

Van Huele JF

MODELING THE QUANTUM COMPUTER
ON THE
CLASSICAL COMPUTER

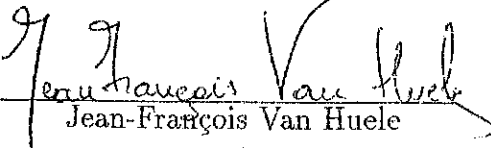
by

Damian P. Menscher

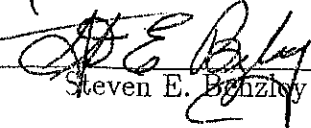
July 1997

Submitted to Brigham Young University in partial fulfillment
of graduation requirements for University Honors.

Advisor:


Jean-François Van Huele

Honors Dean:


Steven E. Benizley

Contents

| | |
|--|-----------|
| Contents | i |
| List of Figures | iii |
| List of Tables | iv |
| Abstract | v |
| Acknowledgements | vi |
| 1 Introduction | 1 |
| 2 The Classical Computer | 2 |
| 2.1 Development of the Computer | 2 |
| 2.2 Quantum Effects in Computers | 3 |
| 3 The Quantum Computer: Background | 4 |
| 3.1 Theory | 4 |
| 3.2 Advantages | 5 |
| 3.3 Challenges | 5 |
| 4 The Quantum Computer: Theory | 7 |
| 4.1 Qubits | 7 |
| 4.2 Conditional Operations | 8 |
| 4.3 Simple Gates | 8 |
| 4.3.1 FLIP Gate | 9 |
| 4.3.2 Controlled-NOT Gate | 9 |
| 4.3.3 Controlled-Controlled-FLIP Gates | 10 |
| 4.3.4 Other Simple Gates | 10 |
| 4.4 N-Bit Gates | 11 |
| 4.5 Networks | 11 |
| 5 Applications | 13 |
| 5.1 Shor's Algorithm | 13 |
| 5.2 Factoring 15 | 15 |
| 6 Experiments in Quantum Computing | 17 |
| 6.1 Theory | 17 |
| 6.2 Results | 18 |
| 6.3 Decoherence Problems and Remedies | 19 |

| | | |
|----------|--|-----------|
| 7 | Simulating the Quantum Computer | 20 |
| 7.1 | State Representation | 20 |
| 7.2 | Gate Representation | 21 |
| 7.2.1 | FLIP Gate | 21 |
| 7.2.2 | Controlled-NOT Gate | 22 |
| 7.2.3 | Controlled-Controlled-FLIP Gates | 22 |
| 7.3 | Networks | 23 |
| 7.4 | The Program | 23 |
| 7.5 | Results | 24 |
| 7.5.1 | Two-Bit Adder | 24 |
| 7.5.2 | Performance | 25 |
| 8 | Conclusions | 27 |
| | Appendices | 29 |
| A | Spin | 29 |
| A.1 | The Basics of Quantum Spin | 29 |
| A.2 | Mixed Spin States | 31 |
| B | The RSA Cryptosystem | 33 |
| B.1 | Theory | 33 |
| B.2 | Signaturization | 34 |
| B.3 | Security | 35 |
| C | The Program | 36 |
| C.1 | Complex.pas | 36 |
| C.2 | QCompute.pas | 37 |
| C.3 | Adder.q | 48 |
| C.4 | Adder2.q | 48 |
| D | An Interactive Session | 50 |
| | Bibliography | 54 |

List of Figures

| | | |
|-----|--|----|
| 4.1 | Pictorial representation of an arbitrary quantum gate acting on three qubits | 9 |
| 4.2 | Five simple gates for the quantum computer | 10 |
| 4.3 | Construction of the Triple-Controlled Not gate | 11 |
| 4.4 | Construction of the two-bit adder | 12 |
| 6.1 | Combined energy levels of two quantum particles before and after an applied electric field | 18 |
| 7.1 | Quantum computer simulator options | 24 |
| 7.2 | Construction of the Toffoli gate | 24 |

List of Tables

| | | |
|-----|--|----|
| 6.1 | Figure of merit \mathcal{M} for various proposed technologies | 19 |
| 7.1 | Gate time of two quantum computer simulators for the CN gate acting on various numbers of qubits | 26 |
| B.1 | Running times of various factoring methods | 35 |
| D.1 | Output of the two-bit adder for non-classical input | 53 |

Abstract

Information is traditionally stored in computers as 1s and 0s called bits and manipulated by classical gates. Similarly, information can be stored at the quantum level in *qubits* and manipulated by quantum gates. Programs can then be written for quantum computers by assembling networks of these quantum gates. If realized, such programs would be able to achieve an exponential speedup over classical algorithms by taking advantage of quantum effects such as interference, superposition, and entanglement.

Because of the difficulty of maintaining quantum coherence, quantum networks will not be operational for some time. I propose to simulate these networks on a classical computer. Although such a simulator does not produce an exponential speedup, it allows the user to test quantum gate networks before implementing them in hardware. The operation of an n -bit quantum computer simulator I have developed is described. The advantages and limitations of the simulator, as well as possible applications, are then discussed.

Acknowledgements

Many people have helped me with this project. First I would like to thank my advisor, Jean-François Van Huele, for introducing me to the subject of quantum computation and providing me with many valuable insights and criticisms. I would also like to thank Zerubbabel Johnson, whose parallel research has allowed us many useful discussions and comparisons. Third, I am thankful to all of my professors and friends in the BYU physics department who have provided me with feedback on my research. Finally, I would like to thank my parents, who made all of this possible.

Chapter 1

Introduction

The quantum computer has received much interest both in the specialized literature [DBE95, M⁺95] and in the general press [Fol95, Bra95] primarily because of its potential for factoring numbers in polynomial time [Sho94]. This work is a study of some simple gates of the quantum computer, on how to represent them, and on how to compute their effects by using the power of a classical computer. My efforts to understand the inner workings of the quantum computer have culminated in the development of an interactive quantum computer simulator.

I show in Chapter 2 that the classical computer, although it has served us well, will not be able to improve much further. In Chapters 3 and 4 I explain what is meant by a quantum computer and how it works. Next, in Chapter 5, I discuss how this computer can be used. That chapter focuses on Shor's algorithm for factoring. Attempts to construct a quantum computer are described in Chapter 6 along with difficulties and some possible remedies. Chapter 7 presents one representation of the inner workings of the quantum computer and describes in detail my contribution to the area, focusing on a quantum computer simulator I designed. I conclude with some suggestions for further work in Chapter 8. In that chapter I describe the difficulties I encountered and how I overcame them. I also comment on aspects of the problem that I would approach differently now that I understand the topic more fully.

Several appendices have been included for completeness. Appendix A describes the quantum property *spin*, which is useful to store quantum information. Appendix B discusses the RSA cryptosystem, a widely-used method of cryptography which may be rendered obsolete if a quantum computer is ever constructed. The Pascal source code for my quantum computer simulator `QCompute` is presented in Appendix C along with some sample gate-network files. Finally, Appendix D shows an interactive session with the simulator.

Chapter 2

The Classical Computer

2.1 Development of the Computer

Society has undergone fundamental changes with the introduction of the computer. Although computers started out as large, slow machines that could only accomplish the simplest of calculations, they have rapidly improved both in power and in versatility. These changes have been accomplished through several improvements, including the invention of the transistor to replace the vacuum tube and later the method of etching transistors onto a wafer of silicon to create more compact circuits. Because of advances in technology and in our ability to create more reliable integrated circuits (ICs) we have seen an exponential growth rate in the power of computers. This trend, commonly called Moore's Law, states that the number of transistors per chip, and therefore the computing power, doubles roughly every 18 months [Gro96, Met96].

There is, however, a limit to the possible improvements we can make on the common digital computer. We have achieved much of the speed we see today because of the ability to etch thinner and thinner traces on a piece of silicon. For example, the current traces are only $.35\mu\text{m}$ across, and Intel already has plans for chips with traces $.25\mu\text{m}$ wide [Met96]. Because atoms are on the order of an Angstrom across, we are already down to the point where our traces are only about a thousand atoms across. Also, we are running them at frequencies up to 200MHz, which means that relatively few electrons can travel down each path. Increasing the power of the computer means that we must have more circuits, so that we must either make the traces smaller or the physical size of the computer bigger. Any increase in the speed of the computer means that we must send shorter pulses of electricity through these traces and therefore pass fewer electrons through these traces. Because the traces must be at least one atom

wide and we must always send at least one electron down each trace to represent the digital “on” state, there is a theory-imposed limit on the maximum power and speed of any classical computer [Tof80]. The restrictions on size and speed lead Intel to believe they will reach this limit by the year 2017 [Met96]. Further progress needs to be sought in a different direction.

2.2 Quantum Effects in Computers

Physics teaches us that, at the microscopic level, there is a limit to the amount of knowledge we can obtain about a system. In particular, the Heisenberg uncertainty principle tells us that we cannot simultaneously determine the position and the momentum of a quantum particle with unlimited precision [Gas96]. Because individual electrons are quantum particles, nonclassical, quantum phenomena will occur when we attempt to construct a computer that uses sufficiently low numbers of electrons to carry information. These effects are leading physicists and theoretical computer scientists to develop a completely new and different type of computer based on quantum mechanics called a *quantum computer* [Fey86, Bar96].

Because they operate on quantum particles such as electrons or photons, quantum computers have several distinct advantages over their classical counterparts. Some of these advantages are derived from the properties of wave mechanics [Fow89] such as *superposition*, *interference*, and *entanglement* [Ben95]. Superposition allows the computer to work on a problem by taking parallel paths, and then combining those paths to get an answer at the end. This way, the computer can attack many aspects of a problem at the same time. When the answers are superposed, interference phenomena will cause some results to be enhanced while causing others to be reduced or even eliminated. Entanglement ensures that all aspects of the problem are taken into account at every step. Because of these three properties, quantum computers can operate on quantum bits of information (called *qubits*) in completely new ways [DJ92, Sim94]. In particular, quantum computation presents a richer structure than classical computation or even classical parallel processing. The advantages of quantum computation have already been applied to the problems of discrete log and factoring (see Chapter 5) [Sho94], both of which are very important to cryptographic systems such as RSA [Bra94, Cle89]. Appendix B gives a description of RSA public-key cryptography.

Chapter 3

The Quantum Computer: Background

3.1 Theory

A quantum computer is based on the principles of quantum mechanics. The computer operates on a quantum system, which may be a single electron, an entire atom, or even a quantum dot. Because of the nature of quantum mechanics, we must describe a quantum particle in terms of probabilities. This is normally accomplished by representing a particle by its wave function — a normalized, complex wave which completely describes the state of the particle [Gas96]. The wave function ψ is related to the probability density P through its modulus squared, so

$$P = |\psi|^2. \quad (3.1)$$

Any actions we take can be expressed as operators acting on this wave function. The wave function may in turn consist of a superposition of several *basis states*, or a set of states in terms of which any arbitrary state may be defined. The heart of quantum computation is therefore to perform conditional actions on the basis states of a quantum particle.

Several mathematical formalisms have been developed for quantum mechanics. I will use one formalism, called Dirac notation [Gas96], to represent wave functions, operators, and measurements. In this notation, a wave function may be represented as $|\psi\rangle$ and an operator on it may be represented as $\hat{U}|\psi\rangle$. A measurement may be accomplished by performing the operation $\langle\psi|\hat{U}|\psi\rangle$. We can also write $|\psi\rangle$ as a column vector, so $\hat{U}|\psi\rangle$ becomes a square matrix multiplied by a column vector, thus

resulting in another column vector. The $\langle\psi|$ is written as a row vector corresponding to the Hermitian adjoint (the complex conjugate of the transpose) of the column vector $|\psi\rangle$. I will use this formalism for the remainder of the discussion and refer the reader to a standard text on quantum mechanics for a more comprehensive treatment of Dirac notation [Gas96].

Using this formalism, the state of a quantum system may be represented as a complex superposition of the several basis states available to it. Thus we may represent the spin state of one electron as

$$|\psi\rangle = \alpha|\downarrow\rangle + \beta|\uparrow\rangle \quad (3.2)$$

where $|\downarrow\rangle$ represents spin down and $|\uparrow\rangle$ represents spin up. (See Appendix A for a discussion of spin.) When relating quantum systems to the quantum computer, we choose systems that have only two possibilities (i.e., a two-level system), which we label $|0\rangle$ and $|1\rangle$. This allows us to use a common notation whether the quantum system is implemented using the spin states of an electron, energy levels in an atom, or some other two-level quantum system. This also makes the correspondence with classical bits easier for us to understand.

3.2 Advantages

In a classical computer, a bit (the elementary unit of information) must be either a 0 or a 1. In a quantum computer, a similar system is used, where each quantum bit, or *qubit*, is in a complex superposition of the $|0\rangle$ and $|1\rangle$ basis states, where the α and β in Eq. 3.2 are complex numbers which represent the degree to which the quantum system is in each state. Because the qubit is in a superposition of both states, it may represent both a classical 0 and a classical 1 simultaneously. This allows us to carry out multiple calculations at the same time, thus resulting in an *exponential* speedup [DJ92, Sim94]. Because of theorems concerning the universality of gates it turns out that it is only necessary to be able to perform one- and two-bit gate operations to perform all operations [Bar95].

3.3 Challenges

There are some problems associated with quantum computation. The most difficult challenge is imposed by the same property which makes the computer so powerful

— the qubits can be in an entangled state. When the system is entangled, it is not possible for us to measure the state of one qubit without affecting the state of the entire system. Therefore any operations we perform must act on the system as a whole, and measurement must be reserved until all calculations have been completed.

Another challenge is that constructing an actual quantum computer will be difficult, if not impossible. Quantum states may decohere, or fall out of their entangled state, if any interaction is permitted with the outside world. Therefore it is necessary to trap the quantum particle in such a way that it will be completely isolated from the outside world. Isolating such a large system is an extremely difficult task for the experimental physicist.

Chapter 4

The Quantum Computer: Theory

4.1 Qubits

The basic component of information in a classical computer is commonly referred to as a *bit*. This bit, which can store either a 0 or a 1, is stored in a classical computer as a voltage, where, for example, 0 V may correspond to the binary 0, and 5 V may correspond to the binary 1. Any operations which are performed act on these bits, or in physical terms, on all of the electrons of which these bits are composed.

Similarly, for a quantum computer, the smallest component of information is called a *qubit*, and can take values corresponding to a complex superposition of the $|0\rangle$ and $|1\rangle$ basis states. This information is stored in a two-level quantum system. For an electron, it might correspond to the spin-down and spin-up states. In an atom, it might correspond to the ground state and the excited state. The actual implementation is not important here — all that is important is that the two states form a basis for the quantum system.

As in the classical computer, we may wish to store more complex information than simply one qubit. To do this, we may construct a *register* which contains several qubits. The register may be interpreted in binary such that an N -bit register can store any of the numbers 0 through $2^N - 1$. The notation used to represent these may take the form of either 6 or 110_2 (the number 6 written in base 2) on a classical computer. Similarly, on a quantum computer we may use the notation $|1\rangle|1\rangle|0\rangle$, $|110\rangle$, or $|6\rangle$, depending on which aspect of the information we wish to emphasize at the time. For example, writing $|1\rangle|1\rangle|0\rangle$ would generally be interpreted as three separate registers. The notation $|110\rangle$ helps to emphasize that the three qubits in question are all part of the same register. Finally, the notation $|6\rangle$ emphasizes that it is the data with which

we are concerned, and the bitwise representation is less important. When operating on these registers we should keep in mind that, just as in the classical computer, we only operate on one qubit at a time. Changes to the entire register are accomplished by performing successive operations on all qubits in the register.

Another important concept is that of entanglement, or the ability of one qubit to be intrinsically correlated with another qubit. For example, we might have a register in the state $\alpha|00\rangle + \beta|11\rangle$. If we were to measure the first qubit and obtain a value of 1, then we would know that the second qubit must also have a value of 1. Because we have determined information about the quantum state, we say that the state collapses, or reduces, to a simpler form. In the example above, the state would collapse to be $\gamma|11\rangle$, where $|\gamma|^2 = 1$.

4.2 Conditional Operations

In order to have a useful computer, it is necessary to be able to perform conditional operations, or operations for which one or more inputs may control whether the operation is performed. These operations may be as simple as constructing an AND gate, where the target bit is set to 1 if and only if both control bits contain the value 1. On a quantum computer all conditional operations are based on conditional quantum physics, in which one system undergoes a coherent evolution that depends on the state of another system [BDEJ95]. For example, we may want to flip the spin state of an electron if and only if it is in the excited state of an atom. This can be accomplished through the use of an electric field and precision lasers as described in Section 6.1.

4.3 Simple Gates

Like a classical computer, a quantum computer may be constructed by combining several simple logic gates. When performing quantum operations, all calculations are reversible, i.e., no information will be lost. This is a consequence of the unitarity of time evolution in quantum mechanics, which is related to the conservation of probabilities. Unlike classical gates, therefore, a gate must have the same number of outputs as inputs. A gate is usually represented pictorially as a box that has wires going in and out of it, as in Figure 4.1.

4.3.1 FLIP Gate

The FLIP gate is the most commonly used one-bit gate. This gate, which is the quantum equivalent of the classical NOT gate, flips the value of its argument. Classically, this gate just changes a 0 to a 1 and a 1 to a 0. The quantum mechanical analog is similar — the gate changes the $|0\rangle$ state to a $|1\rangle$ state and the $|1\rangle$ state to a $|0\rangle$ state. Because a quantum state can be in both the $|0\rangle$ and $|1\rangle$ states at the same time, the gate must actually perform the operation

$$\alpha|0\rangle + \beta|1\rangle \longrightarrow \beta|0\rangle + \alpha|1\rangle . \quad (4.1)$$

Thus it interchanges the complex coefficients of the two basis states. This gate acts on only one qubit, and is represented pictorially as in Figure 4.2(a).

4.3.2 Controlled-NOT Gate

The Controlled-NOT (CN) gate is the quantum analog of the classical exclusive-OR (XOR) gate. This gate flips the target qubit if and only if the control qubit is a 1 (or to the extent that the control qubit is a 1). Thus if we perform the CN operation on a two-bit system with the first qubit controlling the second qubit, we perform the operation

$$B \longrightarrow A \oplus B , \quad (4.2)$$

where \oplus is the classical XOR operation and signifies addition modulo 2. This achieves the transformation

$$\alpha|0\rangle|0\rangle + \beta|0\rangle|1\rangle + \gamma|1\rangle|0\rangle + \delta|1\rangle|1\rangle \longrightarrow \alpha|0\rangle|0\rangle + \beta|0\rangle|1\rangle + \delta|1\rangle|0\rangle + \gamma|1\rangle|1\rangle . \quad (4.3)$$

A pictorial representation of this gate is shown in Figure 4.2(b) where \bullet is placed on the control line and \oplus is placed on the controlled line.

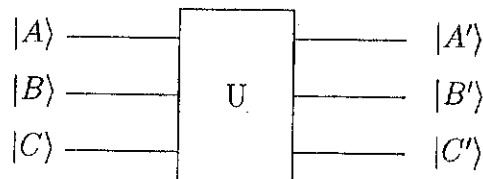


Figure 4.1: Pictorial representation of an arbitrary quantum gate acting on three qubits.

4.3.3 Controlled-Controlled-FLIP Gates

The Controlled-Controlled-FLIP gates are gates with two control bits and one target bit. There are two varieties of the Controlled-Controlled-FLIP gate. The first, the Controlled-Controlled-NOT (CCN) gate, is similar to the CN gate in that it flips the target qubit if and only if both control bits are set to 1. Thus if it is acting on the three bits A , B , and C , where A and B control C , it will change C according to the operation

$$C \longrightarrow (A \wedge B) \oplus C, \quad (4.4)$$

where \wedge is the classical AND operation. This gate, frequently called the Toffoli gate after its inventor, is shown in pictorial form in Figure 4.2(c).

The second gate, the Deutsch gate, works like the Toffoli gate except that it takes an argument, θ , where $\sin \theta$ specifies the degree to which the controlled qubit is flipped if both controls are in the $|1\rangle$ state. Hence it performs the operation

$$\left. \begin{array}{l} |1\rangle|1\rangle(\alpha|0\rangle + \beta|1\rangle) \longrightarrow |1\rangle|1\rangle((i\alpha \cos \theta + \beta \sin \theta)|0\rangle + (\alpha \sin \theta + i\beta \cos \theta)|1\rangle) \\ |1\rangle|0\rangle(\alpha|0\rangle + \beta|1\rangle) \\ |0\rangle|1\rangle(\alpha|0\rangle + \beta|1\rangle) \\ |0\rangle|0\rangle(\alpha|0\rangle + \beta|1\rangle) \end{array} \right\} \text{Unchanged.} \quad (4.5)$$

A representation of this gate is shown in Figure 4.2(d).

4.3.4 Other Simple Gates

There are countless other simple gates, only a few of which are commonly used. In general, a gate will operate on one or more qubits. Because the qubits are generally in an entangled state, a gate will change all of the qubits it acts on — i.e., there may not be a specific control or target qubit. Gates of this type are represented as in Figure 4.2(e).

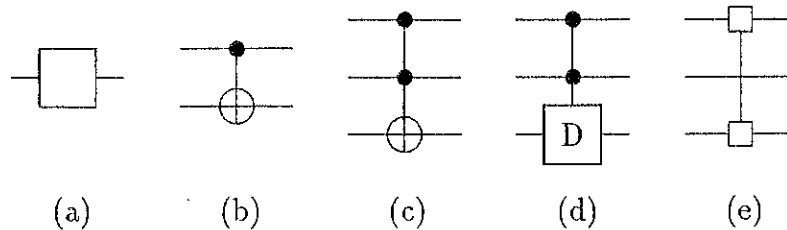


Figure 4.2: Five simple gates for the quantum computer: (a) the FLIP gate, (b) the CN gate, (c) the Toffoli gate, (d) the Deutsch gate, (e) an arbitrary 2-bit gate

4.4 N-Bit Gates

Gates are not restricted to be small like the ones described above in Section 4.3. For a useful computation, it will be necessary to have several large gates. Because one- and two-bit gates form a universal set, large gates can always be constructed from the smaller gates [Bar95]. For example, it is shown that a gate with three control qubits flipping a single target bit can be constructed from three CCN gates (using one qubit for working space) in Figure 4.3 [B⁺95].

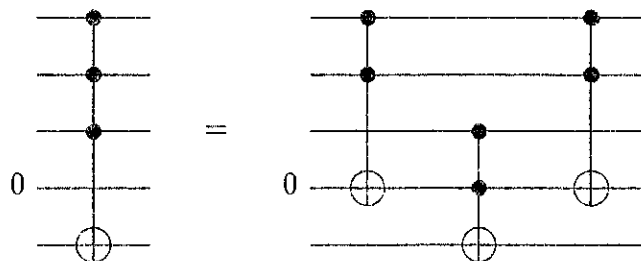


Figure 4.3: Construction of the Triple-Controlled Not gate

4.5 Networks

In order to perform useful computations, we must combine several gates of all sizes in a way such that we can obtain the desired result [Deu89]. Constructing such a network is a complicated process. Consider the two-bit adder. The purpose of this adder is to take in the binary numbers AB and CD and add them to get the result EFG. The construction of the required network using only two- and three-bit gates is shown in Figure 4.4. We see that it is complicated for us to perform even this relatively simple task. This algorithm is actually quite close to the one developed for the classical computer. On the classical computer, the implementation of this network requires the correct wiring of 24 transistors and 3 resistors [Poo92]. On the quantum computer, it would likely be implemented on 7 qubits using a large number of precision lasers and electric fields.

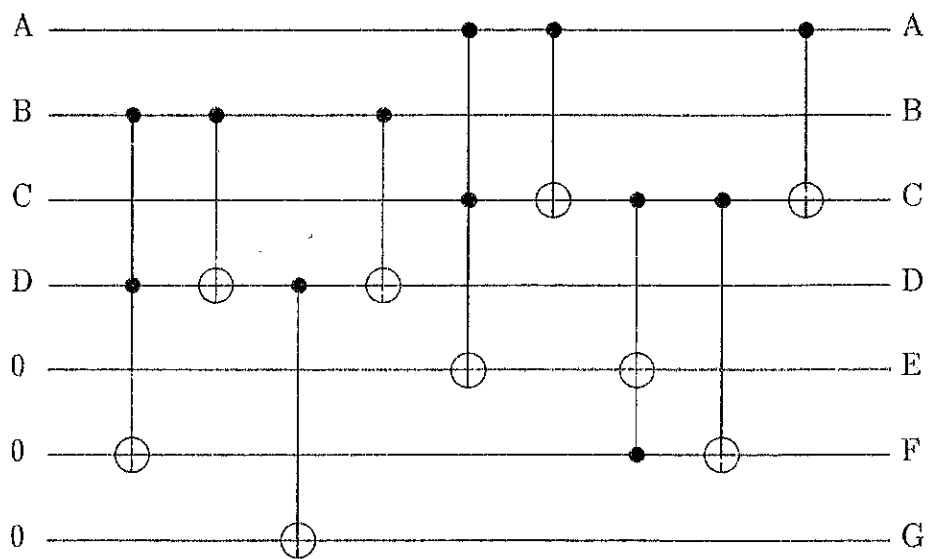


Figure 4.4: Construction of the two-bit adder, which adds the binary numbers $AB + CD = EFG$.

Chapter 5

Applications

The quantum computer has received much publicity because of its predicted performance in certain applications. Because of the exponential speedup available, several classically intractable calculations can be performed in polynomial time on a quantum computer. Two algorithms utilizing this speedup have been developed for the quantum computer so far: to solve the problems of the discrete log and of factoring [Sho94]. Because of the importance of factoring large numbers to encryption (see Appendix B), the factoring algorithm has become the more popular of the two, and is discussed below.

5.1 Shor's Algorithm

The function

$$f_{a,N}(x) = a^x \bmod N, \quad (5.1)$$

where a is any number coprime with N , is periodic. This means that, for a given a and N , if we use the numbers $0, 1, 2, \dots$ as the x inputs, the output $f(x)$ will eventually begin to repeat itself. One can show that factorization is equivalent to finding the period, or number of values of x between repetitions, of this function. Once the period r is known, we can find the two factors of N as

$$\gcd(a^{r/2} \pm 1, N) \quad (5.2)$$

just as in Pollard's $p - 1$ method for factorization [And94].

Can we find the period of this sequence and factor N in polynomial time? A polynomial-time factoring algorithm was designed for the quantum computer by Peter Shor in 1994 [Sho94]. This algorithm is described below [Bar96].

1. We start by preparing a state of the form

$$\frac{1}{2^L} \sum_{x=0}^{2^{2L}-1} |x\rangle|0\rangle \quad (5.3)$$

where $L \geq \lceil \log_2 N \rceil$.

2. Now that we have the computer in a complex superposition of all necessary states we perform the operation $f_{a,N}$ described in Eq. 5.1 on the state to get

$$\frac{1}{2^L} \sum_{x=0}^{2^{2L}-1} |x\rangle|f_{a,N}(x)\rangle. \quad (5.4)$$

This single operation calculates the result of $f_{a,N}$ for all $2^{2L} \approx 2^{2\log_2 N} = N^2$ values of x .

3. We now measure the second register. As with all measurements, this forces the second register's wave function to collapse from a superposition of several values down to a single value. In addition, because the registers are entangled, the first register's wave function must also undergo a collapse, to include only those values which could have given the result measured by the second register. Because the function $f_{a,N}$ is periodic, the values of which the first register consists will be separated by integer multiples of the period, which we will denote r . It is this period that we are ultimately after, so we can throw away the information we gained when we measured the second register, as it is now no longer necessary for our purposes.
4. Notice that although we know that the first register contains the superposition of a sequence of numbers $l, l+r, l+2r, \dots$ we can't directly obtain the period, r , through a measurement because we still don't know the offset, l . If we could do this experiment many times and measure the same value for the second register, then we could get different values here and infer the value of r . In general this will not be possible because the sequence may have a very large period, thus making the probability of measuring the same value for the second register twice exponentially small. If we were to perform the experiment repeatedly until we did manage to do so, we would have returned to an exponential time requirement for the algorithm. Instead, to eliminate the offset, we may perform a discrete Fourier transform (DFT) on the first register. Because a Fourier transform loses phase information, the offset is eliminated. We can now measure the first state to find an integer multiple of the period. If we do the entire procedure two or

more times and obtain periods of r_1, r_2, \dots, r_n , we can infer the actual period r as $\gcd(r_1, r_2, \dots, r_n)$ with reasonable accuracy.

5. Finally, we can calculate the two factors of N as

$$\gcd(a^{r/2} \pm 1, N) \quad (5.5)$$

in polynomial time on a classical computer because the greatest common denominator can be found in polynomial time.

5.2 Factoring 15

As an example of how this method works, let us factor the number 15. The steps are as follows:

1. We start by preparing the necessary state. Here $N = 15$ so $L = \lceil \log_2 15 \rceil = 4$. Thus our starting state is

$$\frac{1}{16} \sum_{x=0}^{255} |x\rangle|0\rangle. \quad (5.6)$$

2. We now need to choose a such that a is coprime with N (otherwise we factor N as $\gcd(a, N)$). Let us choose a to be 7. Now we perform the operation $f_{a,N}$ described in Eq. 5.1 to get the state

$$\frac{1}{16} (|0\rangle|1\rangle + |1\rangle|7\rangle + |2\rangle|4\rangle + |3\rangle|13\rangle + |4\rangle|1\rangle + |5\rangle|7\rangle + \dots + |255\rangle|13\rangle). \quad (5.7)$$

Notice that the result of this function is periodic, namely that for $x = 0, 1, 2, 3, 4, 5, 6, 7, 8, \dots$ the result is $1, 7, 4, 13, 1, 7, 4, 13, 1, \dots$.

3. Now we measure the second register's wave function. It doesn't really matter what the result of our measurement is, but let us assume that we measured a value of 4. Then the second register takes on a value of 4 and the first register collapses to contain only those values which could have produced a result of 4. Those values are $x = 2, 6, 10, \dots, 254$. Hence the first register is now in the state

$$\frac{1}{8} (|2\rangle + |6\rangle + |10\rangle + \dots + |254\rangle). \quad (5.8)$$

4. We want to find out what the period r is of this wave function (Eq. 5.8). Just making a measurement won't help us though, because obtaining a value such as 54 doesn't tell us anything about the period unless we know what the offset l is.

Therefore we now perform a discrete Fourier transform on the register. This has the effect of eliminating the offset. Now we can measure the register and know that the answer we obtain is an integer multiple of r . If we repeat the procedure and find the greatest common denominator of our final measurements, we will know the value of r . In this example, the offset is $l = 2$ and the period is $r = 4$.

5. Now we can find the factors of 15 as

$$\gcd(a^{r/2} \pm 1, N) = \gcd(7^{4/2} \pm 1, 15) = \gcd(48, 15), \gcd(50, 15) = 3, 5. \quad (5.9)$$

Although factoring 15 is a simple example, it uses the same procedure that would be used to factor larger numbers. It is especially interesting to note that this algorithm can factor large numbers just as rapidly as small numbers until step 5. The computer used to do so, however, must have enough qubits to store the large number.

Chapter 6

Experiments in Quantum Computing

6.1 Theory

Because it is universal with one-bit gates [Bar95], the Controlled-NOT (CN) gate has received much attention. In this gate, one qubit (the target) is flipped if and only if another (the control) is on. Thus it transforms the state $|\epsilon_1\rangle|\epsilon_2\rangle$ to $|\epsilon_1\rangle|\epsilon_1 \oplus \epsilon_2\rangle$ where \oplus signifies addition modulo 2. Several methods have been used to implement this gate. Some of the methods include:

- Ramsey atomic interferometry, using a photon as the control qubit and an atom as the controlled qubit [BDEJ95]
- Selective driving of optical resonances of two qubits undergoing a dipole-dipole interaction [BDEJ95] (described below)
- Laser-cooled trapped ions in which the qubits are associated with internal states of the ions [M⁺95]

The most successful method to date is the third one because it is the easiest to implement. According to this method, one qubit is stored as the energy level of the atom, where $|0\rangle$ is the ground state and $|1\rangle$ is the excited state. The other qubit is stored in the spin state, which can be down ($|\downarrow\rangle$) or up ($|\uparrow\rangle$). The controlled-not operation has been achieved using the two $^2S_{1/2}$ hyperfine ground states of a single $^9\text{Be}^+$ ion [M⁺95].

An alternate implementation, based on the second method listed above, is to use two quantum dots in a static electric field \mathbf{E}_0 as the two qubits (Figure 6.1). If the

first qubit spans a larger energy gap than the second qubit, then the ordering of the states will be $|0\rangle|0\rangle$, $|0\rangle|1\rangle$, $|1\rangle|0\rangle$, $|1\rangle|1\rangle$. There will be allowed transitions between them of frequency ω_1 and ω_2 as shown in the figure. Normally we cannot control whether we cause a transition between $|0\rangle|0\rangle$ and $|0\rangle|1\rangle$ or a transition between $|1\rangle|0\rangle$ and $|1\rangle|1\rangle$ because the required energy for the transition, $\hbar\omega_2$, is the same for both transitions. In the presence of an electric field, however, the quantum Stark effect causes all of the levels to shift by an amount $\pm\hbar\bar{\omega}$ as shown in Figure 6.1 [Gas96]. With the presence of this electric field, it is therefore possible to selectively stimulate only the transition $|1\rangle|0\rangle \longleftrightarrow |1\rangle|1\rangle$ with light of frequency $\omega_2 + \bar{\omega}$ [BDEJ95].

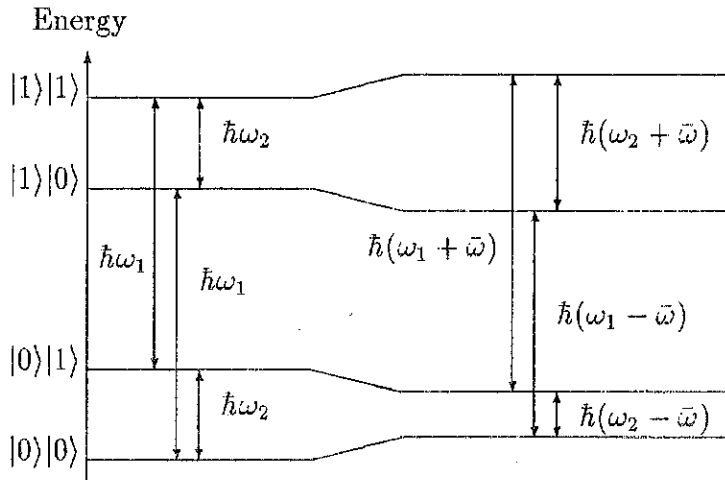


Figure 6.1: Combined energy levels of two quantum particles before and after an applied electric field.

6.2 Results

One group has demonstrated the Controlled-NOT gate acting on a two-qubit system [M⁺95]. In their experiment, the first qubit is the energy level of an electron in a ${}^9\text{Be}^+$ atom and the second qubit is the spin state of that electron. The desired transition, $|1\rangle|\downarrow\rangle \longleftrightarrow |1\rangle|\uparrow\rangle$ was singled out by using a sequence of three precise laser frequencies. The state of the beryllium atom can be detected by pushing one spin state with a laser, while leaving the other spin state alone. In so doing, the two states of the atom can be spatially separated [SS96a, Win96, Lev97]. The results are promising. Accuracies of above 80% for the CN gate starting in all four basis states have been achieved [M⁺95].

6.3 Decoherence Problems and Remedies

A major challenge arises from decoherence. Any interaction with the environment can cause the quantum state to decohere into a mixture of classical states, thereby losing all information in the calculation. Two strategies have been proposed to combat this:

- The first strategy is to minimize decoherence. The relaxation time τ of a qubit is the amount of time the quantum system can be expected to last before decohering into a classical state. The operation time t is the time it takes to perform one gate operation. Then $\mathcal{M} \equiv \tau/t$ provides a figure of merit of how stable the computer is, i.e., how many gate operations we can expect to perform before the state decoheres. The required \mathcal{M} scales like N^3 for Shor's algorithm [HR96], so a larger \mathcal{M} would allow us to factor larger numbers. Researchers are working hard to improve the ratio τ/t . Unfortunately, there is a theoretical limit on how large \mathcal{M} can become for each technology. Table 6.1 shows this ratio for various proposed quantum systems.

| Quantum system | t | τ | \mathcal{M} |
|------------------------|------------|------------|---------------|
| Electrons Au | 10^{-14} | 10^{-8} | 10^6 |
| Electrons GaAs | 10^{-13} | 10^{-10} | 10^3 |
| Electron quantum dot | 10^{-6} | 10^{-3} | 10^3 |
| Electron spin | 10^{-7} | 10^{-3} | 10^4 |
| Mössbauer nucleus | 10^{-19} | 10^{-10} | 10^9 |
| Nuclear spin | 10^{-3} | 10^4 | 10^7 |
| Optical cavities | 10^{-14} | 10^{-5} | 10^9 |
| Superconductor islands | 10^{-9} | 10^{-3} | 10^6 |
| Trapped ions - In | 10^{-14} | 10^{-1} | 10^{13} |

Table 6.1: Figure of merit \mathcal{M} for various proposed technologies. Values taken from [Bar96, DiV95].

- Another method is to watch out for decoherence and account for it. This *watch-dog* strategy works by encoding a $|0\rangle$ as $|000\rangle$ and a $|1\rangle$ as $|111\rangle$ so if any one of the three bits in the register should decohere, the change could be detected and corrected [HR96]. One can show that once \mathcal{M} reaches some threshold value — somewhere between 10^4 and 10^8 — stable computation will be possible using these methods [MW96].

Because neither the theoretical requirements nor our experimental abilities are accurately known, and because of the importance of the possible applications, the possibility of stable computation remains a hotly debated topic [HR96, MW96].

Chapter 7

Simulating the Quantum Computer

Because of the difficulty of actually constructing a quantum computer, I have turned my attention toward simulating the actions of a quantum computer on a classical computer. Although all gate operations suffer an exponential slowdown (so there is no increase in speed in the simulation) there is much to be learned about how quantum computers operate. Advantages over the experimental approach include the ease of use and the ability to monitor changes in the entangled state without disrupting it. This will be especially useful in the development stage of a quantum computer because it allows us to test logic designs before implementing them in hardware.

7.1 State Representation

The first in the construction of the simulator was to develop appropriate representations of both the entangled state and any gate operations one might wish to perform on that state.

I start from the three qubits

$$\begin{aligned} |A\rangle &= \alpha_0|0\rangle + \alpha_1|1\rangle, \\ |B\rangle &= \beta_0|0\rangle + \beta_1|1\rangle, \\ |C\rangle &= \gamma_0|0\rangle + \gamma_1|1\rangle, \end{aligned} \tag{7.1}$$

where the coefficients α , β , γ are complex. I multiply the coefficients together to

obtain the entangled state-vector

$$\mathbf{V} = \begin{pmatrix} \alpha_0\beta_0\gamma_0 \\ \alpha_0\beta_0\gamma_1 \\ \alpha_0\beta_1\gamma_0 \\ \alpha_0\beta_1\gamma_1 \\ \alpha_1\beta_0\gamma_0 \\ \alpha_1\beta_0\gamma_1 \\ \alpha_1\beta_1\gamma_0 \\ \alpha_1\beta_1\gamma_1 \end{pmatrix}. \quad (7.2)$$

Note that the 2^N basis states are used here and can be numbered from 0 to $2^N - 1$ starting from the top.

7.2 Gate Representation

A gate is simply an operator which acts on a state. Thus with the state represented as a vector, we must represent the gate as a matrix. Thus operations are defined by the matrix equation

$$\mathbf{M}\mathbf{V} = \mathbf{V}'. \quad (7.3)$$

Because \mathbf{V} and \mathbf{V}' are vectors in the same space (i.e., they contain the same amount of information) the matrix \mathbf{M} must be square.

7.2.1 FLIP Gate

Consider the case where we want to change the state

$$\alpha|0\rangle + \beta|1\rangle \quad (7.4)$$

to the state

$$\beta|0\rangle + \alpha|1\rangle. \quad (7.5)$$

We do so by multiplying by an appropriate matrix as follows:

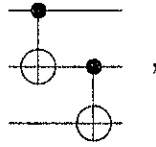
$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}. \quad (7.6)$$

This gate has effectively performed the FLIP operation on the qubit. Thus whenever we wish to perform a FLIP operation, we can use the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}. \quad (7.7)$$

7.3 Networks

Individual gates require the computer to keep track of only those qubits involved in that particular gate. If, however, we wanted to perform an operation such as



we would need to keep track of all three bits in the computer at all times, even though we are only using two at any given time. Hence if we want to flip the second qubit in a 3-qubit system we must use the entangled state of all three. Thus we perform the operation

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha_0\beta_0\gamma_0 \\ \alpha_0\beta_0\gamma_1 \\ \alpha_0\beta_1\gamma_0 \\ \alpha_0\beta_1\gamma_1 \\ \alpha_1\beta_0\gamma_0 \\ \alpha_1\beta_0\gamma_1 \\ \alpha_1\beta_1\gamma_0 \\ \alpha_1\beta_1\gamma_1 \end{pmatrix} = \begin{pmatrix} \alpha_0\beta_1\gamma_0 \\ \alpha_0\beta_1\gamma_1 \\ \alpha_0\beta_0\gamma_0 \\ \alpha_0\beta_0\gamma_1 \\ \alpha_1\beta_1\gamma_0 \\ \alpha_1\beta_1\gamma_1 \\ \alpha_1\beta_0\gamma_0 \\ \alpha_1\beta_0\gamma_1 \end{pmatrix} \quad (7.11)$$

Note that this operation has left qubits *A* and *C* alone, while performing the FLIP operation defined by Eq. 4.1 on qubit *B*.

7.4 The Program

I designed a program, called *QCompute*, that can sequentially perform quantum gate operations on an arbitrary number of qubits. When the program is run, it immediately requests information about each qubit. This information is entered by giving the complex coefficients of each qubit. The program then entangles the *N* qubits into a column vector with 2^N entries. (Note that this allows the user to enter only unentangled states. Entangled states can be obtained by performing appropriate operations.) All non-zero entries of this column vector are now displayed. The user is presented with a series of options as shown in Figure 7.1. When one of the options is selected, *QCompute* will ask the user for more information if necessary and will perform the gate operation. I started with several simple gate operations such as the FLIP gate, the CN gate, and the ability to have a user-defined gate. From here, additions to the program became much easier. I constructed the Toffoli gate from one- and two-qubit

gates using the identity shown in Figure 7.2. The (R)ound off option checks to see if the real or imaginary components of any of the states are less than 10^{-5} , and, if so, sets their values to be zero. This has the effect of countering any round-off errors the classical computer might have. The (E)xperimental gate construction is used for program testing. It is currently used to perform the Controlled-NOT gate for a specified number of iterations in order to test the speed of the program. The Pascal source code for this program is given in Appendix C.

Networks for performing specific tasks can be saved into a text file and then executed for different input. I did this for several networks, including a 1-bit adder (shown in Appendix C.3) and a 2-bit adder (shown in Appendix C.4). A sample of an interactive session with the program is reproduced in Appendix D.

7.5 Results

7.5.1 Two-Bit Adder

The largest test I gave my program was a 2-bit adder, a representation of which is given in Figure 4.4. The file to do this, given in Appendix C.4, was quickly verified to work correctly for several math problems. Then, in order to verify that the quantum

```

(N)ot gate (1-bit gate)           (1)-bit arbitrary gate
(C)ontrolled-Not gate           (2)-bit arbitrary gate
(T)offoli Gate                  (D)eutsch gate
(E)xperimental gate construction (R)ound off and (V)iew
(I)nitialize new quantum state  (Q)uit
Enter an option:

```

Figure 7.1: Quantum computer simulator options

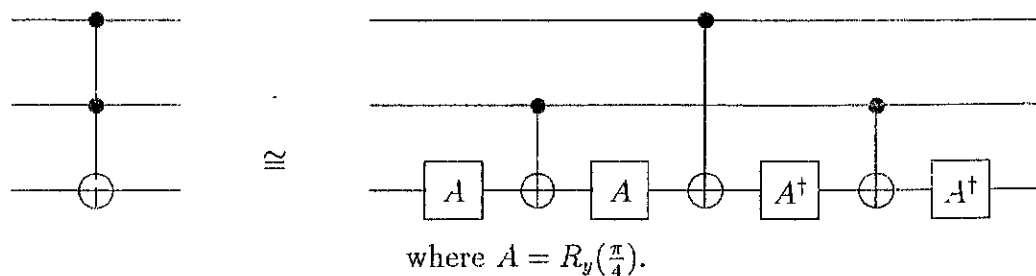


Figure 7.2: Construction of the Toffoli gate

nature of the simulation was working correctly, I initialized the first register to the state $|0\rangle + |1\rangle + |2\rangle + |3\rangle$ and the second register to the same state. The third register, which holds the result, was initialized to be in the $|0\rangle$ state. The result was the equally-weighted superposition

$$\begin{aligned}
& |0\rangle|0\rangle|0\rangle + |0\rangle|1\rangle|1\rangle + |0\rangle|2\rangle|2\rangle + |0\rangle|3\rangle|3\rangle + \\
& |1\rangle|0\rangle|1\rangle + |1\rangle|1\rangle|2\rangle + |1\rangle|2\rangle|3\rangle + |1\rangle|3\rangle|4\rangle + \\
& |2\rangle|0\rangle|2\rangle + |2\rangle|1\rangle|3\rangle + |2\rangle|2\rangle|4\rangle + |2\rangle|3\rangle|5\rangle + \\
& |3\rangle|0\rangle|3\rangle + |3\rangle|1\rangle|4\rangle + |3\rangle|2\rangle|5\rangle + |3\rangle|3\rangle|6\rangle,
\end{aligned} \tag{7.12}$$

as shown in Appendix D. This is precisely the state showing all sixteen addition operations. Incorrect addition problems, such as $|1\rangle|2\rangle|5\rangle$ (signifying $1 + 2 = 5$) have zero probability, while all correct addition problems have a finite probability of occurring. The impressive aspect of this demonstration is that it is no harder for the computer to calculate the answer to all 16 problems than it is for it to calculate the answer to any one of them. Thus an exponential speedup is achieved.

7.5.2 Performance

I had initially expected the program to slow down dramatically for large systems ($N > 3$, where N is the number of qubits in the system) because both the memory required to store a gate operation and the number of operations required to perform the gate operation scale as 2^{2N} . It was not necessary, however, to store the entire gate matrix in the computer at one time. Because the computer constructs the gate matrix, it can do so as it is performing the multiplication, and thus save valuable memory. The program only has to store the current state of the system — a vector which scales as 2^N . Also, because entries in the matrices consist primarily of zeros for operations which modify only one qubit at a time (a universal set, and therefore the only type the program currently allows), most steps in the multiplication can be omitted. By avoiding several multiplications by zero and the corresponding additions of zero, the time requirements can be reduced to scale as 2^N as well. Although still exponential in N , both the space and the time requirements scale according to the square root of my initial predictions, and therefore allow the program to perform well up to $N \approx 10$. Execution times for my program (QCompute) and an independently-developed program (Q) [Joh97] for various numbers of qubits are shown in Table 7.1. The table shows that my program is faster for small systems ($N \leq 5$) while the other program is faster for larger systems. Data analysis shows that QCompute has a runtime which increases according to approximately $2^{1.037N}$ while Q has a runtime

which increases according to approximately $2^{0.997N}$. Both programs are therefore operating close to the predicted runtime of $O(2^N)$. The exact reason why the programs differ slightly from the prediction is undetermined, but is most likely due to the specific compilers used. Compilers often optimize code, and the quality of the executable program they develop may vary between compilers.

| Qubits | CN Gate Time (ms) | |
|--------|-------------------|----------|
| | Q | QCompute |
| 2 | 0.0438 | 0.0394 |
| 3 | 0.0902 | 0.0792 |
| 4 | 0.1746 | 0.1604 |
| 5 | 0.3362 | 0.3274 |
| 6 | 0.6592 | 0.6690 |
| 7 | 1.3282 | 1.3666 |
| 8 | 2.6276 | 2.7846 |
| 9 | 5.3488 | 5.8780 |
| 10 | 10.7552 | 12.2454 |
| 11 | 21.4766 | 25.0450 |
| 12 | 43.0802 | 51.1924 |
| 13 | 86.2758 | — |

Table 7.1: Gate time of two quantum computer simulators for the CN gate acting on various numbers of qubits. QCompute ran out of memory when attempting to operate on a system with 13 qubits. (Timed on an Intel P100 running MS-DOS 6.20 — times given are the average of five trials of 10,000 CN operations, each with accuracy of 0.01s.)

Chapter 8

Conclusions

When I began my study of the quantum computer, I didn't realize how complex the field had become. In a relatively short time, this field has attracted a wide variety of scientists, with specialties ranging from physics to computer science to mathematics. Because of the diversity of the field, I was forced to select only a small segment to study in detail, and could gain only an overview of the other areas. In particular, I chose to examine the underlying theory of the quantum computer and applied this knowledge to develop a quantum computer simulator.

Designing the simulator proved to be a major task. The first problem I encountered was that of entanglement. I discovered that once two qubits are entangled, they may be impossible to disentangle, except through a measurement. Thus my program only deals with the qubits in the entangled states, and never attempts to disentangle them.

When I first conceptualized how the simulator would run, I thought of storing each gate as a $2^N \times 2^N$ matrix. An operation would be performed by multiplying the matrix by the state vector. Some preliminary calculations showed that I would experience some severe limitations due to the time and memory requirements of such a method. Realizing that the matrices consist primarily of zeros and that the computer would have to construct each matrix before using it led me to the idea that I could simulate matrix multiplication by just manipulating the state vector entries in the same way that a matrix would manipulate them. In this way, I avoided constructing the $2^N \times 2^N$ matrix (which would have taken up huge amounts of memory) and reduced the time requirements from being $O(2^{2N})$ to $O(2^N)$. In so doing I made it possible to attack larger problems in shorter times.

One thing that I would change about the simulator if I had to program it all over again is to write it in C++ instead of in Pascal. I originally chose Pascal because

it is easier to understand than C++ for one who is not familiar with programming languages. I wanted to make it easier for a future researcher interested in my simulator to understand and modify the source code. In retrospect, I think that C++ may have been more beneficial, because object oriented programming would have led to a more clear style, which would be easier to understand. Also, a C++ version probably would not have suffered from the same limitations as the Pascal version, allowing it to handle even larger gates and run slightly faster.

The classical computer has served us well and will continue to do so for years to come. It does, however, have its limitations. The quantum computer, if one is ever constructed, will overcome these limitations and lead us into a new era of computing, where exponential speedups will reduce computation times dramatically. It is still uncertain whether a quantum computer will ever be constructed, but I believe there is still much that can be learned, both about theoretical physics and about theoretical computer science, through the study and simulation of such a computer.

Appendix A

Spin

The quantum computer, as it is currently defined, acts on several two-level quantum systems. One of the most common examples of a two-level quantum system is that of electron spin. This appendix will introduce the reader to the concept of quantum spin at a basic level. The interested reader will find more details in a quantum mechanical text [Lib80, Gas96].

A.1 The Basics of Quantum Spin

In quantum theory there are two classes of angular momentum associated with electrons or elementary particles in general, *orbital angular momentum* and *intrinsic*, or *spin angular momentum*. It is the spin of a particle that is addressed here. Spin is independent of the position of a particle, and is therefore an internal property. It is denoted by the symbol $\hat{\mathbf{S}}$. The Cartesian components of $\hat{\mathbf{S}}$ obey the commutation relations

$$[\hat{S}_x, \hat{S}_y] = i\hbar\hat{S}_z, \quad [\hat{S}_y, \hat{S}_z] = i\hbar\hat{S}_x, \quad [\hat{S}_z, \hat{S}_x] = i\hbar\hat{S}_y. \quad (\text{A.1})$$

We may also define the ladder operations for spin as

$$\hat{S}_{\pm} = \hat{S}_x \pm i\hat{S}_y. \quad (\text{A.2})$$

The quantization of spin is given by the conditions on the basis states

$$\hat{S}^2|s, m_s\rangle = \hbar^2 s(s+1)|s, m_s\rangle, \quad (\text{A.3})$$

$$\hat{S}_z|s, m_s\rangle = \hbar m_s|s, m_s\rangle \quad (\text{A.4})$$

where s is a non-negative integer or half-integer. For any given value of s , the quantum number m_s takes values in integral steps from $-s$ to $+s$. For example, for $s = 3/2$, m_s

can take on the values: $-3/2, -1/2, 1/2, 3/2$. The ladder operators have the property that

$$\hat{S}_{\pm}|s, m_s\rangle = \hbar\sqrt{s(s+1) - m_s(m_s \pm 1)}|s, m_s \pm 1\rangle. \quad (\text{A.5})$$

From this information, it is possible to determine the operations $\hat{S}^2, \hat{S}_x, \hat{S}_y, \hat{S}_z$ on any given spin state. An example of how this would be done for a spin $3/2$ particle follows.

For all particles, Eqs. A.3 and A.4 hold. In the case of a spin $3/2$ particle, we may represent the basis states $|s, m_s\rangle$ as:

$$\left|\frac{3}{2}, \frac{3}{2}\right\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad \left|\frac{3}{2}, \frac{1}{2}\right\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \left|\frac{3}{2}, -\frac{1}{2}\right\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad \left|\frac{3}{2}, -\frac{3}{2}\right\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}. \quad (\text{A.6})$$

From this we can determine that for the spin $3/2$ particle

$$\hat{S}^2 = \frac{15\hbar^2}{4} \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix} \quad \text{and} \quad \hat{S}_z = \frac{\hbar}{2} \begin{pmatrix} 3 & & & \\ & 1 & & \\ & & -1 & \\ & & & -3 \end{pmatrix}. \quad (\text{A.7})$$

We now need to determine \hat{S}_x and \hat{S}_y . To do so, we begin by defining

$$\hat{S}_+|s, m_s\rangle = (\hat{S}_x + i\hat{S}_y)|s, m_s\rangle = \hbar\sqrt{s(s+1) - m_s(m_s + 1)}|s, m_s + 1\rangle, \quad (\text{A.8})$$

$$\hat{S}_-|s, m_s\rangle = (\hat{S}_x - i\hat{S}_y)|s, m_s\rangle = \hbar\sqrt{s(s+1) - m_s(m_s - 1)}|s, m_s - 1\rangle. \quad (\text{A.9})$$

Then we can easily find

$$\hat{S}_+ = \hbar \begin{pmatrix} 0 & \sqrt{3} & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & \sqrt{3} \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \text{and} \quad \hat{S}_- = \hbar \begin{pmatrix} 0 & 0 & 0 & 0 \\ \sqrt{3} & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & \sqrt{3} & 0 \end{pmatrix} \quad (\text{A.10})$$

Using the relations $\hat{S}_x = \frac{1}{2}(\hat{S}_+ + \hat{S}_-)$ and $\hat{S}_y = -\frac{1}{2i}(\hat{S}_+ - \hat{S}_-)$ we find

$$\hat{S}_x = \frac{\hbar}{2} \begin{pmatrix} 0 & \sqrt{3} & 0 & 0 \\ \sqrt{3} & 0 & 2 & 0 \\ 0 & 2 & 0 & \sqrt{3} \\ 0 & 0 & \sqrt{3} & 0 \end{pmatrix} \quad \text{and} \quad \hat{S}_y = \frac{i\hbar}{2} \begin{pmatrix} 0 & -\sqrt{3} & 0 & 0 \\ \sqrt{3} & 0 & -2 & 0 \\ 0 & 2 & 0 & -\sqrt{3} \\ 0 & 0 & \sqrt{3} & 0 \end{pmatrix}. \quad (\text{A.11})$$

If we wish to check to make sure our answers are correct, we may do so using the relation $\hat{S}_x^2 + \hat{S}_y^2 + \hat{S}_z^2 = \hat{S}^2$. In this case, we find

$$\hat{S}_x^2 + \hat{S}_y^2 + \hat{S}_z^2 = \frac{\hbar^2}{4} \begin{pmatrix} 3 & 0 & 2\sqrt{3} & 0 \\ 0 & 7 & 0 & 2\sqrt{3} \\ 2\sqrt{3} & 0 & 7 & 0 \\ 0 & 2\sqrt{3} & 0 & 3 \end{pmatrix} \quad (\text{A.12})$$

$$\begin{aligned}
& + \frac{\hbar^2}{4} \begin{pmatrix} 3 & 0 & -2\sqrt{3} & 0 \\ 0 & 7 & 0 & -2\sqrt{3} \\ -2\sqrt{3} & 0 & 7 & 0 \\ 0 & -2\sqrt{3} & 0 & 3 \end{pmatrix} \\
& + \frac{\hbar^2}{4} \begin{pmatrix} 9 & & & \\ & 1 & & \\ & & 1 & \\ & & & 9 \end{pmatrix} \\
& = \frac{15\hbar^2}{4} \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix} \\
& = \hat{S}^2.
\end{aligned}$$

This concludes the consistency check of these expressions. It should be noted that analogous results may easily be derived for arbitrary values of s .

A.2 Mixed Spin States

Up to this point, only idealized conditions have been considered. The particle was isolated from and uncoupled to the external environment. Let us now consider the case in which the system *is* coupled to the external environment. Under these conditions, it may not be possible to determine the wavefunction, and we may say that the system does not have a wavefunction [Lib80]. A system that does not have a wavefunction is said to be in a *mixed state* while a system that does have a wavefunction is said to be in a *pure state*.

Under conditions when we do not know the state of the system, we take the point of view that although a wavefunction exists for the system, it is not completely determined [Lib80]. In place of this wavefunction, we introduce the density operator $\hat{\rho}$. If A is some property of the system, we can find the expectation value of A through the relation

$$\langle A \rangle = \text{Tr } \hat{\rho} A \quad (\text{A.13})$$

where $\text{Tr } \hat{\rho} = 1$ and $\hat{\rho} = \hat{\rho}^\dagger$ where \dagger indicates Hermitian conjugation. The trace operation, denoted Tr , sums over the diagonal elements of a matrix. Therefore,

$$\text{Tr } A = \sum_n A_{nn}. \quad (\text{A.14})$$

A particle has an *isotropic distribution* if

$$\langle \hat{S}_x \rangle = \langle \hat{S}_y \rangle = \langle \hat{S}_z \rangle = 0 \quad (\text{A.15})$$

and

$$\langle \hat{S}_x^2 \rangle = \langle \hat{S}_y^2 \rangle = \langle \hat{S}_z^2 \rangle . \quad (\text{A.16})$$

This means that the expectation values for all three directions of spin are zero. If we wish to find the ρ matrix that corresponds to this situation, we must find the matrix ρ which satisfies the relations

$$\begin{aligned} \langle \hat{S}_x \rangle &= \text{Tr } \hat{\rho} \hat{S}_x = 0 , \\ \langle \hat{S}_y \rangle &= \text{Tr } \hat{\rho} \hat{S}_y = 0 , \\ \langle \hat{S}_z \rangle &= \text{Tr } \hat{\rho} \hat{S}_z = 0 , \end{aligned} \quad (\text{A.17})$$

and

$$\begin{aligned} \langle \hat{S}_x^2 \rangle &= \text{Tr } \hat{\rho} \hat{S}_x^2 = \frac{1}{3} \hat{S}^2 , \\ \langle \hat{S}_y^2 \rangle &= \text{Tr } \hat{\rho} \hat{S}_y^2 = \frac{1}{3} \hat{S}^2 , \\ \langle \hat{S}_z^2 \rangle &= \text{Tr } \hat{\rho} \hat{S}_z^2 = \frac{1}{3} \hat{S}^2 . \end{aligned} \quad (\text{A.18})$$

For the case of the spin 3/2 particle we might guess, based on symmetry, that

$$\rho = \frac{1}{4} \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{pmatrix} . \quad (\text{A.19})$$

This matrix satisfies the conditions $\text{Tr } \hat{\rho} = 1$ and $\hat{\rho} = \hat{\rho}^\dagger$. Also, using the matrices found earlier for $\hat{S}_x, \hat{S}_y, \hat{S}_z, \hat{S}_x^2, \hat{S}_y^2, \hat{S}_z^2, \hat{S}^2$ we may quickly verify that this choice of $\hat{\rho}$ satisfies the other six requirements as well. In all generality, we could start with a matrix which only satisfied the last two conditions, and then attempt to find conditions which would help us to determine the correct matrix. In this example, the starting matrix would be

$$\rho = \begin{pmatrix} X & a + i\alpha & b + i\beta & c + i\gamma \\ a - i\alpha & Y & d + i\delta & e + i\epsilon \\ b - i\beta & d - i\delta & Z & f + i\zeta \\ c - i\gamma & e - i\epsilon & f - i\zeta & 1 - (X + Y + Z) \end{pmatrix} . \quad (\text{A.20})$$

From this starting point, we could apply all of the above conditions until we determined the correct matrix for $\hat{\rho}$.

Appendix B

The RSA Cryptosystem

One of the most promising applications of the quantum computer is that of factoring large numbers using Shor's algorithm (described in Section 5.1). Factoring is important, not just to the study of mathematics, but also to the science of cryptography. The most popular method of public-key encryption in use today relies on the difficulty of factoring numbers for its security. This method, named RSA after its inventors Rivest, Shamir, and Adleman, is currently being used by individuals, banks, and governments for secure communication. If a quantum computer is constructed, it would be able to decipher any of these communications in seconds. A general reference on RSA can be consulted for further details [Fah93].

B.1 Theory

Each user must develop a pair of large (100 digits or more) prime numbers, denoted p and q . These numbers are multiplied together to get a number m , the modulus. Also, the user comes up with integers d and e such that de is relatively prime to $n \equiv (p-1)(q-1)$. The numbers m and e (the encryptor) are made public, and d (the decryptor) is kept secret. The primes p and q are no longer necessary, and should be destroyed.

Now, when Alice wishes to send an encrypted message to Bob, Alice must use Bob's m and e in the function

$$C = P^e \pmod{m}, \quad (\text{B.1})$$

where P is the plaintext and C is the ciphertext. Once Bob receives the message, he decrypts it according to the function

$$P = C^d \pmod{m}. \quad (\text{B.2})$$

Bob will get the original plaintext message back because he is actually performing the following steps:

1. First, by the Chinese remainder theorem [And94],

$$[P^e \pmod{m}]^d \pmod{m} = P^{de} \pmod{m}. \quad (\text{B.3})$$

2. Now, we know from Euler that if $\gcd(P, m) = 1$ then $P^{\phi(m)} = 1 \pmod{m}$ for $\phi(m) = \phi(pq) \equiv (p-1)(q-1) = n$ (the number of positive integers less than m and relatively prime to m) [Gar95]. Hence as long as P is not a multiple of p or of q , we may write

$$P^{de} \pmod{m} = P^{kn+1} \pmod{m} = P \cdot (1)^k \pmod{m} = P \quad (\text{B.4})$$

where k is an integer.

B.2 Signaturization

RSA also allows Alice to sign a message to authenticate that it is real. She does this by encrypting it using her decryptor, denoted d_A , before encrypting it using Bob's encryptor, denoted e_B . Thus, the resulting operation would be

$$C = [P^{d_A} \pmod{m_A}]^{e_B} \pmod{m_B}. \quad (\text{B.5})$$

When receiving the message, Bob would decrypt the message using d_B and m_B , and then encrypt it using e_A and m_A , according to the function

$$P = [C^{d_B} \pmod{m_B}]^{e_A} \pmod{m_A}. \quad (\text{B.6})$$

But this is the same as performing the operation

$$\begin{aligned} P &= [[P^{d_A} \pmod{m_A}]^{e_B} \pmod{m_B}]^{d_B} \pmod{m_B}]^{e_A} \pmod{m_A} \quad (\text{B.7}) \\ &= P^{d_A e_A d_B e_B} \pmod{m_A} \pmod{m_B} = P, \end{aligned}$$

where the Chinese remainder theorem and Euler's theorem have been used as in Eqs. B.3 and B.4 above. Therefore Bob will only be able to read the message if it was indeed encrypted using Alice's decryptor d_A , which only Alice knows.

B.3 Security

RSA derives its security from the difficulty of factoring large numbers. Computers can easily multiply numbers together, but there is no known method for easily factoring large numbers. So far, the most commonly used factoring methods are Pollard's $p-1$ method, the elliptic curve method, the number field sieve, and the multiple polynomial quadratic sieve (MPQS). These have running times, in big-O notation, as shown in Table B.1. With two 100-digit primes, m would be about 200 digits long, so the best

| | |
|--------------------|---|
| Pollard Rho | $O(\sqrt{p})$ |
| Pollard's $p-1$ | $O(p')$ where p' is the largest prime factor of $p-1$ |
| Elliptic Curve | $O(e^{\sqrt{2 \ln p \ln \ln p}})$ |
| Number Field Sieve | $O(e^{1.9(\ln n)^{1/3}(\ln \ln n)^{2/3}})$ |
| MPQS | $O(e^{\sqrt{\ln n \ln \ln n}})$ |

Table B.1: Running times of various factoring methods, where n is the number to be factored and p is a prime factor of n [Fah93].

known factoring method, the MPQS, would take on the order of

$$e^{\sqrt{\ln(10^{200}) \ln(\ln(10^{200}))}} \simeq e^{53.14} \simeq 10^{23} \quad (\text{B.8})$$

operations. The world's fastest computer runs at 1.08 Tflops (short for trillion floating point operations per second) [SS96b]. Therefore it would take this computer over 3000 years to factor the number.

Factoring m would allow the intruder to calculate the person's decryptor, d . It is not, however, necessary to factor m to do this. The only other known way to crack the cipher would be to discover the recipient's decryptor, d , directly. In general, d will also be a large number, and, because the encrypting and decrypting processes are time-consuming, it would take similarly long amounts of time to try every possible value for d . Barring a significant factoring breakthrough, therefore, the RSA cryptosystem will always be secure. As computers improve all that is necessary to maintain the integrity of a message is to increase the lengths of p and q , and therefore m .

Appendix C

The Program

The following is the Pascal source code of the quantum computer simulator I wrote. The program consists of two program files and two optional network files. The first program file, `Complex.pas`, handles all manipulation of complex numbers for the program. The second program file, `QCompute.pas` is the heart of the program. This file takes care of performing all gate operations and provides the user interface. The two network files, `Adder.q` and `Adder2.q`, are script files which perform one- and two-bit addition networks, respectively. The programs shown below are included on the enclosed disk. In addition to the source code, the disk also contains the DOS executable. For more information, view the `readme.txt` file on the disk.

C.1 `Complex.pas`

```
{$IFDEF COMPLEX}
{$DEFINE COMPLEX}

type
  complex = record
    Re: real;
    Im: real;
  end; {record}

procedure AddComplex(a, b: complex; var c: complex);
{Desc: Adds two complex numbers and returns the result.
 Pre: a and b are complex numbers.
 Post: The sum a+b has been returned as c.
 Uses: None.}
begin {procedure AddComplex}
  c.Re := a.Re + b.Re;
  c.Im := a.Im + b.Im;
end; {procedure AddComplex}
```

```

procedure SubComplex(a, b: complex; var c: complex);
{Desc: Subtracts two complex numbers and returns the result.
 Pre: a and b are complex numbers.
 Post: The difference a-b has been returned as c.
 Uses: None.}
begin {procedure SubComplex}
  c.Re := a.Re - b.Re;
  c.Im := a.Im - b.Im;
end; {procedure SubComplex}

procedure MulComplex(a, b: complex; var c: complex);
{Desc: Multiplies two complex numbers and returns the result.
 Pre: a and b are complex numbers.
 Post: The result of a*b has been returned as c.
 Uses: None.}
begin {procedure MulComplex}
  c.Re := a.Re*b.Re - a.Im*b.Im;
  c.Im := a.Re*b.Im + a.Im*b.Re;
end; {procedure MulComplex}

{$ENDIF}

```

C.2 QCompute.pas

```

{
Program Name: Quantum Computer Simulator
Version: 1.0
Programmer: Damian Menscher
Description: Simulates the actions of a quantum computer.
Date: July 9, 1997
}

program QCompute;

uses Dos;

{$F+} {enable procedural types}

{$I GenUtils.pas} {general utilities package}

const
  MAXQBITS=7; {Arbitrary}
  VECTORSIZE=128; {=2^MAXQBITS;}
  UMATRIXSIZE=4; {=2*2;}
  PI=3.14159265358979; {for rotation matrices}

{$I Complex.pas} {package to handle complex numbers}

type
  StateVector=array[0..VECTORSIZE-1] of complex;
  UArrayType=array[1..UMATRIXSIZE] of complex;

```

```

var
  CurState:      StateVector;
  infile:        text;
  ch:            string;
  counter, iter: integer;
  UArray,
  ToffoliArray: UArrayType;
  qubit,
  Control,
  Controlled:    integer;
  zero, one:     complex;
  theta:         real;
  h0, m0, s0,
  hund0, hf, mf,
  sf, hundf:     Word;

```

```

procedure RoundOff(var curstate: StateVector);
{Desc: Rounds off values close to zero to be exactly zero.
 Pre:  Receives the curstate StateVector
 Post: Returns a rounded StateVector
 Uses: None}

```

```

var
  counter: integer;
begin {procedure RoundOff}
  for counter:=0 to VECTORSIZE-1 do
    begin {for}
      if abs(curstate[counter].Re) < 1E-5 then
        curstate[counter].Re := 0;
      if abs(curstate[counter].Im) < 1E-5 then
        curstate[counter].Im := 0;
    end; {for}
end; {procedure RoundOff}

```

```

function PowOf2(exponent: integer): integer;
{Desc: Finds a power of two.
 Pre:  Receives an exponent that is an integer.
 Post: Returns the power of two that was requested.
 Uses: Error}

```

```

var
  counter,
  temp:    integer;
begin {function PowOf2}
  if exponent < 0 then
    Error('PowerOfTwo must receive a non-negative integer')
  else
    begin {else}
      temp := 1;
      while exponent > 0 do
        begin {while}
          temp := temp * 2;
          exponent := exponent - 1;
        end;
    end;
end;

```

```

        end; {while}
    end; {else}
    PowOf2 := temp;
end; {function PowOf2}

```

```

procedure SwapComplex(var a, b: complex);
{Desc: Swaps contents of complex variables a and b.
Pre: Variables a and b exist.
Post: The contents of a and b have been exchanged.
Uses: None}
var
    temp: complex;

begin {procedure SwapComplex}
    temp := b;
    b := a;
    a := temp;
end; {procedure SwapComplex}

```

```

procedure ViewState(CurrentState: StateVector);
{Desc: Displays the current statevector.
Pre: CurrentState exists.
Post: The current statevector has been displayed for the user.
Uses: None}
var
    qubit,
    counter : integer;
    prob:    real;
begin {procedure ViewState}
    writeln;
    for qubit := 0 to VECTORSIZE-1 do
        if (CurrentState[qubit].Re<>0) or (CurrentState[qubit].Im<>0) then
            begin {if}
                write('    State ', qubit:3, '=');
                for counter := MAXQBITS-1 downto 0 do
                    if (qubit and PowOf2(counter)) > 0 then
                        write('1')
                    else
                        write('0');
                write(': ');
                write(CurrentState[qubit].Re:6:3, ' + i', CurrentState[qubit].Im:6:3);
                prob := Sqr(CurrentState[qubit].Re) + Sqr(CurrentState[qubit].Im);
                writeln(' Probability: ', prob*100:7:3, '%');
            end; {if}
        end;
    end; {procedure ViewState}

```

```

function PromptQubit(s: string): integer;
{Desc: Displays string s to ask for a qubit letter. If the user gives invalid
input, an error message is displayed and the user is asked again.
Pre: None.
Post: A valid qubit number has been returned.

```

```

Uses: None.}
var
  ch:   string;
  input,
  qubit: integer;

begin {function PromptQubit}
  write(s);
  readln(infile, ch);
  input := -1;
  repeat
    qubit := ord(ch[1])-ord('A');
    if ((qubit >= 0) and (qubit < MAXQBITS)) then
      input := qubit
    else
      begin {else}
        qubit := ord(ch[1])-ord('a');
        if ((qubit >= 0) and (qubit < MAXQBITS)) then
          input := qubit
        else
          begin {else}
            writeln('Qubit must between A and ', chr(MAXQBITS-1+ord('A')), '.');
            readln(infile, ch);
          end; {else}
        end; {else}
      until (input >= 0);
      PromptQubit := input;
  end; {function PromptQubit}

```

```

procedure Initialize(var CurrentState: StateVector);
{Desc: Initializes the StateVector CurrentState to what the user specifies.
Pre: The StateVector CurrentState exists.
Post: CurrentState holds the values corresponding to the user input.
Uses: MultiplyComplex}

```

```

var
  qubit,
  coeff: integer;
  coeffs: array[1..MAXQBITS, 0..1] of complex;
  one: complex;
  norm: real;

```

```

begin {procedure Initialize}
  one.Re := 1;
  one.Im := 0;
  for coeff := 1 to MAXQBITS do
    begin {for}
      write('Enter complex value for qubit ', chr(coeff-1+ord('A')), ' in |0> ');
      readln(coeffs[coeff, 0].Re, coeffs[coeff, 0].Im);
      write('Enter complex value for qubit ', chr(coeff-1+ord('A')), ' in |1> ');
      readln(coeffs[coeff, 1].Re, coeffs[coeff, 1].Im);
      norm := Sqr(coeffs[coeff, 0].Re)+Sqr(coeffs[coeff, 0].Im);
      norm := Sqr(coeffs[coeff, 1].Re)+Sqr(coeffs[coeff, 1].Im)+norm;
    end;
  end;

```

```

    if (norm>0) then
      begin
        coeffs[coeff,0].Re := coeffs[coeff,0].Re/Sqrt(norm);
        coeffs[coeff,0].Im := coeffs[coeff,0].Im/Sqrt(norm);
        coeffs[coeff,1].Re := coeffs[coeff,1].Re/Sqrt(norm);
        coeffs[coeff,1].Im := coeffs[coeff,1].Im/Sqrt(norm);
      end
    else
      Error('Can't have zero probability for a qubit');
    end; {for}
  for qubit := 0 to VECTORSIZE-1 do
    begin {for qubit}
      CurrentState[qubit] := one;
      for coeff := 1 to MAXQBITS do
        if (qubit mod PowOf2(MAXQBITS-coeff+1)) < PowOf2(MAXQBITS-coeff) then
          MulComplex(CurrentState[qubit], coeffs[coeff, 0], CurrentState[qubit])
        else
          MulComplex(CurrentState[qubit], coeffs[coeff, 1], CurrentState[qubit]);
        end; {for coeff}
      end; {for qubit}
    end; {procedure Initialize}
  end;

```

```

procedure GetUArray(var UArray: UArrayType);
{Desc: Gets the U-Matrix from the user.
Pre: UArray exists.
Post: UArray has values such as the user specifies.
Uses: None.}

```

```

var
  counter: integer;

begin {procedure GetUArray}
  writeln('Input the form of the U-matrix you want to multiply by');
  for counter := 1 to UMATRIXSIZE do
    begin {for}
      write('Real and imaginary componets for position ', counter, ': ');
      readln(infile, UArray[counter].Re, UArray[counter].Im);
    end; {for}
  end; {procedure GetUArray}

```

```

procedure RotateY(theta: real; var UArray: UArrayType);
{Desc: Creates a UMatrix corresponding to a rotateY of theta.
Pre: UArray exists.
Post: UArray has been set to its proper value.
Uses: None}

```

```

begin {procedure RotateY}
  UArray[1].Re := cos(theta/2);
  UArray[1].Im := 0;
  UArray[2].Re := sin(theta/2);
  UArray[2].Im := 0;
  UArray[3].Re := -sin(theta/2);
  UArray[3].Im := 0;
  UArray[4].Re := cos(theta/2);
  UArray[4].Im := 0;

```



```
end; {procedure RotateY}
```

```
procedure RotateZ(theta: real; var UArray: UArrayType);  
{Desc: Creates a UMatrix corresponding to a rotateZ of theta.  
Pre: UArray exists.  
Post: UArray has been set to its proper value.  
Uses: None}  
begin {procedure RotateZ}  
  UArray[1].Re := cos(theta/2);    {exp(i theta/2)}  
  UArray[1].Im := sin(theta/2);  
  UArray[2].Re := 0;  
  UArray[2].Im := 0;  
  UArray[3].Re := 0;  
  UArray[3].Im := 0;  
  UArray[4].Re := cos(theta/2);    {exp(-i theta/2)}  
  UArray[4].Im := -sin(theta/2);  
end; {procedure RotateZ}
```

```
procedure NotGate(qubit: integer; var CurState: StateVector);  
{Desc: NOTs one of the quantum bits in CurState.  
Pre: CurState exists.  
Post: One of the bits in CurState has been NOTed.  
Uses: SwapComplex.}  
var  
  pos: integer;
```

```
begin {procedure NotGate}  
  for pos := 0 to VECTORSIZE-1 do  
    if (pos mod PowOf2(MAXQBITS-qubit)) < PowOf2(MAXQBITS-(qubit+1)) then  
      SwapComplex(CurState[pos], CurState[pos+PowOf2(MAXQBITS-(qubit+1))]);  
end; {procedure NotGate}
```

```
procedure OneBitGate(UArray: UArrayType; bit: integer; var CurState: StateVector);  
{Desc: Performs a one-bit gate operation.  
Pre: CurState exists.  
Post: The one-bit operation specified by UArray has been performed on the  
      bit specified by bit.  
Uses: PowOf2}
```

```
var  
  pos,  
  Uoffset: integer;  
  temp: complex;  
  NewState: StateVector;  
  
begin {procedure OneBitGate}  
  Uoffset := PowOf2(MAXQBITS-(bit+1));  
  for pos := 0 to VECTORSIZE-1 do  
    begin {for pos}  
      NewState[pos].Re := 0;  
      NewState[pos].Im := 0;  
    end; {for pos}
```

```

for pos := 0 to VECTORSIZE-1 do
  begin {for pos}
    if (pos mod PowOf2(MAXQBITS-bit)) < PowOf2(MAXQBITS-(bit+1)) then
      begin {if}
        MulComplex(UArray[1], CurState[pos], temp);
        AddComplex(temp, NewState[pos], NewState[pos]);
        MulComplex(UArray[2], CurState[pos+Uoffset], temp);
        AddComplex(temp, NewState[pos], NewState[pos]);
        MulComplex(UArray[3], CurState[pos], temp);
        AddComplex(temp, NewState[pos+Uoffset], NewState[pos+Uoffset]);
        MulComplex(UArray[4], CurState[pos+Uoffset], temp);
        AddComplex(temp, NewState[pos+Uoffset], NewState[pos+Uoffset]);
      end; {if}
    end; {for pos}
  CurState := NewState;
end; {procedure OneBitGate}

procedure ControlledMult(UArray: UArrayType;
  Ctrl, Ctrlled: integer;
  var CurState: StateVector);
{Desc: Multiplies the CurState by a new operator matrix to come up with
  the new CurState.
Pre: CurState exists.
Post: CurState has been multiplied by the operator matrix corresponding
  to the u-matrix, control bit, and controlled bit specified.
Uses: PowOf2, AddComplex, MulComplex}
var
  NewState: StateVector;
  pos: integer;
  temp: complex;
  Uoffset: integer;

begin {procedure ControlledMult}
  Uoffset := PowOf2(MAXQBITS-(Ctrlled+1));
  for pos := 0 to VECTORSIZE-1 do
    begin {for pos}
      NewState[pos].Re := 0;
      NewState[pos].Im := 0;
    end; {for pos}
  for pos := 0 to VECTORSIZE-1 do
    begin {for pos}
      if (pos mod PowOf2(MAXQBITS-Ctrl)) < PowOf2(MAXQBITS-(Ctrl+1)) then
        AddComplex(NewState[pos], CurState[pos], NewState[pos]) {a 1 in matrix}
      else
        if pos mod PowOf2(MAXQBITS-Ctrlled) < PowOf2(MAXQBITS-(Ctrlled+1)) then
          begin {if}
            {found a U00 entry}
            MulComplex(UArray[1], CurState[pos], temp);
            AddComplex(temp, NewState[pos], NewState[pos]);
            MulComplex(UArray[2], CurState[pos+Uoffset], temp);
            AddComplex(temp, NewState[pos], NewState[pos]);
            MulComplex(UArray[3], CurState[pos], temp);
            AddComplex(temp, NewState[pos+Uoffset], NewState[pos+Uoffset]);
            MulComplex(UArray[4], CurState[pos+Uoffset], temp);
            AddComplex(temp, NewState[pos+Uoffset], NewState[pos+Uoffset]);
          end;
        end;
    end;
  end;
end;

```

```

        end; {if}
    end; {for pos}
    CurState := NewState;
end; {procedure ControlledMult}

```

```

{
procedure ControlledNot(Control, Controlled: qubit; var CurState: StateVector);
{Desc: Performs the Controlled-Not operation.
Pre: CurState exists.
Post: The CN operation has been performed on Curstate.
Uses: ControlledMult}

```

```

procedure ThreeGate(UArray: UArrayType;
                   Ctrl1, Ctrl2, Ctrlled: integer;
                   var CurState: StateVector);
{Desc: Multiplies the CurState by a new operator matrix to come up with
the new CurState.
Pre: CurState exists.
Post: CurState has been multiplied by the operator matrix corresponding
to the u-matrix, control bits, and controlled bit specified.
Uses: PowOf2, AddComplex, MulComplex}

```

```
var
```

```

    NewState:    StateVector;
    pos:         integer;
    temp:        complex;
    Uoffset:     integer;

```

```

begin {procedure ThreeGate}
    Uoffset := PowOf2(MAXQBITS-(Ctrlled+1));
    for pos := 0 to VECTORSIZE-1 do
        begin {for pos}
            NewState[pos].Re := 0;
            NewState[pos].Im := 0;
        end; {for pos}
    for pos := 0 to VECTORSIZE-1 do
        begin {for pos}
            if (pos mod PowOf2(MAXQBITS-Ctrl1)) < PowOf2(MAXQBITS-(Ctrl1+1)) then
                AddComplex(NewState[pos], CurState[pos], NewState[pos]) {a 1 in matrix}
            else if (pos mod PowOf2(MAXQBITS-Ctrl2)) < PowOf2(MAXQBITS-(Ctrl2+1)) then
                AddComplex(NewState[pos], CurState[pos], NewState[pos]) {a 1 in matrix}
            else
                if pos mod PowOf2(MAXQBITS-Ctrlled) < PowOf2(MAXQBITS-(Ctrlled+1)) then
                    begin {if}
                        {found a U00 entry}
                        MulComplex(UArray[1], CurState[pos], temp);
                        AddComplex(temp, NewState[pos], NewState[pos]);
                        MulComplex(UArray[2], CurState[pos+Uoffset], temp);
                        AddComplex(temp, NewState[pos], NewState[pos]);
                        MulComplex(UArray[3], CurState[pos], temp);
                        AddComplex(temp, NewState[pos+Uoffset], NewState[pos+Uoffset]);
                        MulComplex(UArray[4], CurState[pos+Uoffset], temp);
                        AddComplex(temp, NewState[pos+Uoffset], NewState[pos+Uoffset]);
                    end; {if}
                end; {for pos}
    end; {for pos}

```

```

CurState := NewState;
end; {procedure ThreeGate}

procedure DeutschUArray(var UArray: UArrayType; theta: real);
{Desc: Returns the correct UArray for the Deutsch gate.}
begin {procedure DeutschUArray}
  UArray[1].Re:=0;
  UArray[1].Im:=cos(theta);
  UArray[2].Re:=sin(theta);
  UArray[2].Im:=0;
  UArray[3].Re:=sin(theta);
  UArray[3].Im:=0;
  UArray[4].Re:=0;
  UArray[4].Im:=cos(theta);
end; {procedure DeutschUArray}

procedure ShowMenu;
{Desc: Displays the menu options.}
begin {procedure ShowMenu}
  writeln('(N)ot gate (1-bit gate)           (1)-bit arbitrary gate');
  writeln('(C)ontrolled-Not gate           (2)-bit arbitrary gate');
  writeln('(T)offoli Gate           (D)eutsch gate');
  writeln('(E)xperimental gate construction (R)ound off and (V)iew');
  writeln('(I)nititalize new quantum state (Q)uit');
end; {procedure ShowMenu}

begin {Main}
  assign(infile, paramstr(1));
  reset(infile);
  zero.Re := 0;
  zero.Im := 0;
  one.Re := 1;
  one.Im := 0;
  ToffoliArray[1] := zero;
  ToffoliArray[2] := one;
  ToffoliArray[3] := one;
  ToffoliArray[4] := zero;

  Initialize(CurState);
  ViewState(CurState);
  GetTime(h0, m0, s0, hund0);
  ShowMenu;
  repeat
    write('Enter an option: ');
    ch[1] := ' '; {so if there is no user input nothing will happen}
    readln(infile, ch);
    case ch[1] of
      'N', 'n': begin {case Not Gate}
        qubit := PromptQubit(' Which line should be NOTted? ');
        NotGate(qubit, CurState);
      end; {case Not Gate}
      '1': begin {case 1-bit gate}
        GetUArray(UArray);
        qubit := PromptQubit(' Perform the gate on which line? ');

```

```

    OneBitGate(UArray, qubit, CurState);
end; {case 1-bit gate}
'C', 'c': begin {case Controlled-Not gate}
    Control := PromptQubit(' Which bit is the control bit? ');
    Controlled := PromptQubit(' The controlled bit? ');
    ControlledMult(ToffoliArray, Control, Controlled, CurState);
end; {case Controlled-Not gate}
'2': begin {case 2-bit gate}
    GetUArray(UArray);
    Control := PromptQubit(' Which bit is the control bit? ');
    Controlled := PromptQubit(' The controlled bit? ');
    ControlledMult(UArray, Control, Controlled, CurState);
end; {case 2-bit gate}
'T', 't': begin {case Toffoli}
    Control := PromptQubit(' Which is the first control bit? ');
    qubit := PromptQubit(' Which is the second control bit? ');
    Controlled := PromptQubit(' Which bit is being controlled? ');
    RotateY(Pi/4, UArray);    {UArray becomes a Pi/4 Y rotation}
    OneBitGate(UArray, Controlled, CurState);
    ControlledMult(ToffoliArray, Control, Controlled, CurState);
    OneBitGate(UArray, Controlled, CurState);
    ControlledMult(ToffoliArray, qubit, Controlled, CurState);
    RotateY(-Pi/4, UArray);    {UArray becomes a -Pi/4 Y rotation}
    OneBitGate(UArray, Controlled, CurState);
    ControlledMult(ToffoliArray, Control, Controlled, CurState);
    OneBitGate(UArray, Controlled, CurState);
end; {case Toffoli}
'3': begin {case 3-bit gate}
    writeln(' This procedure uses the square root of the UArray. ');
    GetUArray(UArray);
    Control := PromptQubit(' Which is the first control bit? ');
    qubit := PromptQubit(' Which is the second control bit? ');
    Controlled := PromptQubit(' Which bit is being controlled? ');
    ThreeGate(UArray, Control, qubit, Controlled, CurState);
    RotateY(Pi/4, UArray);    {UArray becomes a Pi/4 Y rotation}
    OneBitGate(UArray, Controlled, CurState);
    ControlledMult(ToffoliArray, Control, Controlled, CurState);
    OneBitGate(UArray, Controlled, CurState);
    ControlledMult(ToffoliArray, qubit, Controlled, CurState);
    RotateY(-Pi/4, UArray);    {UArray becomes a -Pi/4 Y rotation}
    OneBitGate(UArray, Controlled, CurState);
    ControlledMult(ToffoliArray, Control, Controlled, CurState);
    OneBitGate(UArray, Controlled, CurState);
end; {case Toffoli}

'D', 'd': begin {case Deutsch}
    Control := PromptQubit(' Which is the first control bit? ');
    qubit := PromptQubit(' Which is the second control bit? ');
    Controlled := PromptQubit(' Which bit is being controlled? ');
    write(' What is the value of theta (Pi/2=1.570796327)? ');
    readln(infile, theta);
    DeutschUArray(UArray, theta);
    ThreeGate(UArray, Control, qubit, Controlled, CurState);
(* An alternate method for the Deutsch gate building up from smaller gates

```

```

{First we do a rotateZ}
RotateZ(PI/4, UArray);    {UArray becomes a Pi/4 Z rotation}
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, Control, Controlled, CurState);
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, qubit, Controlled, CurState);
RotateZ(-PI/4, UArray);  {UArray becomes a -Pi/4 Z rotation}
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, Control, Controlled, CurState);
OneBitGate(UArray, Controlled, CurState);
{Then we multiply by the V-matrix}
VMatrix(0, PI/2, theta, UArray);
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, Control, Controlled, CurState);
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, qubit, Controlled, CurState);
VMatrix(0, PI/2, -theta, UArray);{UArray becomes a -Pi/4 Z rot}
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, Control, Controlled, CurState);
OneBitGate(UArray, Controlled, CurState);
{Finally we do a -rotateZ}
RotateZ(-PI/4, UArray);  {UArray becomes a -Pi/4 Z rotation}
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, Control, Controlled, CurState);
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, qubit, Controlled, CurState);
RotateZ(PI/4, UArray);    {UArray becomes a Pi/4 Z rotation}
OneBitGate(UArray, Controlled, CurState);
ControlledMult(ToffoliArray, Control, Controlled, CurState);
OneBitGate(UArray, Controlled, CurState);

```

*)

```

end; {case Deutsch}
'R', 'r': begin {case RoundOff}
  RoundOff(CurState);
  ViewState(CurState);
end; {case RoundOff}
'V', 'v': ViewState(CurState);
'E', 'e': begin {case Experimental}
  write(' How many iterations of the CN gate? ');
  readln(iter);
  GetTime(h0, m0, s0, hund0);
  for counter := 1 to iter do
    ControlledMult(ToffoliArray, 0, 1, CurState);
    GetTime(hf, mf, sf, hundf);
    writeln(' Start Time: ', h0, ':', m0, ':', s0, '.', hund0);
    writeln(' End Time: ', hf, ':', mf, ':', sf, '.', hundf);
    if hund0 > hundf then {If starting hund > ending hund}
      begin {if} {then we must fix it so}
        sf := sf-1; {borrow from seconds...}
        hundf := hundf+100; {and put onto hunds.}
      end; {if} {Now hundf > hund}
    if s0 > sf then
      begin {if}
        mf := mf-1;

```

```

        sf := sf+60;           {And a similar fix for seconds!}
    end; {if}
    if m0 > mf then
    begin {if}
        hf := hf-1;
        mf := mf+60;         {And another one for the minutes!}
    end; {if}                 {Skip the one for hours}
    write(' ', iter, ' iterations on ', MAXQBITS, ' bits took ');
    write(60*(60*(hf-h0)+(mf-m0))+(sf-s0)+(hundf-hund0)/100:5:2);
    writeln(' seconds');
    end; {case Experimental}
    'I', 'i': Initialize(CurState);
else
    if not((ch[1] = 'Q') or (ch[1] = 'q')) then
    begin
        writeln(' Illegal operation: ', ch[1]);
        ShowMenu;
    end;
    end; {case}
    until (ch[1] = 'Q') or (ch[1] = 'q');
end. {Main}

```

C.3 Adder.q

Toffoli

a
b
c

Controlled-Not

a
b

Toffoli

b
d
c

Controlled-Not

b
d

Controlled-Not

a
b

Quit

%%

First two bits are bits to add. Third is the sum, which can contain carry from a previous adder. Fourth bit is the carry, and must start out as zero.

C.4 Adder2.q

Toffoli

b
d
f

Controlled-Not

b

d

Controlled-Not

d

g

Controlled-Not

b

d

Toffoli

a

c

e

Controlled-Not

a

c

Toffoli

c

f

e

Controlled-Not

c

f

Controlled-Not

a

c

RoundOff

Quit

%%

This is a procedure to add two-bit binary numbers. It can add 0,1,2,3 to 0,1,2,3 to get 0,1,2,3,4,5,6, and theoretically 7. The bits should be encoded as follows:

$$\begin{array}{r} A B \\ + C D \\ \hline E F G \end{array}$$

where A,B,C,D are arbitrary and E,F,G should initially be set to zero.

Appendix D

An Interactive Session

The following is an example of an interactive session with QCompute. The example follows the `Adder2.q` file found in Appendix C.4. This quantum gate network performs two-bit addition on seven qubits. In this particular example, the qubits start in the state

$$\begin{aligned} |A\rangle &= |0\rangle + |1\rangle \\ |B\rangle &= |0\rangle + |1\rangle \\ |C\rangle &= |0\rangle + |1\rangle \\ |D\rangle &= |0\rangle + |1\rangle \\ |E\rangle &= |0\rangle \\ |F\rangle &= |0\rangle \\ |G\rangle &= |0\rangle. \end{aligned} \tag{D.1}$$

These are added according to the operation

$$|AB\rangle + |CD\rangle = |EFG\rangle \tag{D.2}$$

where the registers are interpreted by reading the digits A, B, \dots, G in binary. The overall equation is stored in the simulator by combining it into one large register, where the qubits are placed as

$$|ABCDEFG\rangle. \tag{D.3}$$

```
Enter complex value for qubit A in |0> 1 0
Enter complex value for qubit A in |1> 1 0
Enter complex value for qubit B in |0> 1 0
Enter complex value for qubit B in |1> 1 0
Enter complex value for qubit C in |0> 1 0
```

```

Enter complex value for qubit C in |1> 1 0
Enter complex value for qubit D in |0> 1 0
Enter complex value for qubit D in |1> 1 0
Enter complex value for qubit E in |0> 1 0
Enter complex value for qubit E in |1> 0 0
Enter complex value for qubit F in |0> 1 0
Enter complex value for qubit F in |1> 0 0
Enter complex value for qubit G in |0> 1 0
Enter complex value for qubit G in |1> 0 0
State 0=0000000: 0.250 + i 0.000 Probability: 6.250%
State 8=0001000: 0.250 + i 0.000 Probability: 6.250%
State 16=0010000: 0.250 + i 0.000 Probability: 6.250%
State 24=0011000: 0.250 + i 0.000 Probability: 6.250%
State 32=0100000: 0.250 + i 0.000 Probability: 6.250%
State 40=0101000: 0.250 + i 0.000 Probability: 6.250%
State 48=0110000: 0.250 + i 0.000 Probability: 6.250%
State 56=0111000: 0.250 + i 0.000 Probability: 6.250%
State 64=1000000: 0.250 + i 0.000 Probability: 6.250%
State 72=1001000: 0.250 + i 0.000 Probability: 6.250%
State 80=1010000: 0.250 + i 0.000 Probability: 6.250%
State 88=1011000: 0.250 + i 0.000 Probability: 6.250%
State 96=1100000: 0.250 + i 0.000 Probability: 6.250%
State 104=1101000: 0.250 + i 0.000 Probability: 6.250%
State 112=1110000: 0.250 + i 0.000 Probability: 6.250%
State 120=1111000: 0.250 + i 0.000 Probability: 6.250%

```

```

(N)ot gate (1-bit gate) (1)-bit arbitrary gate
(C)ontrolled-Not gate (2)-bit arbitrary gate
(T)offoli Gate (D)eutsch gate
(E)xperimental gate construction (R)ound off and (V)iew
(I)nititalize new quantum state (Q)uit

```

```

Enter an option: Toffoli
Which is the first control bit? b
Which is the second control bit? d
Which bit is being controlled? f
Enter an option: Controlled-Not
Which bit is the control bit? b
The controlled bit? d
Enter an option: Controlled-Not
Which bit is the control bit? d
The controlled bit? g
Enter an option: Controlled-Not
Which bit is the control bit? b
The controlled bit? d
Enter an option: Toffoli
Which is the first control bit? a
Which is the second control bit? c
Which bit is being controlled? e
Enter an option: Controlled-Not
Which bit is the control bit? a
The controlled bit? c
Enter an option: Toffoli
Which is the first control bit? c
Which is the second control bit? f
Which bit is being controlled? e

```

Enter an option: Controlled-Not
 Which bit is the control bit? c
 The controlled bit? f

Enter an option: Controlled-Not
 Which bit is the control bit? a
 The controlled bit? c

Enter an option: RoundOff

| | | | | |
|-------|--------------|------------------|--------------|--------|
| State | 0=0000000: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 9=0001001: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 18=0010010: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 27=0011011: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 33=0100001: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 42=0101010: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 51=0110011: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 60=0111100: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 66=1000010: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 75=1001011: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 84=1010100: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 93=1011101: | -0.250 + i 0.000 | Probability: | 6.250% |
| State | 99=1100011: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 108=1101100: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 117=1110101: | 0.250 + i 0.000 | Probability: | 6.250% |
| State | 126=1111110: | 0.250 + i 0.000 | Probability: | 6.250% |

Enter an option: Quit

Note that the results listed here are only those with a non-zero probability of occurring. Note also that because the probability equals the modulus squared of the wave function, all sixteen of these states have an equal probability of occurring. Interpreted by splitting these states up into registers according to the operation

$$|ABCDEFG\rangle \longrightarrow |AB\rangle|CD\rangle|EFG\rangle \quad (D.4)$$

and then reading them as binary addition equations, we find that these results are the sixteen correct additions, as shown in Table D.1.

| State | Binary Representation | Equivalent Addition Problem |
|-------|-----------------------|-----------------------------|
| 0 | 00 00 000 | $0+0=0$ |
| 9 | 00 01 001 | $0+1=1$ |
| 18 | 00 10 010 | $0+2=2$ |
| 27 | 00 11 011 | $0+3=3$ |
| 33 | 01 00 001 | $1+0=1$ |
| 42 | 01 01 010 | $1+1=2$ |
| 51 | 01 10 011 | $1+2=3$ |
| 60 | 01 11 100 | $1+3=4$ |
| 66 | 10 00 010 | $2+0=2$ |
| 75 | 10 01 011 | $2+1=3$ |
| 84 | 10 10 100 | $2+2=4$ |
| 93 | 10 11 101 | $2+3=5$ |
| 99 | 11 00 011 | $3+0=3$ |
| 108 | 11 01 100 | $3+1=4$ |
| 117 | 11 10 101 | $3+2=5$ |
| 126 | 11 11 110 | $3+3=6$ |

Table D.1: Output of the two-bit adder for non-classical input.

Bibliography

- [And94] Keri L. Anderson. Restrictions on the integer N necessary to secure the RSA public key cryptosystem. Thesis, Brigham Young U., 1994.
- [B⁺95] Adriano Barenco et al. Elementary gates for quantum computation. *Phys. Rev. A*, 52(5):3457–67, November 1995.
- [Bar95] Adriano Barenco. A universal two-bit gate for quantum computation. *Proc. R. Soc. London, Ser. A*, 449(1937):679–83, June 1995.
- [Bar96] Adriano Barenco. Quantum physics and computers [review]. *Contemporary Physics*, 37(5):375–89, September 1996.
- [BDEJ95] Adriano Barenco, David Deutsch, Artur Ekert, and Richard Jozsa. Conditional quantum dynamics and logic gates. *Phys. Rev. Lett.*, 74(20):4083–86, May 1995.
- [Ben95] Charles H. Bennett. Quantum information and computation. *Phys. Today*, pages 24–30, October 1995.
- [Bra94] G. Brassard. Cryptography column — Quantum computing: The end of classical cryptography? *SIGACT News*, 25(4):15, 1994.
- [Bra95] Samuel L. Braunstein. Quantum computation: A tutorial. Available at chemphys.weizmann.ac.il/~schmuel/comp/comp.html, August 1995.
- [Cle89] R. Cleve. *Reversible Programs and Simple Product Ciphers*. PhD thesis, U. of Toronto, 1989. In *Methodologies for Designing Block Ciphers and Cryptographic Protocols*.
- [DBE95] David Deutsch, Adriano Barenco, and Artur Ekert. Universality in quantum computation. *Proc. R. Soc. London, Ser. A*, 449(1937):669–77, June 1995.

- [Deu89] David Deutsch. Quantum computational networks. *Proc. R. Soc. London, Ser. A*, 425:73–90, 1989.
- [DiV95] David P. DiVincenzo. Two-bit gates are universal for quantum computation. *Phys. Rev. A*, 51(2):1015–22, February 1995.
- [DJ92] David Deutsch and R. Jozsa. Rapid solution of problems by quantum computation. *Proc. R. Soc. London, Ser. A*, 439:553–58, 1992.
- [Fah93] Paul Fahn. Answers to frequently asked questions about today's cryptography. Technical report, RSA Laboratories, September 1993. Version 2.0, draft 2f.
- [Fey86] Richard P. Feynman. Quantum mechanical computers. *Found. Phys.*, 16(6):507–31, 1986.
- [Fol95] Tim Folger. The best computer in all possible worlds. *Discover*, pages 91–99, October 1995.
- [Fow89] Grant R. Fowles. *Introduction to Modern Optics*. Dover Publications, second edition, 1989.
- [Gar95] Lynn E. Garner. Public key cryptography. Talk given at Math Club Symposium, BYU, October 1995.
- [Gas96] Stephen Gasiorowicz. *Quantum Physics*. John Wiley & Sons, second edition, 1996.
- [Gro96] Andrew S. Grove. The connected PC. Available at www.intel.com/pressroom/archive/releases/asgspeech.htm, May 1996.
- [HR96] Serge Haroche and Jean-Michel Raimond. Quantum computing: Dream or nightmare? *Phys. Today*, 49(8):52–52, August 1996.
- [Joh97] Zerubbabel A. Johnson. The simulation of a quantum computer on a classical computer. Thesis, Brigham Young U., 1997.
- [Lev97] Barbara Goss Levi. Trap holds condensates of two different spin states at once. *Phys. Today*, 50(3):18–19, March 1997.

- [Lib80] Richard L. Liboff. *Introductory Quantum Mechanics*, chapter 11, pages 413–90. Holden-Day, 1980.
- [M⁺95] C. Monroe et al. Demonstration of a fundamental quantum logic gate. *Phys. Rev. Lett.*, 75(25):4714–17, December 1995.
- [Met96] Cade Metz. Intel pushes Pentium Pro. *PC Magazine*, 15(14):36, August 1996.
- [MW96] Christopher Monroe and David Wineland. Future of quantum computing proves to be debatable. *Phys. Today*, 49(11):107–8, November 1996.
- [Poo92] Lon Poole. Inside the processor. *MacWorld*, 9:136–43, October 1992.
- [Sho94] Peter W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proc. 35th Annual Symposium on FOCS*, pages 124–34. IEEE Computer Society Press, 1994.
- [Sim94] D. Simon. On the power of quantum computation. In *Proc. 35th Annual Symposium on FOCS*, page 116. IEEE Computer Society Press, 1994.
- [SS96a] Phillip F. Schewe and Ben Stein. Schrödinger’s cat-ion. *SPS Newsletter*. 29(1):4, September 1996.
- [SS96b] Phillip F. Schewe and Ben Stein. World’s fastest computer. *SPS Newsletter*, 29(1):4, September 1996.
- [Tof80] T. Toffoli. Reversible computing. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, pages 632+. Springer, 1980.
- [Win96] Jeffrey Winters. Quantum cat tricks. *Discover*, page 26, October 1996.