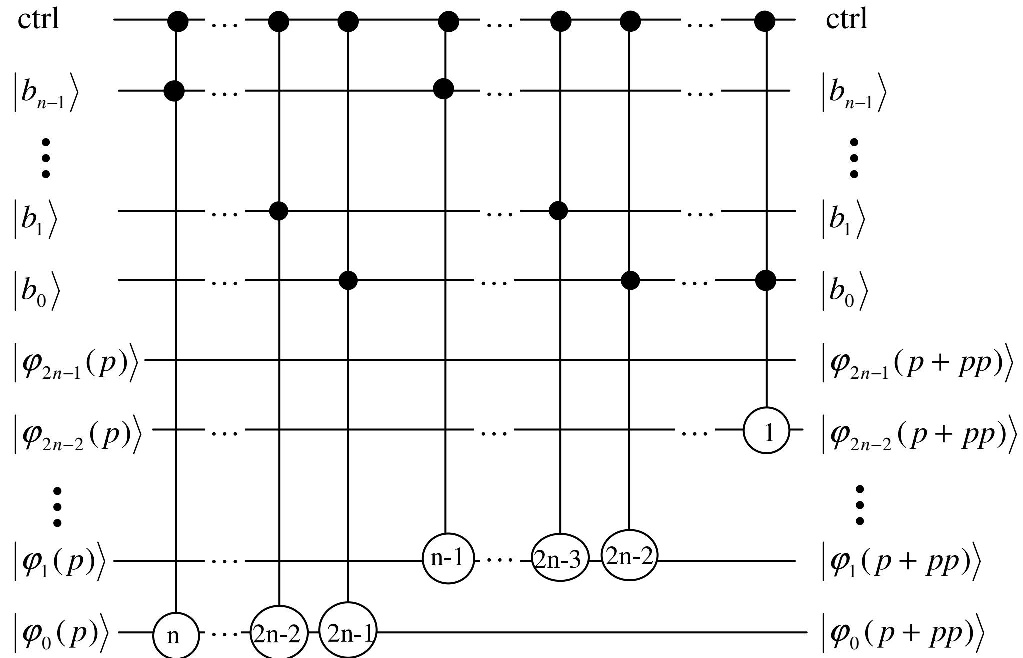


Figure 4.5 Customized FA gate for second step in multiplication circuit. p denotes current content of product register and pp denotes the partial product at this step

We found the way to overcome this by designing customized *FA* circuit for each step which adds n-bit number, b, to appropriate portion of product register if the correspondent bit in number a for the step is *1*. One of such circuit, circuit for adding partial product to the product at second step, *FOURIER_ADDER_1*, is described in figure 4.5

The complete circuit for multiplication is presented in figure 4.6. In the figure *FOURIER_ADDER_j* is the partial product addition circuit for j^{th} step. In this design same number b acts as partial product in every step. The output of the circuit is the Quantum Fourier Transform (QFT_{2n}) of the product $a*b$.

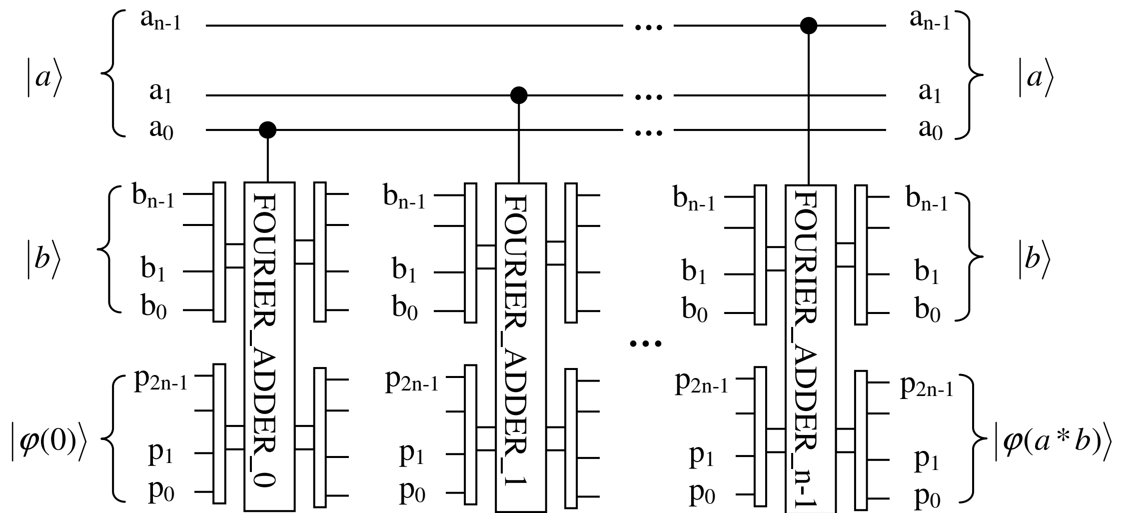


Figure 4.6 Quantum circuit for multiplication using controlled *Fourier Adder* gates. $|\varphi(x)\rangle$ denotes $QFT_{2n}(x)$

It's easy to see that the circuit is universal, with the meaning that it can compute product of any two n-bit numbers. This circuit uses the output of a *QFT* circuit as its input: $|\varphi(0)\rangle$ and the *QFT* circuit in figure 1.7 is universal therefore there is no restriction for the result register. All *FOURIER_ADDER_j* gates that are similar to the one that is represented in figure 4.5 also use no hard-coded values. The qubits in the registers *a* and *b* are used as control qubits only therefore they carry no restriction. The complete usage, or end-to-end circuit, needs a *QFT* composite gate and inverse *QFT* composite gate to perform quantum Fourier transform for the result register. We can prepare input for registers *a* and *b* with any values that they can hold and get the product out of the end-to-end circuit.

The circuit is reversible and we can un-compute the result to get back zero bits in product register. Like *FA* gate, the inverse circuit for multiplication is not a simple

reversing of gates in the forward circuit. The inverse circuit for multiplication uses inverse *FA* gates to achieve the result.

Our circuit uses exactly *4n* qubits to compute product of two *n*-qubit numbers therefore it is optimal in term of qubit usage. The most complex gates in our design are *3*-qubit controlled-controlled rotation gates as shown in figure 4.5.

As we mentioned earlier all gates in the *FA* circuit are independent, they can be re-grouped to allow parallel execution, as the author of [Dra00] points out. Our design preserves this feature. We hope this design will encourage scientists to use multiplication in their design.

4.3. Preparing arbitrary superposition for a quantum system

It is clear that any quantum system for computation must be able to prepare any one of its basis computational states. Output state of a quantum circuit can serve as input state for other circuits. It is not required that such system must be able to prepare its state in arbitrary superposition, but it is useful if we can prepare arbitrary superposition for a quantum system so that it can serve as input for other computation. In this section we show that one can prepare arbitrary superposition for n-qubit system using only one- and two-qubit gates.

4.3.1. Preparing a superposition for 1-qubit system

A superposition for one qubit of the form $|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle$ can be achieved by applying the gate U to state $|\psi\rangle = |0\rangle$ where

$$U = \begin{bmatrix} \alpha & x \\ \beta & y \end{bmatrix} \tag{4.1}$$

if there exists an unitary matrix U of the form (4.1). Because coefficients α and β satisfy $|\alpha|^2 + |\beta|^2 = 1$; therefore all we have to do here is to find x and y that make matrix U unitary. We know that all matrices of the form

$$\begin{bmatrix} e^{iu}\cos(\theta) & -e^{iu}\sin(\theta) \\ e^{iv}\sin(\theta) & e^{iv}\cos(\theta) \end{bmatrix} \tag{4.2}$$

where u, v, and θ are real numbers are unitary.

The equation system

$$\begin{cases} e^{iu}\cos(\theta) = \alpha \\ e^{iv}\sin(\theta) = \beta \end{cases} \tag{4.3}$$

where $|\alpha|^2 + |\beta|^2 = 1$ always have solution and it is easy to solve (4.3). One possible solution is:

$$\begin{cases} u = \arctan(\alpha.imag()/\alpha.real()) \\ v = \arctan(\beta.imag()/\beta.real()) \\ \theta = \arccos(\alpha.real()/\cos(u)) \end{cases} \tag{4.4}$$

if none of the denominators are zero. Let $x = -e^{iu}\sin(\theta)$ and $y = e^{iv}\cos(\theta)$ where *u, v*, and θ are a solution to the equation system (4.3) and we have the unitary matrix for the U gate.

4.3.2 Preparing arbitrary superposition for n-qubit

For n-qubit quantum system we have 2^n computational basis states, $|0\rangle$, $|1\rangle$,…, $|N\rangle$, and general state of the system is of the form $|\varphi\rangle = a_0|0\rangle + a_1|1\rangle + ... + a_N|N\rangle$, $N = 2^n-1$, where are a_i satisfy

$$|a_0|^2 + |a_1|^2 + ... + |a_N|^2 = 1 \tag{4.5}$$

In many cases some of coefficients a_i are zeros. Without loss of generality we can write the state $|\varphi\rangle$ as

$$|\varphi\rangle = c_1|b_1\rangle + c_2|b_2\rangle + ... + c_m|b_m\rangle, \tag{4.6}$$

where all c_m are non-zero and $b_1 < b_2 < \ldots < b_m$.

If $m=1$ $|\varphi\rangle$ is a basis state, this is trivial case. Let us consider non-trivial case where $m > 1$. In some articles, such as [3b], indices b_i are called planes. In the article [Cyb01] Cybenco shows that one can apply one-qubit gate to any pair of planes. Here we outline the method with some modification to fit our design:

Cybenco circuit for applying 1-qubit operator V on two arbitrary planes i and j, where i < j:

1. Express i and j in binary format, $B(i)$ and $B(j)$, and find the bit position, say k, that has value 0 in $B(i)$ and value 1 in $B(j)$. Such a position exists because i is different from j and $i < j$
2. Apply *CNOT* gate with k is control bit to all qubits that have different value in $B(i)$ and $B(j)$, except *k-th* qubit.
3. Apply *NOT* gate to all qubits that are 0 in $B(i)$ except *k-th* qubit.

4. Apply controlled-V operator on *k-th* qubit with all other qubits as control bits.

5. Do the same as in step 3

6. Do the same as in step 2.

7. The result is the application of operator V on planes i and j

Figure 4.7 shows a sample of the Cybenco circuit in which planes with indices *1* and *12* are the target of V gate.

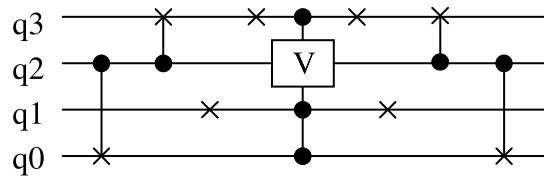


Figure 4.7. Cybenco circuit for application the V gate on the planes #1 and #12.

Their binary representations are *0001* and *1100,* respectively.

We use the Cybenco circuit described above to build our circuit that produces state $|\varphi\rangle$ as the following:

1. Initialize the system in basis state $|b_1\rangle$.

2. Set i=1. Set $S = \{c_i, c_{i+1}, \ldots, c_m\}$. If S has only two elements then go to step 7.

3. Let

$$\begin{cases} r_{i-1} = \sqrt{|c_i|^2 + |c_{i+1}|^2 + \ldots + |c_m|^2} \\ r_i = \sqrt{|c_{i+1}|^2 + |c_{i+2}|^2 + \ldots + |c_m|^2} \end{cases} \tag{4.7}$$

Create unitary operator V_i with matrix $V_i = \begin{bmatrix} \frac{c_i}{r_{i-1}} & x_i \\ \frac{r_i}{r_{i-1}} & y_i \end{bmatrix}$ where x_i and y_i are derived from $\frac{c_i}{r_{i-1}}$ and $\frac{r_i}{r_{i-1}}$ as in the case of 1-qubit system.

4. Apply operator V_i to planes b_i and b_{i+1} using Cybenco construction.
5. Set $S = S \setminus \{c_i\}$. Set $i = i+1$. If S has more than one element then go back to step 3.
6. Perform steps 3, 4, and 5 with an exception that $V_i = V_{m-1} = \begin{bmatrix} \frac{c_{m-1}}{r_{m-1}} & x_i \\ \frac{c_m}{r_{m-1}} & y_i \end{bmatrix}$ this time.
7. The system state is now $|\varphi\rangle$

The circuit in our design consists of *m-1* stages. Each stage acts on exactly two planes. After each stage one more basis state from the equation (4.6) is added to the result with the required coefficient, except for the last stage, when two remaining basis states are added at the same time.

In our construction the most difficult gate is controlled-V_i gate (see figure 4.8a)

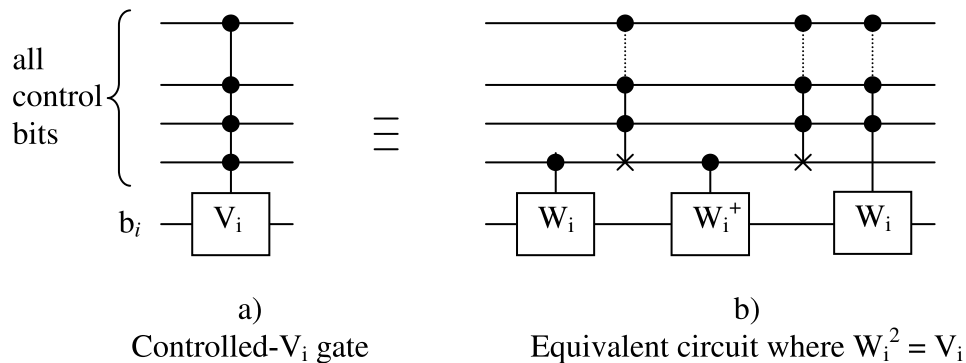


Figure 4.8: Controlled-V_i gate and the equivalent circuit

In [BBV+95] authors proved that multiple-controlled gate can be downsized as described in figure 4.8b. The downsizing process is repeated until all gates have only two qubits, and single-controlled gate can be implemented as circuit of 1-qubit and *CNOT* gates. Therefore our construction can be implemented with only 1-qubit and *CNOT* gates. *This result is important for quantum circuit simulators, because it allows them to accept any superposition as input state for computation*. Our simulator has a special directive to specify a superposition as the input for a circuit: {INITIAL_STATE}. If a designer wants the computation to begin with a certain superposition he can use the directive to specify it. Full specification for the directive is given in the Appendix A.

4.4. Quantum circuit to generate random state for computation

Random numbers are very important for both classical information theory and the theory of quantum computation. Random unitary matrices are studied with many details in [ZK94] and [PZK98]. For quantum communication theory random numbers play crucial role. Random unitary operators are used for the superdense coding of arbitrary quantum states [HHL04]. With such a superdense coding the classical communication cost (in bits) for remote state preparation reduces [BHL+05]. Random unitary operators also increase the efficiency for data-hiding schemes [HLS04].

In [PZK98] authors give explicit method of constructing a $N \times N$ uniform random matrix via composed ensemble of random unitary matrices (*CUE*). For quantum computation $N=2^n$ where n is the number of qubits in the quantum random circuit. The direct application of the method for quantum random matrices requires huge number of elementary quantum gates (with one and two qubits) on the order of $n^2 2^{2n}$ along with 2^{2n} independent random parameters. The exponential resource requirement makes such

construction impractical. In [EWS+03] the authors introduce a quantum circuit that creates a pseudo-random state that requires only polynomial resource with respect to the number of qubits. The construction of the circuit is described in figure 4.9. The random quantum states created by such circuit is biased with respect to the states from uniform random operators, therefore the authors call the circuit pseudo-random. However the measure over pseudo-random quantum states converges exponentially to the uniform measure on *U(N)*.

In our simulator we implemented two quantum gates that allow us to build the pseudo-random unitary operator, or pseudo-random quantum circuit, as described in [EWS+03]. They are *RANDOM_U2* and *TWO_BODY_INTERACTION*.

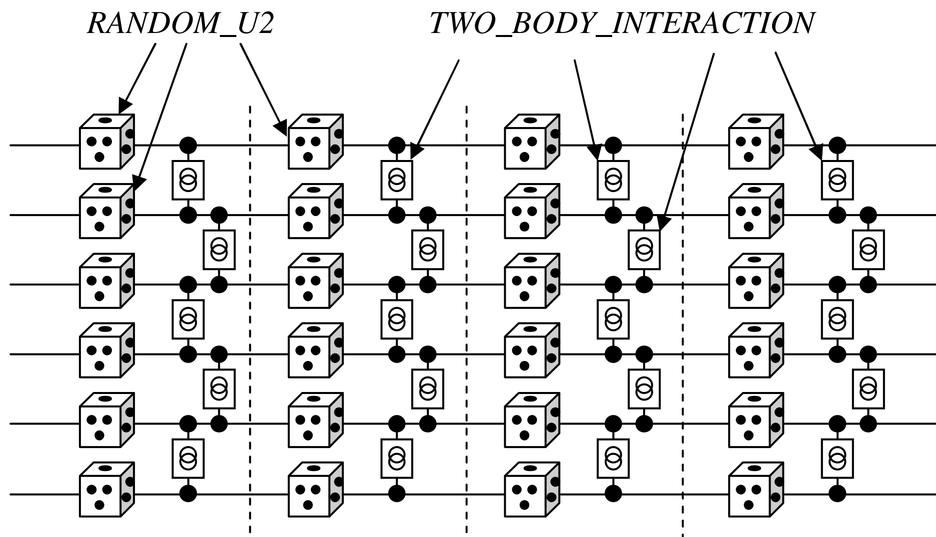


Figure 4.9. Quantum circuit to generate pseudo-random state with four identical columns. More columns create better randomness.

The matrix for *RANDOM_U2* gate is generated using the construction for circular unitary ensemble (*CUE*) as described in [PZK98].

$$RANDOM_U2 = \begin{bmatrix} \cos\varphi\, e^{i\psi} & \sin\varphi\, e^{i\chi} \\ -\sin\varphi\, e^{-i\chi} & \cos\varphi\, e^{-i\psi} \end{bmatrix}, \tag{4.8}$$

where ψ, χ are uniformly taken from the intervals $0 \le \psi < 2\pi$, $0 \le \chi < 2\pi$, and φ is defined as:

$$\varphi = \arcsin(\theta^{1/2}), \tag{4.8a}$$

with θ is uniformly taken from the interval $0 \le \theta < 1$. Each *RANDOM_U2* gate has its own matrix.

The role of *TWO_BODY_INTERACTION* gates is to create the entanglement between adjacent qubits. They share the same matrix:

$$TWO_BODY_INTERACTION = \begin{bmatrix} e^{i\pi/4} & 0 & 0 & 0 \\ 0 & e^{-i\pi/4} & 0 & 0 \\ 0 & 0 & 0 & e^{-i\pi/4} \\ 0 & 0 & e^{i\pi/4} & 0 \end{bmatrix} \tag{4.9}$$

In order to verify the correctness of the implementation of those gates we construct quantum circuits for *8* qubits with different number of columns and simulate them with our simulator and calculate the average bipartite entanglement between each qubit and the rest of the system. The measure of multi-qubit entanglement as a function of pure state was introduced by Meyer and Wallach in [MW02] and it was developed further by Brennen in [Bre03]. The measure of multi-qubit entanglement for the state $|\psi\rangle$ is defined as:

$$Q(|\psi\rangle) = \frac{4}{n}\sum_{k=1}^{n} D(|\tilde{u}^k\rangle, |\tilde{v}^k\rangle), \tag{4.10}$$

where vectors $\left|\tilde{u}^k\right\rangle$ and $\left|\tilde{v}^k\right\rangle$ are non-normalized and obtained by projecting on the state $|\psi\rangle$ by local projectors on the *k*-th qubit,

$$|\psi\rangle = |0\rangle_k \left|\tilde{u}^k\right\rangle + |1\rangle_k \left|\tilde{v}^k\right\rangle, \tag{4.11}$$

The function $D\left(\left|\tilde{u}^k\right\rangle, \left|\tilde{v}^k\right\rangle\right)$ is defined as:

$$D\left(\left|\tilde{u}^k\right\rangle, \left|\tilde{v}^k\right\rangle\right) = \sum_{i<j1} \left|\tilde{u}_i^k \tilde{v}_j^k - \tilde{u}_j^k \tilde{v}_i^k\right|, \tag{4.12}$$

and it computes a "distance" between the two vectors $\left|\tilde{u}^k\right\rangle$ and $\left|\tilde{v}^k\right\rangle$.

Consider the system of two qubits with the state vector:

$$|\varphi\rangle = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{4}|10\rangle + \frac{\sqrt{7}}{4}|11\rangle \tag{4.12a}$$

The projected vectors for first qubit are:

$$\left|\tilde{u}^0\right\rangle = \frac{1}{2}|0\rangle + \frac{1}{4}|1\rangle, \qquad \left|\tilde{v}^0\right\rangle = \frac{1}{2}|0\rangle + \frac{\sqrt{7}}{4}|1\rangle \tag{4.12b}$$

And the distance between these two vectors is:

$$D\left(\left|\tilde{u}^0\right\rangle, \left|\tilde{v}^0\right\rangle\right) = \frac{1}{2} * \frac{\sqrt{7}}{4} - \frac{1}{4} * \frac{1}{2} = \frac{\sqrt{7}-1}{8} \tag{4.12c}$$

It can be proved that the function $D\left(\left|\tilde{u}^k\right\rangle, \left|\tilde{v}^k\right\rangle\right)$ in the equation (4.12) can be expressed via partial trace for the reduced density operator of the *k*-th qubit:

$$D\left(\left|\tilde{u}^k\right\rangle, \left|\tilde{v}^k\right\rangle\right) = \frac{1}{2}\left(1 - Tr\left[\rho_k^2\right]\right), \tag{4.13}$$

The function Q in the equation (4.10) has the property: $0 \le Q \le 1$ with $Q=0$ for completely factorable states and $Q=1$ for generalized Bell states.

Figure 4.10 shows the distribution of Q for quantum pseudo-random circuits for 8 qubits with different number of columns.

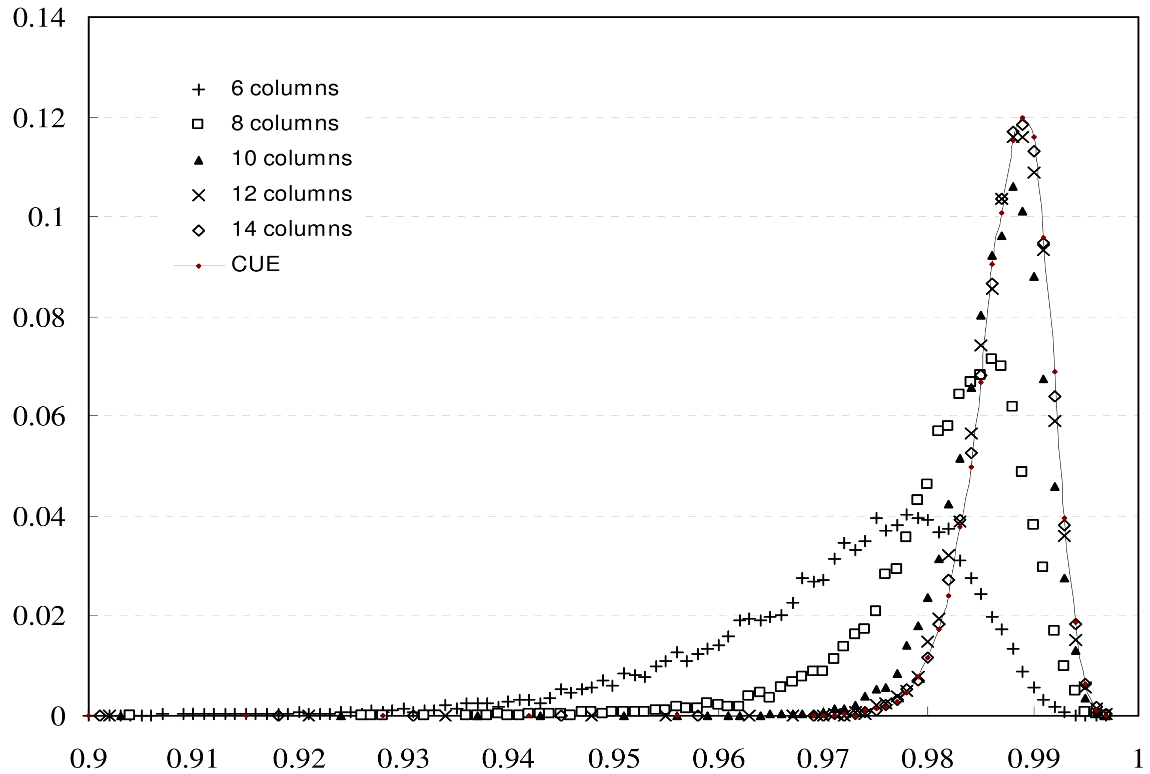


Figure 4.10. Distribution of Q for the columns of the pseudo-random quantum circuits with 8 qubits calculated by our simulator. The solid line is the Q distribution for the uniform (*CUE*) random unitary matrices.

As we can see from the figure 4.10 the result of the circuit with 14 columns is un-distinguishable with the result from the uniform random unitary matrices. This result is in good agreement with the result from [EWS+03].

4.5 Solovay-Kitaev algorithm and quantum circuit for factoring using only three gate types: Hadamard, T, and CNOT

A quantum gate is a unitary operator that can be represented by a unitary matrix. Any unitary matrix can be used as a quantum gate; therefore the number of quantum gates is infinite. Not all of gates can be directly implemented in the reality. Usually many quantum gates are built from finite set of elementary gates. *If a set of quantum gates can be used to approximate any gate with arbitrary number of inputs up to arbitrary accuracy then the set is considered universal*. The accuracy is estimated with the distance function between two operators U and S: $d(U,S) \equiv \|U - S\| \equiv \sup_{\|\psi\|=1} \|(U - S)\psi\|$. The S gate is said to be an approximation with accuracy ε if $d(U,S) < \varepsilon$. The choice of gates for a universal set depends on the feasibility of their implementation, but the following gates are often used: *CNOT, HADAMARD,* and *T*. The matrix for the *T* gate is:

$$T = \begin{bmatrix} e^{-i\pi/8} & 0 \\ 0 & e^{i\pi/8} \end{bmatrix} \tag{4.14}$$

The followings are some known universal sets:

- *CNOT* gate and all 1-qubit gates.
- *CNOT* gate, *Hadamard* gate, suitable phase shifts (e.g. T and T^{-1}).
- *Tofolli* gate and *Hadamard* gate.

It is important that not only a set can approximate a gate up to arbitrary accuracy, but also how many gates are needed to do that. Fortunately Solovay and Kitaev proved that the approximation can be done effectively [Kit97], [KSV02]. The process of finding a good approximation is called compilation. The Solovay-Kitaev theorem states the following:

> If $G \subseteq SU(d)$ is a universal set of gates, G is closed under inverse (i.e. for each g in G the inverse g^{-1} is also in G) and G generates a dense subset of $SU(d)$, then there exists a coefficient c such that for any $U \in SU(d)$ there exists a finite sequence of gates S from G of length $O(\log^c(1/\varepsilon))$ and such that $d(U,S) < \varepsilon$.

The Solovay-Kitaev theorem guarantees that a universal gate set can simulate any gate quickly, to a very good approximation. From a practical point of view, this is important in compiling a quantum circuit into a form that can be implemented fault-tolerantly. In article [DiV96] Dawson and Nielsen gives an explicit algorithm for compiling a gate based upon the Solovay-Kitaev theorem. They called the universal set of gates as "*instruction set*" and called each gate in the set as "*instruction*". Dawson posted a C-language implementation for the algorithm on his website, which can be used to compile an arbitrary gate into a sequence of instructions for any instruction set.

We use a universal set of gates W = {*HADAMARD, CNOT, T* and T^{-1} } to build quantum circuits for factoring with different levels of accuracy, simulate them with our simulator and compare the results from those circuit with the result from a perfect circuit. For convenience we call *HADAMARD* gate by *H* and T^{-1} by *t*.

The quantum circuit for Shor factoring algorithm using the universal set of gates {*H, CNOT, T* and *t* } is built from regular quantum circuit for factoring in the following three steps.

1. *Downsizing*: replace all gates with many control qubits by sequences of single-qubit gates and CNOT gates as described in figure 4.8b.
2. *Decomposing*: all gates with one control qubit are replaced by sequences of five gates (two *CNOT* and three single-qubit gates) as described in [BBV+95]

3. *Compiling*: all one qubit gates are replaced by sequences of T, t and H gates using Solovay-Kitaev algorithm.

We perform those steps to a quantum circuits for factoring the number *21* with the constant *2*. Recall that the Shor's circuit for factorization has four parts, in this case they are:

1. Hadamard part to put variable register into equal superposition.
2. Calculation of the function $2^n \mod 21$.
3. Perform inverse *QFT* for the variable register.
4. Measure the variable register to get the result.

The first part has only *HADAMARD* gates. The part for the calculation of the function $2^n \mod 21$ is built after the construction from [VBA95] and is consists of TOFFOLI, *CNOT, NOT* gates. The inverse *QFT* part consists of *HADAMARD, SWAP*, and *CTRL_ROTATE_n* gates where n runs from *2* to *10*. The *CTRL_ROTATE_n* gate is the controlled version of the *ROTATE_n* gate with the matrix:

$$ROTATE_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^n} \end{bmatrix} \tag{4.15}$$

The *SWAP* gate is implemented with three *CNOT* gates. The *NOT* gate has compilation *HttttH*. The *CTRL_ROTATE_n* is decomposed into the sequence of gates from *W* using the construction from the article [BBV+95], which shows that any gate V in SU(2) can be decomposed as the sequence *A.B.C*, where

$$A = \begin{bmatrix} e^{i\alpha/2} & 0 \\ 0 & e^{-i\alpha/2} \end{bmatrix} \quad B = \begin{bmatrix} \cos\theta/2 & \sin\theta/2 \\ -\sin\theta/2 & \cos\theta/2 \end{bmatrix} \quad C = \begin{bmatrix} e^{i\beta/2} & 0 \\ 0 & e^{-i\beta/2} \end{bmatrix} \tag{4.16}$$

and the controlled version of V gate is implemented as:

$$Ctrl\text{-}V(t,c) = C(t)CNOT(t,c)B(t)CNOT(t,c)A(t) \tag{4.17}$$

The execution of gates in (4.17) is performed from left to right. We wrote a small program to decompose the matrix for *ROTATE_n* gate into three matrices: *ROTATE_A_n, ROTATE_B_n, ROTATE_C_n* as specified in [BBV+95] so that its controlled version can be implemented as:

$$CTRL_ROTATE_n(t,c) =$$

$$ROTATE_C_n(t)CNOT(t,c)\ ROTATE_B_n(t)CNOT(t,c)ROTATE_A_n(t) \tag{4.18}$$

The decomposition for the *TOFFOLI* is taken from [DiV96]:

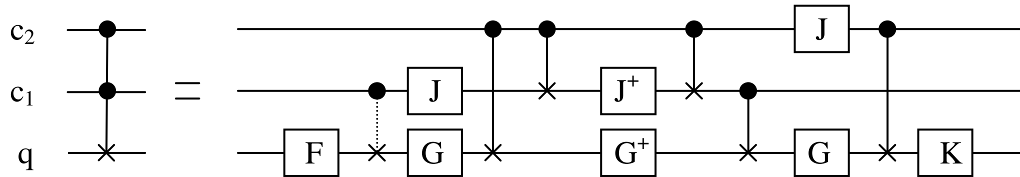


Figure 4.11. One of the simplest known decompositions of the *TOFFOLI* gate.

The gate sequence for a quantum simulator for the decomposition is:

$$TOFFOLI(q,c_1,c_2) = K(q)\ CNOT(q,c_2)\ G(q)\ J(c_2)\ CNOT(q,c_1)\ CNOT(c_1,c_2)\ G^+(q)\ J^+(c_1)$$

$$CNOT(c_1,c_2)\ CNOT(q,c_2)\ G(q)\ J(c_1)\ CNOT(q,c_1)\ CNOT(q,c_1)\ F(q) \tag{4.19}$$

where

$$F = \begin{bmatrix} e^{i\pi/4}\cos\pi/8 & e^{i\pi/4}\sin\pi/8 \\ -e^{i\pi/4}\sin\pi/8 & e^{-i\pi/4}\cos\pi/8 \end{bmatrix} \qquad G = \begin{bmatrix} \cos\pi/8 & -\sin\pi/8 \\ \sin\pi/8 & \cos\pi/8 \end{bmatrix}$$

$$K = \begin{bmatrix} e^{i\pi/4} & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix} \qquad J = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\pi/4} \end{bmatrix} \tag{4.20}$$

The matrix *J* in (4.20) doesn't belong to SU(2) group but we can use its equivalent as

$$J_{equivalent} = \begin{bmatrix} e^{i\pi/8} & 0 \\ 0 & e^{-i\pi/8} \end{bmatrix} \tag{4.21}$$

Our numerical calculation shows that: *F = TTHTHTTTT, G = ttHTHTT,* G^{+} *= ttHtHTT, J= t.* The right hand side of the equation (4.19) can be considered as the composite *TOFFOLI* gate and it differs from real *TOFFOLI* gate only by floating point representation error of *1.67e-16*, which is very small.

There is a very useful directive in our simulator for quantum circuit analysis: {HISTOGRAM}. The primary purpose of histogram is to graphically represent the probability distribution for the whole quantum system, but we found that sometimes we just need the probability distribution over values of a group of qubits (or a quantum register). Shor's circuit for quantum factoring is a perfect example of such need. The last operation in the circuit is measuring the variable register, therefore only the probability distribution for that register matters. We implement {HISTOGRAM} directive with two parameters that specify the start and the end of a quantum register. These two parameters are enough to specify the whole quantum system or a single qubit if needed. The {HISTOGRAM} directive displays the probability for every possible index for the quantum register on screen or writes the data to a log file. With the histogram for the variable register we can predict the outcome of the measurement without actually measuring it. Of course this can be done only in a simulator. The real implementation of a quantum circuit needs a measurement to obtain the result.

The compilations for *ROTATE_n* gates with the recursion depth of *1* have trace distance errors range from *0.0* to *0.0981* with the average of *0.025*. Simulation of the approximated circuit gives good result: the histogram has same peaks as the ones for optimal circuit (see figure 4.12). The difference in probabilities for the two circuits is

0.22 or *22%*. As we can see from the histograms the success rate for factorization of 21 is the sum of probabilities for the indices: *170, 171, 341, 342, 512, 682, 683, 853, 854*; all these indices lead to a factor of *21* in the post-experiment calculation. The probability for index *0* is very high, but it contributes nothing to the success rate because no information could be derived from *0*. With such consideration the success rate for perfect circuit is *0.737* and success rate for the approximated circuit is *0.669* and the difference is only *6.8%.* The trace distance errors range from 0.0 to 0.0406 with the average of 0.016 for the recursion depth of 2; the success rate is 0.699 and it differs from the success rate for the ideal circuit by 3.8%.

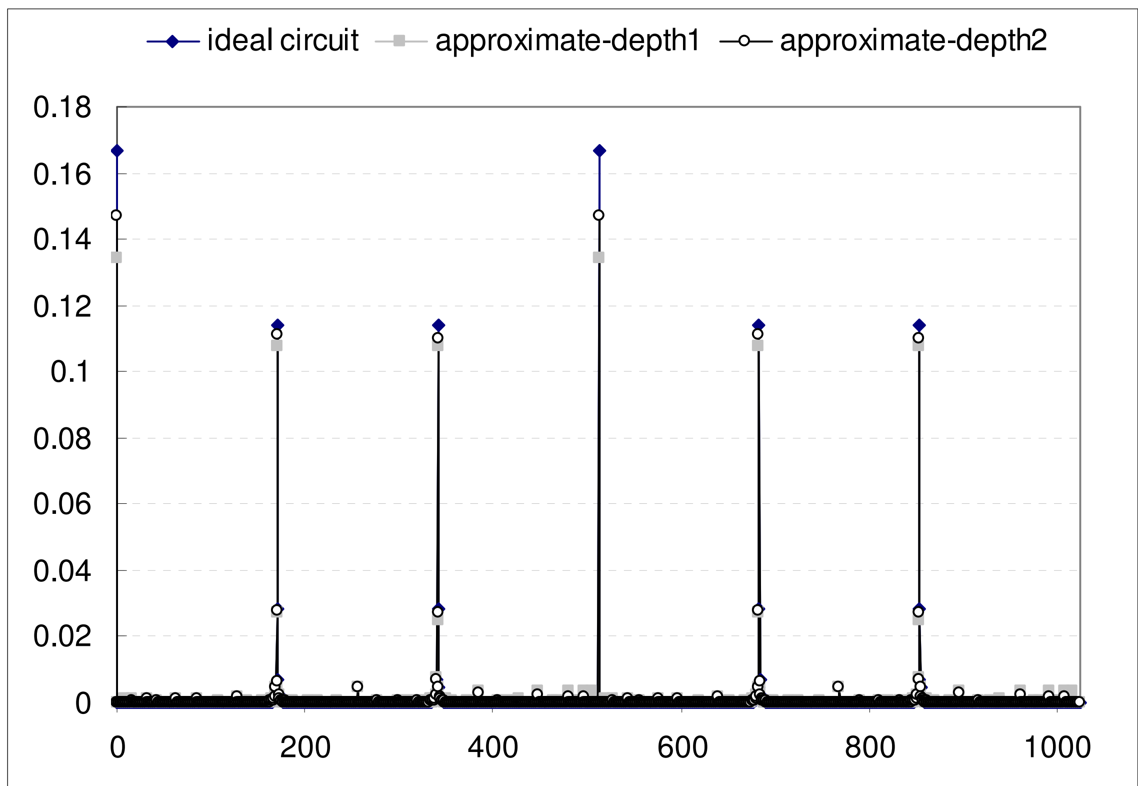


Figure 4.12. Histograms for the variable register at the end of quantum computation for the ideal and approximated circuits for factoring. The indices with high probabilities are 0, 170, 171, 341, 342, 512, 682, 683, 853, 854.

4.6 Semi-classical quantum circuit for factorization

The Shor circuit for quantum factorization uses two quantum registers: one for the result and the other for the variable x of the function $a^x \bmod N$. The size for the variable register must be at least double the size of the result register to ensure the probability of success. With the current state of technology creating a quantum system with many qubits is extremely difficult. The quantum system with most qubits is an *NMR* system with *7* qubits and it seems to reach the limit for the technology.

After Shor published his research for quantum factorization in [Sho94] scientists began research to reduce the number of qubits in the circuit. Recall that the last quantum computation step in the circuit is performing inverse Quantum Fourier Transform on the variable register followed by a measurement all qubits in the register (figure 4.13). This process uses many two-qubit gates.

In [GN96] R.B. Griffiths and C. Niu proposed a method to perform *QFT* in a semi-classical fashion. The main approach here is measuring separate qubits and using the results to control the next quantum operators in the circuit. This is a new technique, in which the result of the measurement is converted into a classical signal that can control the execution of one or more quantum operators. Such a technique is mentioned in [BBC+93] for quantum teleportation. Griffiths and Niu proved that the measurement can be performed separately for each qubit, beginning with the qubit in the lowest line to the qubit in the highest line, and the result of each measurement is used for subsequent quantum operations. This semi-classical process gives the same result as the pure

quantum process. The use of a classical signal to control a quantum gate in the *QFT* process frees all two-qubits gates for the process; that is very encouraging. The circuit is greatly simplified because of that.

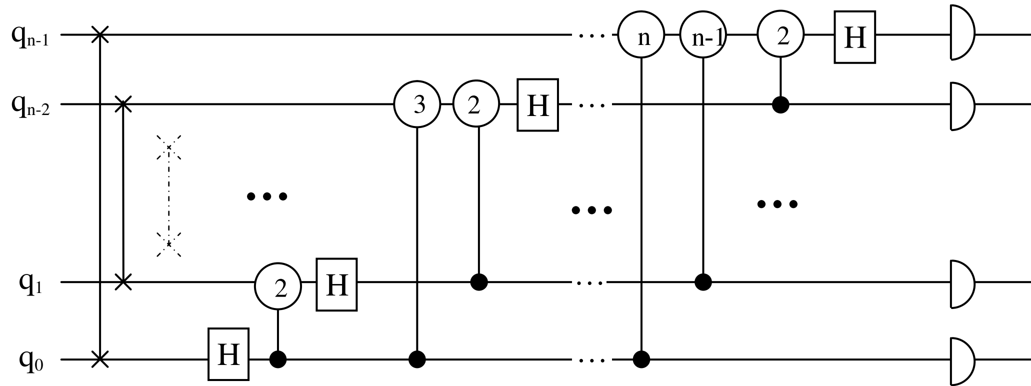


Figure 4.13. The last part of Shor circuit for quantum factorization.

Half circles are measurement gates.

In [PP00] S. Parker and M.B. Plenio used the result from [GN96] to develop a quantum circuit that uses only one qubit for the variable register for all the operations: calculation of the function $a^x \ mod \ N$, performing the inverse *QFT* and the measuring. They observed that each qubit in variable register controls a separate block (or, in our terminology, a composite gate) and it in order to get the "kick-back" phase from the block. By re-arranging those blocks to perform the $Ctrl_U^{2^{n-1}}$ first and measure the control qubit, then perform $Ctrl_U^{2^{n-2}}$ and measure the control qubit, …, and perform $Ctrl_U^{2^0}$ last and measure the control qubit it is possible to use only one qubit for this purpose. Here the $Ctrl_U^k$ gate is the controlled version of the U^k gate that is defined in the formula (1.44)

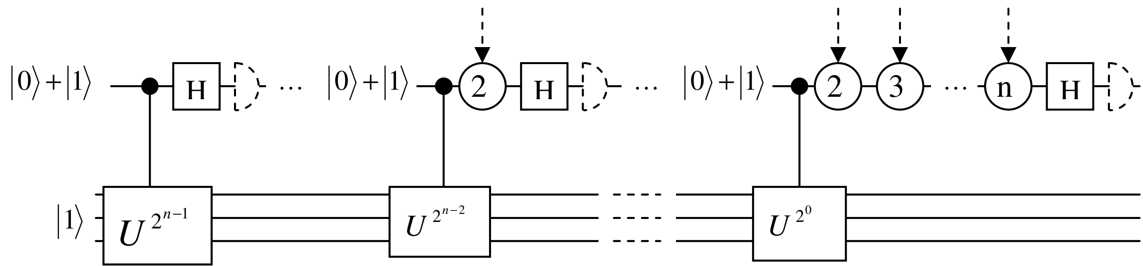


Figure 4.14. Semi-classical circuit for factoring. The dashed half circles represent the measurements with classical signal output. The circles with dashed arrow are quantum gates that are controlled by a classical signal.

Special care must be taken so that the control qubit is always in equal superposition before performing $Ctrl_U^{2^k}$. The schematic diagram for the semi-classical circuit is represented in figure 4.14.

As we mentioned earlier none of the quantum simulators that we know of allow a user to design and to simulate such a semi-classical circuit. Our simulator provides two special gates for measurement with classical output signal and for using the classical signal to control the execution of a quantum gate.

The measurement gate for this purpose is called *INTERNAL_MEASURE*. Its parameters include *startBit*, *endBit* and *storageIndex*, where first two parameters specify a quantum register and the last one specifies an internal storage, hence the name of the gate has the "internal" word. The storage index is used to specify the out filename as well. One can measure the same quantum register and store the result in different storage every time. The value from an internal storage is used to control the execution of a quantum gate via another special gate: *INTERNAL_CONTROL*. There are two parameters for the gate; the

first one specifies the internal storage and the other one specifies a value. If the provided value equals the value in the storage then next quantum gate will be executed, otherwise the quantum gate will be skipped. All internal storages can be considered as one semi-classical register; but unlike the classical registers mentioned in sub-section 4.1 the content of this register is unknown in designing phase and its value is updated at the run time. We don't provide a directive to set a value for an internal storage, because if one knows the value beforehand without running the circuit he should use a simpler gate with same effect to make the circuit cleaner.

With the *INTERNAL_MEASURE* and *INTERNAL_CONTROL* gates and we build a semi-classical circuit for factoring number 21, simulate it. The result of the circuit, or the value for the variable register, is combined from all internal measurements. Unlike pure quantum circuit for factoring this circuit could not provide the histogram for the variable register in regular manner via {HISTOGRAM} directive. To obtain the histogram for semi-classical circuit we simulate it many times and accumulate the result into a file. The evaluated histogram is compared with the histogram from pure quantum circuit in figure 4.15.

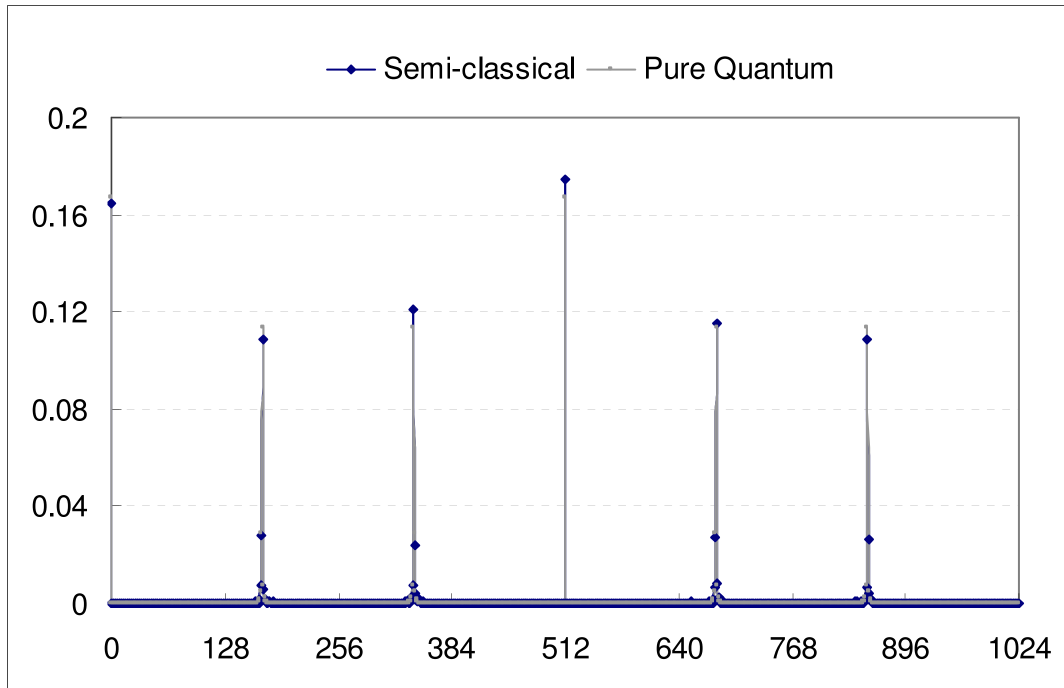


Figure 4.15: Histograms for pure and semi-classical quantum circuits for factoring.

As we can see from the figure the semi-classical circuit gives almost the same result for the computation. Detailed calculation shows that the total difference for probabilities is 0.08; and the difference for success rate is 0.034 or 3.4%. The pure quantum circuit uses $5n+1$ qubits where the variable register has $2n$ qubits. The semi-classical circuit uses $3n+2$ with only one qubit for the variable register.

4.7. Future research

Qsim is an effective tool for design, test and analyze of quantum circuits. In near future we can:

1. Use qsim to research problems in quantum cryptography and Braid group.
2. Improve the simulation time for qsim for circuits with non-sparse state vector. This can be done with a static system state vector as an array in memory, qsim will run in dynamic option as default (with *QSV* state vector) and if it detects that the state vector is non-sparse it will suggest the user to stop and run qsim again with static option.
3. Implement the system state vector via external storage to increase the actual size for the state vector that can be simulated. Because qsim uses only sequential access to the state vector it is possible to implement the system state vector as a file on a hard drive and extend the number of circuits that can be simulated by qsim. With this approach and 64 GB of available storage qsim can simulate circuits with actual state vector size of 2^{32} components. Some researchers expressed the desire to use qsim to solve their problem with *30* or more qubits in independent superposition, which result in state vector size of 2^{30} or more.
4. Develop a graphical user interface to allow users to design and test simple circuits.

Conclusion

We developed an innovative method for calculation of the new state vector after application of a quantum gate in a quantum simulator that allows us to utilize a compact data structure to store the state vector for a quantum system and directly perform all operations of a quantum simulator with it. The specialized data structure, *QSV*, makes it possible to speed up the simulation time. The quantum simulator that is built with these techniques overcomes many shortcomings in other simulators and it has many advanced features.

Our simulator can simulate many 64-qubit circuits while allowing users to define gates with up to 10 qubits with simple syntax. Nevertheless, the simulation time is much less than the one from other simulators for many circuits. The specifications for the simulator allow users to design complicated quantum circuits and test them with many debugging features. It is a great advantage that the output from a measure gate is stored in a file which allows users to establish interaction between quantum and classical parts of a computation as it is in the Shor algorithm for quantum factorization. With many useful directives in our simulator users can get many parameters from the quantum system being simulated such as Shannon entropy, partial trace, or histogram for a group of qubits. The simulator also defines some gates that can be used to create random quantum states that are useful in designing and testing circuits for quantum communication.

With the simulator in this thesis we developed some new quantum circuits and verified many designs and constructions. The simulator allows us to build a semi-classical

quantum circuit for factoring with only one qubit for the variable register and a quantum circuit for factoring using only three gate types: *CNOT, Hadamard* and *T* as well as successfully simulate them. We also use our simulator to perform numerical analysis for many quantum circuits.

The additional information from the quantum state vector such as the Shannon entropy, partial trace for qubits and histograms allows using the simulator as an educational tool as well.

Appendix A

Qsim specifications

A.1 The circuit file

Like any other general purpose quantum simulator *qsim* needs a quantum circuit to simulate. Many simulators accept only quantum circuits that are built and saved by themselves using some graphical user interface (*GUI*). This has an advantage for beginners but it raises a serious disadvantage for complicated circuits such as *QFT* circuit, the circuit for addition in *QFT* state, or the circuits for modular exponentiation that sometimes use more than *1000* gates.

Qsim uses a text file to specify a quantum circuit. This allows designers to use any text editor to design their circuits. The main advantage for this approach is that the designers can use some scripts to create complicated circuits with many modules and connections. Scripts can also be used to extract some part of other circuits and to perform some parameters substitutions before putting that part into a new circuit. Using a script to create complicated circuits is more reliable than using *GUI*.

Qsim has a built-in parser that parses and loads a circuit file into the computer memory for simulation. The followings are rules for a circuit file.

1. A circuit must begin with {CIRCUIT_BEGIN} {CIRCUIT_WIDTH n} and it ends with {CIRCUIT_END}. {CIRCUIT_WIDTH} directive specifies the number of qubits in the circuit.

2. Comment begins with # and lasts till end of line.

3. All gates and directives are after {CIRCUIT_WIDTH n}. They have the format:

   {<gate-name>|<directive-name> [<bit-list>|<parameters>] [;<matrix-elements>] [;<value>]}

   Here we use the following conventions:

   - Items inside square brackets [] are optional
   - Items that are separated by | are mutually exclusive, only one of them appears at a time.
   - Each item inside a pair of angular brackets is a non-separable part of a gate or a directive.

     a. <gate-name>

        The part specifies a gate name. A gate name can use any characters and digits, except for white spaces and "{}(),;:". The ":" character is used for special set of built-in gates, the *Gate-Families*, to separate the family name and the parameter to those families. The gate name uniquely identifies the operation it performs. *Qsim* uses descriptive names, such as *HADAMARD*, *CNOT, TOFFOLI*, ... Any gate name not found in list of built-in or previously-defined gates is treated as user-defined gate. A user-defined gate needs *matrix-elements*, the list of coefficients for its matrix, for its first appearance in the circuit. Subsequent use of the same user-defined gate doesn't need the matrix part. This will reduce the error chances and make the circuit easier to read. There is another method for specifying a user-defined gate that will be explained in subsection A.5.7.

b. <directive-name>

This part specifies a directive. Directives have the same format as gates, with the exception that some of them may not have the second part. Many directives have parameter part after their names.

c. <bit-list>

The next item after the gate name is the bit list. Every gate has one or more pins. *<bit-list>* specifies the pin assignments to real qubits for those pins. *<bit-list>* is the list of numbers separated by a comma. The first number in the list specifies the connection for the first pin of the gate, the second number specifies connection for the second pin,… If the gate has *<matrix-elements>* or *<value>* part then the bit list needs a separator ";". The size of *<bit-list>* must match the number of pins for the gate. If the gate is user-defined then the *<bit-list>* in its first appearance defines the number of pins for the gate.

d. <parameters>

This part specifies parameters for a directive. The number of parameters depends on particular directives.

e. <matrix-elements>

This part specifies a matrix. As mentioned earlier, a user-defined gate needs a matrix for its first appearance in the circuit. The matrix for a gate must be square one and its elements are listed from left to right, top to bottom in the complex number format: (*real, imaginary*). Coefficients are

separated by commas. If the size for the <*bit-list*> is n then the number of matrix elements must be exactly n^2.

f. <value>

Some gates and directives require additional parameter: a value. E.g. an {INPUT} directive for a qubit expects a value 0 or 1 for its initial state

4. White space characters such as blanks and tabs between gates and directives will be discarded. White space characters after commas are discarded as well.
5. All gates and directives will be introduced in the following sections.

If *qsim* finds an error while it's parsing a circuit file it displays a descriptive error message with the line number and stops. *Qsim* begins the simulation only after the circuit file is successfully parsed and loaded. When *qsim* finishes the simulation it displays the result on screen or writes the message to a log file.

A.2 Input for a circuit

A quantum circuit is designed to do some task or some function. The designer has to specify the initial state of the quantum system before the calculation begins. As the article [DiV00] specifies any real quantum system for computation must be able to initialize to a fiducial state. Therefore a computational basis state is a possible input for *qsim*. *Qsim* has two options to specify a computational basis state:

1. Use {INPUT bit; value} directives for each qubit, e.g. {INPUT 2; 1} that means the qubit with index *2* has initial value of *1*.

2. Use {INPUT startBit, endBit; value} directive for group of qubits, e.g. {INPUT 1,4; 6}. This is the most convenient way to for the designers, because we often group qubits

into registers to perform a function, and we want to specify the state of the whole group as single value.

The third method to specify initial state for *qsim* is using {INITIAL_STATE} directive. The format for the directive is:

{INITIAL_STATE {index; (real, imaginary)} [,{index; (real, imaginary)}] ...}. Every tuple {index; (real, imaginary)} is for one computational basis state where *index* is the index of the basis state and (*real, imaginary*) pair is the probability amplitude for the basis state. The probabilities for all basis states in the directive must sum to one, otherwise the parser in the simulator will stop and display an error message.

This allows the designer to use any superposition state as initial state and not worrying about how to achieve that state. Later when the design for the circuit will be complete with desired result, the designer can use the construction in section 4.3 to achieve the superposition state.

A.3 Output for a circuit

When all the gates in a quantum circuit finish their action the quantum computation ends. A quantum circuit may be designed to do complete work without consulting other calculation, e.g. a circuit to perform addition of to numbers that are stored in its registers and the result is read out at the end of the computation. For such circuits the designer can specify the expected result for all or some qubits via {OUTPUT} directives. There are two formats for the {OUTPUT} directives, similarly to the {INPUT} ones:

1. {OUTPUT bit; expected-value}

2. {OUTPUT startBit, endBit; expected-value}

With the {OUTPUT} directives our simulator can check for the correctness of the computation. If the designer specifies the expected value for a qubit to be 1, but after the simulation if qsim finds that its value is 0, then it displays an error message for the computation. This is very useful for the designers: they can check if the circuit works as they want.

Sometimes some qubits are left at superposition at the end of computation and the designer cannot specify *0* or *1* for them. For such situations *qsim* uses the value *-1* for the expected-value and those qubits will not be used to determine the correctness of the computation.

If the quantum circuit is a part of broader computation after quantum computation completes other parts will take the result from the circuit to perform further calculations. Shor's algorithm for quantum factorization is an example. From the output files for *MEASURE* gates other programs can obtain the result from the quantum computation. Therefore *qsim* provides seamless integration between classical and quantum computations.

There may be another output file from *qsim*: the log file. If *qsim* is invoked with *-l* option it will redirect most of its messages to a log file. If the *-l* option has a filename then *qsim* uses that to log messages, otherwise it uses <*circuit_name*>*.log* to log.

A.4 Built-in gates in *qsim*

A.4.1 Gates without parameters

In this section we list all built-in gates that don't have any parameter in the *qsim* format in alphabet order. In *qsim* the target qubit is always the first onr in the bit-list. Qubits count from *0*. The letter *q* is often used for target qubit, the letter *c* is often used for control qubit.

1. {CNOT q, c}

   The qubit q flips its state if the qubit c is 1.

2. {FREDKIN q0, q1, q2}

   The matrix for this gate is: $FREDKIN = \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & & 1 & \\ & & & & & 1 & & \\ & & & & & & & 1 \end{bmatrix}$

3. {HADAMARD q}

   The matrix for this gate is: $H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$

4. {INTERNAL_CONTROL storage_id; value}

   If the value in the storage with index *storage_id* equals provided *value* then next quantum gate will be executed, otherwise next quantum gate will be skipped. E.g. the sequence {INTERNAL_CONTROL 0; 1} {NOT 2} will skip the *NOT* gate for the qubit *2* if internal storage #*0* has value *0*.

5. {INTERNAL_MEASURE start_bit, end_bit; storage_id}

This gate measures the quantum register that is specified by *start_bit* and *end_bit*, appends the value to file with filename <*circuit-name*>.*im*<*n*> where <*n*> is the storage_id and stores the value in the internal storage with index storage_id. E.g. for the circuit name is *factoring.cir* the {INTERNAL_MEASURE 0, 2; 3} gate measures qubits *0,1,* and *2*, stores the result in the internal storage #1 and appends the result to the file *factoring.im3* as well.

6. {MEASURE start_bit, end_bit}

   This gate measures the quantum register that is specified by *start_bit* and *end_bit*, appends the value to a file with filename <*circuit-name*>.*mr*<*n*> where <*n*> is the order of the *MEASURE* gate. First *MEASURE* gate in the circuit is associated with the extension *mr1*, the second *MEASURE* gate is associated with the extension *mr2*, and so forth.

7. {NOT q}

   This gate flips the state of the qubit *q*.

8. {RANDOM_U2 q}

   The matrix for this gate is filled at loading time. Its coefficients are generated using the construction for circular unitary ensemble (*CUE*) as described in [PZK98], see (4.8).

9. {SQ_RT_NOT q}

   The matrix for this gate is: $SQ_RT_NOT = \frac{1}{\sqrt{2}}\begin{bmatrix} i & 1 \\ 1 & i \end{bmatrix}$

10. {SWAP q1, q2}

    The states of two qubits *q1* and *q2* are swapped.

11. {TOFFOLI target, c1, c2}

The state of the qubit *q* is flipped if the state of two control qubit *c1* and *c2* is $|11\rangle$

The gates #12 through #19 act the same as the TOFFOLI gate, with more control qubits only.

12. {TOFFOLI3 target, c1, c2, c3}

13. {TOFFOLI4 target, c1, c2, c3, c4}

14. {TOFFOLI5 target, c1, c2, c3, c4, c5}

15. {TOFFOLI6 target, c1, c2, c3, c4, c5, c6 }

16. {TOFFOLI7 target, c1, c2, c3, c4, c5, c6, c7 }

17.{TOFFOLI8 target, c1, c2, c3, c4, c5, c6, c7, c8 }

18. {TOFFOLI9 target, c1, c2, c3, c4, c5, c6, c7, c8, c9}

19. {TOFFOLI10 target, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10}

20. {TWO_BODY_INTERACTION q1, q2}

The matrix for this gate is: $\begin{bmatrix} e^{i\pi/4} & 0 & 0 & 0 \\ 0 & e^{-i\pi/4} & 0 & 0 \\ 0 & 0 & 0 & e^{-i\pi/4} \\ 0 & 0 & e^{i\pi/4} & 0 \end{bmatrix}$

A.4.2 Gates with a special parameter

These gates are special. They have a parameter as part of the gate name. The parameter appears after “:”. The letter *q* specified the qubit; the letter *c* specifies control qubit or qubits.

21. {ROTATE_ONE:n q}

The matrix for this gate is: $ROTATE_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{i2\pi/2^n} \end{bmatrix}$

22. {INV_ROTATE_ONE:n q}

The matrix for this gate is: $INV_ROTATE_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{-i2\pi/2^n} \end{bmatrix}$

23. {CTRL_ROTATE:n q, c}

The matrix for this gate is: $CTRL_ROTATE_n = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i2\pi/2^n} \end{pmatrix}$

Notes: {CTRL_ROTATE:n q, c} is equivalent to {CTRL_ROTATE:n c, q}, i.e. all involved qubits have the same role. The same feature applies for three gates below.

24. {INV_CTRL_ROTATE:n q, c}

The matrix for this gate is: $INV_CTRL_ROTATE_n = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{-i2\pi/2^n} \end{pmatrix}$

25. {CTRL_CTRL_ROTATE:n q, c1, c2}

This gate acts the same as {CTRL_ROTATE:n} gate except it has one more control qubit, *c2*.

26. {CTRL_INV_CTRL_ROTATE:n q, c1, c2}

This gate acts the same as {INV_CTRL_ROTATE:n} gate except it has one more control qubit, *c2*.

A.5 All *qsim* directives

In this section we describe all directives for *qsim*. All command-line options will be explained in next section.

1. {DUMP}

   This directive displays all elements in the state QSV vector to the screen or outputs them to the log file if the option *-l* is specified. The output format is:

   Dump at gate #N {index, amplitude}{index, amplitude} … E.g.:

   Dump at gate #1 {0,(0.707107,0)}{1,(0.707107,0)}

   If the option *-b* is specified then the index will be written in binary format. There is a letter *B* at the end of binary string to distinguish the binary format. This is useful for debugging. E.g.

   Dump at gate #1 {000B,(0.707107,0)}{001B,(0.707107,0)}

2. {LONG_DUMP}

   This directive acts the same as the {DUMP} directive except each element has one line and the format is more verbose. E.g.

   Dump at gate #6 state |1> amplitude=(0.707107,0)

   state |3> amplitude=(0.707107,0)

   The option *-b* can be used with this directive as well.

3. {HISTOGRAM start_bit, end_bit}

   This directive calculates histogram for the quantum register that is specified by *start_bit* and *end_bit* and displays the result on screen or writes the log file if the option *-l* is used.

4. {HISTOGRAM_COMPACT start_bit, end_bit}

This directive acts the same as {HISTOGRAM} directive except all zero probabilities will not be displayed (or will not be written to the log file).

5. {ENTROPY}

This directive calculates the Shannon entropy for the quantum system and appends the result to a log file as a new line in the format "order_of_ENTROPY_directive, entropy, global_order". The global order is the order for all gates and directives. If the first {ENTROPY} directive has global order 15 and the entropy of the system at the gate/directive #15 is 2.5 then the output line is: "1, 2.5, 15". The Shannon entropy for the quantum system is calculated using the following formula:

$$Shannon_entropy = -\sum_{p_x \neq 0} p_x \log(p_x)$$

The result file for the entropy is *<circuit-name>.ent*.

6. {PARTIAL_TRACE q}

This directive calculates the partial trace for the qubit *q* and writes the result to a file with the filename *<circuit-name>.trc<n>* where *<n>* is the value of *q*. E.g. {PARTIAL_TRACE 2} in the *factoring.cir* circuit will writes the log file *factoring.trc2*. This directive should not be used for circuits with more then 20 qubits, because calculation of the partial trace requires huge memory.

7. {UDEF:<gate_name> <n_Pins>; <matrix_elements>}

This directive defines a new gate for later use. The gate name is specified after "*UDEF:*", the number of pins is specified by *<n_Pins>* parameter, and the coefficients for the matrix are specified by *<matrix_elements>* parameters like they are defined in subsection A.1.3.d. After a new gate is defined by this directive it can appear in the circuit file as a built-in gate.

A.6 Command-line options for *qsim*

Qsim has the following command-line options:

-b

Use binary representation for amplitude indices

-c <circuit_name>

Used to specify the filename for a quantum circuit without extension. The circuit file itself must have extension *.cir*.

-d

Used to set the debug flag is ON. Qsim will output much more of information for debugging purpose.

-g

Used to display circuit's gates and pin assignments without matrices.

-h

Used to display help.

-H

Used to display basic rules for circuit.

-i <input_file>

Used to specify input data file for batch processing. Each line in the file has three numbers, the first number specifies the initial basis state, the second one specifies the final basis state and the last one specifies the mask. The mask is used for setting un-defined-on-output qubits to 0. E.g. if a circuit has a *Hadamard* gate for qubit 1 as the last gate, then qubit 1 is in superposition and one cannot predict its

state. The designer can use the mask to clear this bit in the final state before *qsim* checks for the correctness of the computation. Setting both the mask and the final basis state to *0* always makes *qsim* to report “success”. The values for all three parameters can be in decimal, hexadecimal (e.g. *0xA3*) or binary (e.g. *1001B*) formats. For each correct line *qsim* will perform a simulation and then checks for the correctness and displays or logs messages. All lines with error will be skipped without notice.

-l [log_filename]

Used to redirect most messages to a log file. If a filename is specified then *qsim* will write its messages to the file, otherwise it uses default log filename as *<circuit_name>.log.*

-m

Used to display names for all gates, their pin assignments and their matrix. All directives and some special gates, which have no matrix, will not display the matrix part. This option should not be used for circuits that have {TOFFOLI5}, {TOFFOLI6}, .., {TOFFOLI10} gates because the matrix part for them will be very long.

-p

Used to parse the circuit file only without simulation.

Appendix B

Sample circuit for the quantum simulator

Quantum circuit for addition of two 3-bit numbers, after [VBE95]

It uses three registers: 1) source_reg from qubit0 to qubit2

2) result_reg from qubit3 to qubit5 3) working_reg from qubit6 to qubit8

Content of source_reg will be added to content of result_reg; source_reg is unchanged

working_reg is working register; its content must be zero on input

On output the highest bit of working_reg is the carry, other bits are zeros

```
{CIRCUIT_BEGIN}

{CIRCUIT_WIDTH 9}

{INPUT 0,2; 7}

{INPUT 3,5; 2}

{INPUT 6,7; 0}

{INPUT 8; 0}

# Begin gate part

{TOFFOLI 6, 0, 3}

{TOFFOLI 7, 1, 4}

{CNOT 4, 1}

{TOFFOLI 7, 6, 4}

{TOFFOLI 8, 2, 5}

{CNOT 5, 2}

{TOFFOLI 8, 7, 5}
```

```
{CNOT 5, 7}

{TOFFOLI 7, 6, 4}

{CNOT 4, 1}

{TOFFOLI 7, 1, 4}

{CNOT 4, 1}

{CNOT 4, 6}

{TOFFOLI 6, 0, 3}

{CNOT 3, 0}

# End gate part

{OUTPUT 0,2; 7}

{OUTPUT 3,5; 1}

{OUTPUT 6,7; 0}

{OUTPUT 8; 1}

{CIRCUIT_END}
```

Appendix C

Comparison with other quantum simulators

We tried to download and tested many general-purpose quantum simulators. Many of them could not run, due to unavailability of some components or libraries.

We are able to download and run *QCAD* version 1.80 from the website http://acolyte.t.u-tokyo.ac.jp/~kaityo/qcad. The simulator has a graphical user interface with few gates and it can not simulate any circuit with 11 qubits; therefore we consider it not suitable for serious quantum simulation and we don't try to compare the results from QCAD with ours.

The most advanced quantum simulator we found is the *Fraunhofer Quantum Computing Simulator*. This is a web-based service that runs on 32-node Linux cluster at http://www.qc.fraunhofer.de/. You have to subscribe to the service and log-in before you can use it. It uses graphical user interface with many basic gates, such as NOT, CNOT, HADAMARD, SWAP, TOFFOLI, …

It has a gate called *control-phase* that is similar to our CTRL_ROTATE gate that can be used to implement QFT circuit, but it doesn't have the inverse gate for that one therefore one cannot implement inverse QFT circuit. It allows user to define only arbitrary two-qubit gates. It has some *function-gates* that are just another name for our composite gates, such as Grover-Oracle gate, Grover-Iteration gate, QFT, …. On input all qubits are set to

the state $|0\rangle$. On output all qubits have their own Bloch representation and one has to use the specified coordinate to calculate the value for $|0\rangle$ and $|1\rangle$ components.

We could not design the Draper addition circuit with the Fraunhofer simulator due to lack of required gates. There is no convenient MEASURE gate to implement different versions of Shor's quantum circuit for factoring as well. Therefore we just created four test circuits with many gates to measure the timing to compare with our simulator.

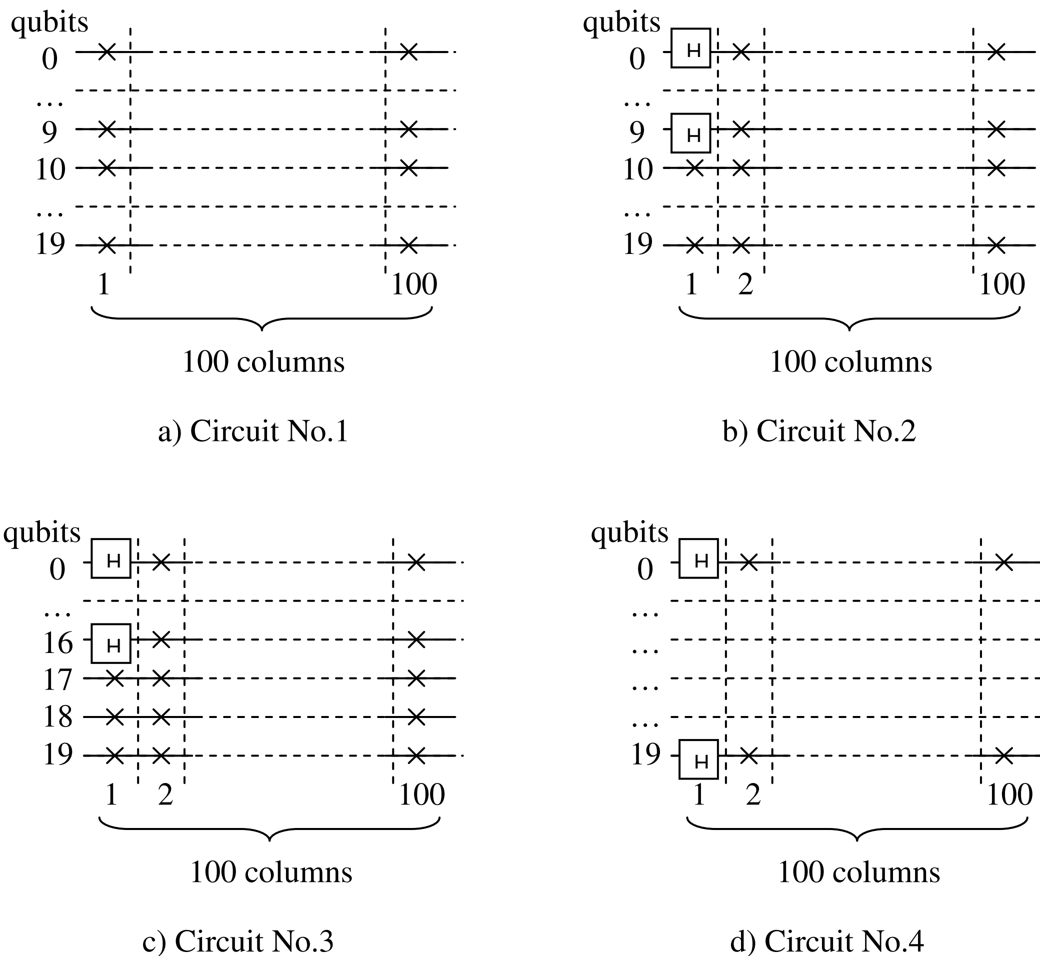


Figure C1: Four quantum circuits for timing comparison between qsim and Fraunhofer simulators. Each circuit has *20* qubits.

Recall that simulation time for qsim depends on the size of the QSV state vector, or the number of non-zero components in it. The test circuits will create different state vector sizes. All of them have *20* qubits and *20* columns (the collection of quantum gates that can be performed in parallel) with the same *19* last columns that have only NOT gates. The first circuit has all NOT gates in the first column as well, which results in *20* identical columns. The second circuit has *10* HADAMARD gates for 10 qubits and *10* NOT gates for last *10* qubits, and the third circuit has *17* HADAMARD gates for *17* qubits and *3* NOT gates for last *3* qubits in the first column. The last circuit has *20* HADAMARD gates for all qubits in the first column. The total number of gates for all circuits is *2000*. The simulation time for qsim and the Fraunhofer simulator is reported in the table C1.

Circuit No.	Maximum QSV vector size	Simulation time	
		Qsim	Fraunhofer simulator
1	1	< 1 sec.	11min. 4 sec.
2	2^{10}	5 sec.	11 min. 13 sec.
3	2^{17}	11min. 14sec.	11min. 3sec.
4	2^{20}	94 min.19 sec.	11min. 7sec.

Table C1. Simulation time for test circuits for qsim and Fraunhofer simulators.

As we can see from the table the simulation time for the Fraunhofer simulator is almost the same for all test circuits while the simulation time for qsim varies. The simulation time for qsim is much less for circuits with sparse quantum state vector. When the quantum system has *3* qubits not in superposition (or they can be in a dependent superposition as in the Bell states) the simulation times for both simulators are nearly equal. Qsim performance is behind for the circuits for which the state vector is not sparse.

This is the result of memory allocation / de-allocation and that can be fixed as we mentioned in the section 4.7.

Bibliography

[AG04] S. Aaronson, D. Gottesman, *Improved Simulation of Stabilizer Circuits*, Phys. Rev. A 70, 052328, 2004.

[BBC+93] C.H. Bennett, G. Brassard, C. Crépeau, R. Jozsa, A. Peres, W.K. Wootters, *Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels,* Phys. Rev. Lett. 70, pp.1895–1899, 1993.

[BBV+95] A. Barenco, C. H. Bennett, R. Cleve, D.P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, H. Weinfurter, *Elementary gates for quantum computation*, Phys. Review A, March 1995.

[Bea02] S. Beauregard, *Circuit for Shor's algorithm using 2n+3 qubits*, online archives quant-ph/0205095.

[Ben73] C. H. Bennett, *Logical reversibility of computation*, IBM J. Res. Dev., 17(6), 525–32, 1973.

[BEST96] A. Barenco, A. Ekert, K. Suominen, P. Torma, *Approximate Quantum Fourier Transform and Decoherence*, Physical Review A 54, pp. 139 – 146, 1996.

[BHL+05] C.H. Bennett, P. Hayden, D. Leung, P.W. Shor, A. Winter, *Remote preparation of quantum states, Information Theory*, IEEE Transactions, Vol. 51, Issue 1, Jan. 2005.

[BM03] S.S. Bullock, I.L. Markov, *Arbitrary two-qubit computation in 23 elementary gates*, Phys. Rev. A 68, 2003.

[BM04] S.S. Bullock, I.L. Markov, *Smaller Circuits for Arbitrary n-qubit Diagonal Computations*, Quantum Information and Computation, vol. 4, no. 1, pp. 27-47, January 2004.

[Bre03] G.K. Brennen, *An Observable Measure of Entanglement For Pure State of Multi-qubits Systems*, Quantum Infor. Comput. 3, 2003.

[CBL+05] J. Chiaverini, J. Britton, D. Leibfried, E. Knill, M. D. Barrett, R. B. Blakestad, W.M. Itano, J.D. Jost, C. Langer, R. Ozeri, T. Schaetz, D.J. Wineland, *Implementation of the semiclassical quantum Fourier transform in a scalable system*, Science 308, 997-1000, 2005.

[CEM98] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, *Quantum algorithms revisited*, Proc. R. Soc. London A, 454(1969), 339–354, 1998.

[CFH97] D. G. Cory, A. F. Fahmy, T. F. Havel, *Ensemble quantum computing by NMR spectroscopy*, Proc. Nat. Acad. Sci. USA, 94, 1634–1639, 1997.

[CHP04] J.L. Cole. L.C.L. Hollenberg, S. Prawer, *An algorithm for simulating the Ising model on a type-II quantum computer*, Computer Physics Communications 161, 18, 2004.

[CLR99] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, the MIT Press, 1999.

[Cop94] D. Coppersmith, *An approximate Fourier transform useful in quantum factoring*, IBM Research Report No. RC19642, 1994.

[CPH98] D. G. Cory, M. D. Price, T. F. Havel, *Nuclear magnetic resonance spectroscopy: An experimentally accessible paradigm for quantum computing*, Physica D, 120, 82–101, 1998.

[CW00] R. Cleve, J. Watrous, *Fast parallel circuits for the quantum Fourier transform*, Proc. of 41st Symp. on Foundations of Comp. Sci., 2000.

[Cyb01] G. Cybenco, *Reducing Quantum Computations to Elementary Unitary Operations*. Computing in Science and Engineering, pp 27-32 , March/April 2001.

[CZ00] J. I. Cirac, P. Zoller, *A scalable quantum computer with ions in an array of microtraps*, Nature, 404, 579–581, 2000.

[CZ95] J. I. Cirac, P. Zoller, *Quantum computations with cold trapped ions*, Phys. Rev. Lett., 74, 4091,1995.

[DB00] I.H. Deutsch, G.K. Brennen, *Quantum computing with neutral atoms in an optical Lattice*, Fortschr. Phys., 48(9-11), 925–943, 2000.

[Deu85] D. Deutsch, *Quantum theory, the Church-Turing principle and the Universal quantum computer*, Proceedings of the Royal Society of London, A 400, pp 97-117, 1985.

[DiV00] D. P. DiVincenzo, *The Physical Implementation of Quantum Computation*, Fortschritte der Physik, 48(9-11), pp.771-783, 2000.

[DiV94] D. P. DiVincenzo, *Two-bit gates are universal for quantum computation*, Phys. Rev. A, 1994.

[DiV96] D. P. DiVincenzo, *Quantum Gates and Circuits*, Phil. Trans. R. Soc. Lond. A, 1996.

[DKRS06] T. Draper, S. Kutin, E. Rains, K. Svore, *A Logarithmic-Depth Quantum Carry-Lookahead Adder*, to appear in Quantum Information and Computation, 2006.

[DN05] C.M. Dawson, M.A. Nielsen, *The Solovay-Kitaev Algorithm*, Online Archives, quant-ph/0505030.

[Dra00] T.G. Draper, *Addition on a Quantum Computer*, Online Archives quant-ph/0008033.

[EWS+03] J. Emerson, Y.V. Weinstein, M. Saraceno, S. Lloyd, D.G. Cory, *Pseudo-Random Unitary Operators for Quantum Information Processing*, Science, Vol. 302, p.2098, 2003.

[Fey82] R. Feynman, *Simulating physics with computers*, Keynotes in the International Journal of Physics, 1982.

[GC97] N. Gershenfeld, I. L. Chuang, *Bulk spin resonance quantum computation*, Science, 275, 350–356, 1997.

[GN96] RB Griffiths, C. Niu, *Semiclassical Fourier Transform for Quantum Computation*, Phys.Rev.Lett. 76 pp. 3228-3231, 1996.

[Gro96] L.K. Grover, *A fast quantum mechanical algorithm for database search*, Proceedings, 28th Annual ACM Symposium on the Theory of Computing, pp.212-219, 1996.

[Gro97] L. K. Grover, *Quantum mechanics helps in searching for a needle in a haystack*, Phys. Rev. Lett., 79(2), 325, 1997.

[Haa91] F. Haake, *Quantum Signature of Chaos*, Springer, New York, 1991.

[HBZ02] M. Hillery, V. Buzek, M. Ziman, *Probabilistic implementation of universal processor*, Phys. Rev A, vol. 65, 2002.

[HH00] L. Hales, S. Hallgren, *An improved quantum Fourier transform algorithm and applications*, Proc. of 41st Annual Symposium on Foundations of Comp. Sci., 2000.

[HHL04] A. Harrow, P. Hayden, D. Leung, *Superdense Coding of Quantum States*, Phys. Rev. Lett. 92, 2004.

[HHR+05] H. Häffner, W. Hänsel, C. F. Roos, J. Benhelm, D. Chek-al-kar, M. Chwalla, T. Körber, U. D. Rapol, M. Riebe, P. O. Schmidt, C. Becher, O. Gühne, W. Dür, R. Blatt, *Scalable multiparticle entanglement of trapped ions*, Nature 438, 643, 2005.

[HLS04] P. Hayden, D. Leung, P.W. Shor, A. Winter, *Randomizing quantum states: Constructions and applications*, Communications in Mathematical Physics, Springer Berlin-Heidelberg,Vol. 250, Number 2, Sept. 2004.

[HRC02] A.W. Harrow, B. Recht, I.L. Chuang, *Efficient Discrete Approximations of Quantum Gates*, Journal of Mathematical Physics, Vol. 43, Issue 9, pp. 4445-4451, Sept. 2002.

[Hsu02] E. Hsu, Quantum Computing Simulation Optimizations and Operational Errors on Various 2-qubit Multiplier Circuits, Online Archives quant-ph/0208113.

[JM98] J. A. Jones, M. Mosca, *Implementation of a quantum algorithm to solve Deutsch's problem on a nuclear magnetic resonance quantum computer*, J. Chem. Phys., 109, 1648, 1998.

[JMH98] J. A. Jones, M. Mosca, R. H. Hansen, *Implementation of a quantum search algorithm on a nuclear magnetic resonance quantum computer*, Nature, 393(6683), 344, 1998.

[Joz01] R. Jozsa, *Quantum Factoring, Discrete Logarithms, and the Hidden Subgroup Problem, Computing in Science and Engineering*, March/April 2001, pp. 34-43.

[JOZ01] R. Jozsa, *Quantum factoring, discrete logarithms and the hidden subgroup problem*, IEEE Computing in Science and Engineering, March/April 2001.

[Kay04] P.Kaye, *Reversible addition circuit using one ancillary bit with application to quantum computing*, Online archives quant-ph/0408173.

[Kit97] A.Y. Kitaev, *Quantum Computations: algorithm and error correction*, Russ. Math. Surv., 52(6): 1191-1249, 1997.

[KLM01] E. Knill, R. Laflamme, G. J. Milburn, *A scheme for efficient quantum computation with linear optics*, Nature, 409, 46–52, 2001.

[KSV02] A. Y. Kitaev, A. H. Shen, M. N. Vyalyi, *Classical and quantum computation*, Vol. 47 of Graduate Studies in Mathematics, American Mathematical Society, Providence, Rhode Island, 2002.

[Kun03] N. Kunihiro, *Practical running time of factoring by quantum circuits*, in Proc. Conference on Quantum Information Science, Sept. 2003.

[Lan61] R. Landauer, *Irreversibility and heat generation in the computing process*, IBM J. Res. Develop., pages 183-191, July 1961.

[LD98] Daniel Loss and David P. DiVincenzo, *Quantum computation with quantum dots*, Phys. Rev. A, 57, 120–126, 1998.

[MD03] D. Maslov, G. Dueck, *Improved Quantum Cost for n-bit Toffoli Gates*, IEEE Letters, vol. 39, issue 25, pp. 1790-1791, Dec. 2003.

[Mil89] G. J. Milburn, *Quantum optical Fredkin gate*, Phys. Rev. Lett., 62(18), 2124–2127, 1989.

[MMK+95] C. Monroe, D.M. Meekhof, B.E. King, W.M. Itano, D.J. Wineland, *Demonstration of a fundamental quantum logic gate*, Phys. Rev. Lett. 75, 4714,1995.

[MR03] K. Michielsen, H.D. Raedt, *QCE: A Simulator for Quantum Computer Hardware*, Turk. J. Phys., 27, pp. 1-29, 2003.

[MS99] K. Molmer, A. Sorensen, *Multiparticle entanglement of hot trapped ions*, Phys. Rev. Lett., 82, 1835–1838, 1999.

[MV05] M. Mottonen, J. Vartiainen, *Decompositions of general quantum gates*, Online archives quant-ph/0504100.

[MW02] D.A. Meyer, N.R. Wallach, *Global Entanglement in Multiparticle System*, J. of Math. Phys., 43, pp 4273, 2002.

[NC00] M.A. Nielsen, I.L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2000.

[NMI02] J. Niwa, K. Matsumoto, H. Imai, *General-Purpose Parallel Simulator for Quantum Computing*, Proceedings of Third International Conference, UMC 2002.

[OD98] K.M. Oberland, A. Despain, *A Parallel Quantum Simulator*, High Performance Computing, 1998.

[PGCZ95] T. Pellizzari, S.A. Gardiner, J.I. Cirac, P. Zoller, *Decoherence, continuous observation, and quantum computing: A cavity QED model*, Phys. Rev. Lett. 75, pp. 3788-3791, 1995.

[PK96] M.B. Plenio, P.L. Knight, *Realistic lower bounds for the factorization time of large numbers on a quantum computer*, Phys. Rev. A 53, 1996.

[PP00] S.Parker, M.B. Plenio, *Efficient factorization with a single pure qubit and logN mixed qubits*, Phys. Rev. Lett., 85, pp. 3049 – 3052, 2000.

[PST+99] M.D. Price, S.S. Somaroo, C.H. Tseng, J.C. Gore, A.F. Fahmy, T.F. Havel, D.G. Cory, *Construction and implementation of NMR quantum logic gates for two spin systems*, Journal of Magnetic Resonance,140pp 371-378, 1999.

[PZK98] M. Pozniak, K. Zyczkowski, M. Kus, *Composed Ensembles of Random Unitary Matrices*, J. Phys. A31, pp. 1059-1071, 1998.

[RAK+04] H. Rose, T. Abelmeyer-Maluga, M. Kolbe, F. Niehoster, A. Schram, *A web-based Simulator of Quantum Computing Processes*, Online Archives quant-ph/0406089.

[SC03] A.J. Scott, C.M. Caves, *Entangling Power of Quantum's Baker Map*, J. Phys. A: Mathematical and General, Issue 36, Sept. 2003.

[SHO+06] D. Stick1, W. K. Hensinger1, S. Olmschenk1, M. J. Madsen1, K. Schwab2, C. Monroe1, *Ion trap in a semiconductor chip*, Nature Physics 2, pp. 36-39, 2006.

[Sho94] P. Shor, *Algorithm for Quantum Computation: Discrete Logarithm and Factoring, Proceedings*, 35th Ann. Symp. on Foundation of Computer Science, pp. 124-134, 1994.

[Sho97] P. Shor, *Polynomial-Time Algorithm for Prime Factorization and Discrete Logarithms on a Quantum Computer*, SIAM J. Comp. 26(5), pp. 1484-1509, 1997.

[Sim94] D. Simon, *On the power of quantum computation*, In Proceedings, 35th Annual Symposium on Foundations of Computer Science, pp. 116–123, IEEE Press, Los Alamitos, CA, 1994.

[SPM03] V.V. Shende, A.K. Prasad, I.L. Markov and J.P. Hayes, *Synthesis of Reversible Logic Circuits*, IEEE Trans. on CAD 22, 710, 2003.

[Ste96] A. M. Steane, *Error correcting codes in quantum theory*, Phys. Rev. Lett., 77, 793, 1996.

[SZ02] M. O. Scully, M. S. Zubairy, *Cavity QED implementation of the discrete quantum Fourier transform*, Phys. Rev. A 65, 052324, 2002.

[Unr95] W.G. Unruh, *Maintaining coherence in quantum computers*, Phys. Rev. A 51, 992, 1995.

[VBE95] V. Vedral, A. Barenco, and A. Ekert, *Quantum Networks for Elementary Arithmetic Operations*, Phys Rev A, 1995.

[VMH03] G.F. Viamontes, I.L. Markov, J.P. Hayes, *Improving Gate-Level Simulation of Quantum Circuits*, Quantum Information Processing, 2 (5), pp. 347-380,

October 2003.

[VRMH03] G. F. Viamontes, M. Rajagopalan, I. L. Markov, J. P. Hayes, *Gate-Level Simulation of Quantum Circuits*, In Proc. of the Asia South Pacific Design Automation Conference (ASPDAC), pp. 295-301, Kitakyushu, Japan, January 2003.

[VSB+00] L.M.K. Vandersypen, M. Steffen, G. Breyta, C.S. Yannoni, M.H. Sherwood, I.L. Chuang, *Experimental Realism of an Order-Finding Algorithm with an NMR Quantum Computer*, Phys. Rev. Lett., Vol. 85, pp. 5452-5455, 2000.

[VSB+01] L.M.K. Vandersypen, M. Steffen, G. Breyta, C.S. Yannoni, M.H. Sherwood, I.L. Chuang, *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*, Nature, Vol. 414, pp. 883-887, Dec. 2001.

[Yao93] A. C. Yao, *Quantum circuit complexity*, Proc. of the 34th Ann. IEEE Symp. on Foundations of Computer Science, pp. 352–361, 1993.

[ZK94] K. Zyczkowski, M. Kus, *Random Unitary Matrices*, J. Phys. A: Math Gen. 27, pp. 4235 – 4245, 1994.

[ZS05] K. Zyczkowski, H. Sommers, *Average Fidelity between random quantum states*, Physical Review A 71, 2005.