Jean François Van Huele

The Simulation of a Quantum Computer on a Classical Computer

by

Zerubbabel A. Johnson

July 1997

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

Advisor: Jean-François Van Huele

Signature: JF Van Huele

Honors Dean: Steven E. Benzley

Signature: S. E. Benzley

Abstract

Recent developments in physics and theoretical computer science indicate the potential for highly efficient computing using quantum principles. Specific problems of interest out of the reach of current computers, such as the factoring of very large numbers, would become feasible using quantum techniques. I give a brief background on quantum computing, and examine its fundamental concepts. I cover these concepts in comparison and contrast with the concepts fundamental to conventional computing. In order to provide a way of exploring quantum computing, I write a program for a classical computer that simulates a quantum computer. The writing of this simulation provides a means of comparison between quantum and classical computing, and provides a platform for the exploration of quantum computing. Because quantum computing includes conventional computing, however, significant limitations are present in the simulation. I present the details of these limitations, and suggest possible reasons for their existence. I also present results obtained in the execution of the simulation, and provide suggestions for the use and expansion of the simulation.

Table of Contents

List of Tables

Acknowledgments

I would like to thank Jean-François Van Huele and Damian Menscher for assistance in research done for this thesis and in the preparation of the thesis itself. I would also like to thank the Physics Department at Brigham Young University which provided support for this research.

1 Introduction

Quantum computing is a relatively new and rapidly growing field. As an application of quantum physics to computer science, its focus is on overcoming current limitations on the abilities of computers. To do so, both new computational hardware and new computing algorithms based on quantum mechanical principles are envisioned.

This work is a report on my attempts at understanding the fundamental principles behind quantum computing. The best way to understand the workings of a quantum computer would be to observe one directly. However, as explained below, constructing an operating quantum computer is currently impractical. A natural alternative is to build a simulation. While previous research into quantum computing has used mathematical models, I chose to simulate a quantum computer using a conventional computer. That is, I wrote a program for a conventional digital computer that simulates quantum computing information and operations. It is this simulation effort that I report on here.

In this first chapter I present the motivations for this work and give an outline of it. The second chapter is a brief background on the field of quantum computing in general. As this work contains only an overview of some of its basic concepts, the background gives a broader outlook on the field, both on its development and future prospects as well as its current difficulties and points of research. In addition, as much of this work is based on the ideas of others, I give references to previous work which I used in my research.

The third chapter covers the implementation of information in both the digital and quantum computing models. As the quantum information model has both strong parallels and distinct differences with the digital information model, the two are presented together, along with comparisons where appropriate. In addition, I give an explanation of the notation standard to the quantum model. The fourth chapter uses a similar format to cover the operations used in both models. Again, I compare and contrast the digital and quantum models, and detail the notation for quantum computing operators.

In the fifth chapter I present the central issue of this work, the task of simulating a quantum computer on a conventional computer. I review the parallels and differences in the two models which pertain to the simulation, and establish some methods available to handle them. The methods I actually used or experimented with in my program are explained, and I discuss their mutual advantages and disadvantages. In this chapter I also review the methods used by Damian Menscher [1] in his efforts to solve the same problem.

The sixth chapter is a presentation of the results. I discuss the strengths and the limitations of my program, both from a general and a numerical point of view. I detail the errors I encountered in my simulation, along with some possible explanations. I compare my findings with Damian Menscher's [1], noting strengths and weaknesses of both programs. In the seventh chapter I give my conclusions. I review briefly the overall implications of my research, and I identify some possibilities for continued research.

2 Background

The field of quantum computing comes as a direct outgrowth of the field of computer science. As conventional computing is based on classical mechanics (and is referred to in quantum computing literature as classical), quantum computing proposes to utilize quantum mechanics to perform computations. Logistically the two approaches are similar. In both approaches, a set of information, divided into basic units of information, or bits, is subjected to a set of operations, or gates, which change the information in well-defined ways. In contrast, however, the classical and quantum computers use different models for the implementation of information and operations, as will be detailed in this work.

The first step from the classical model of computing to the quantum model was in the recognition that classical computing could be made reversible [2]. A basic requirement of any computational system is that there exist a set of operations that are adequate; that is, a set of gates which are sufficient for the representation of any arbitrary operation. If a single gate fulfills that requirement, that gate is said to be universal; the NAND gate is an example of a gate universal to the conventional computer. For reversible computations, Fredkin and Toffoli [3] demonstrated the existence of a universal gate operating on three bits. The application of reversibility to computing led to the possibility of using quantum systems for computation. The evolution of elementary quantum systems is usually described by a unitary (reversible) operator in a Hilbert space. Consequently, implementations of universal reversible three-bit gates were proposed [4] using quantum systems.

The implementation of computation using quantum systems led to the possibility of using quantum properties as a fundamental part of the computation [2, 5]. Deutsch [6] proposed the use of distinctly quantum phenomena in computation to obtain a higher efficiency in simulating quantum systems. He also suggested that the use of quantum phenomena might also be able to solve some conventional problems faster than classical methods. Shor [7] subsequently used Deutsch's methods to develop quantum computing algorithms for factoring numbers and calculating logarithms. Shor's algorithms are

significant in that they are, in theory, significantly faster than any algorithms used by classical computers. Shor's discovery established the ability of the theoretical quantum computer to operate at a higher efficiency than the classical computer, and led to the level of interest quantum computing currently enjoys.

A subject of interest in quantum computing was the search for a universal quantum gate. Deutsch [8] showed that a simple modification of the Toffoli [9] gate was universal for the quantum computer. While Deutsch's gate operated on three quantum bits, later papers showed that a universal two-bit gate exists [10], and even that a wide variety of universal two-bit gates exist [11].

With the basic properties of the quantum computer established and universal gates identified, subjects of current research are in developing practical aspects and in countering potential problems. Specifically, in developing practical aspects, work is being done in the designing of quantum computing networks and in the building of quantum computing components. In designing computing networks two subjects of interest are the decomposition of larger operations into component gates [2, 12] and the development of networks that efficiently implement Shor's algorithms [13, 14]. In the building of actual quantum computing gates a variety of methods are being used, some with limited success [15, 16, 17].

Countering potential problems is the subject of most interest at this point. While some actual implementations of quantum computing theory have been proposed, and a few prototype gates have been built, fundamental problems remain in the quantum phenomenon of decoherence. In a quantum computer, decoherence would randomly change stored information [18], resulting in the regular appearance of errors. While some believe that the problem of decoherence renders the quantum computer impractical [19], others are optimistic that it can be overcome [13, 16, 18]. In the absence of solid evidence verifying one viewpoint or the other, however, much of the current effort in quantum computing is toward developing efficient error correction code [12, 20].

3 Information Models

As explained above, the field of quantum computing comes as an outgrowth of the field of computer science. In the literature on quantum computing, conventional computers are referred to as classical, in reference to the fact that they utilize phenomena associated with classical mechanics. The quantum computer's name appropriately reflects the fact that it is based on quantum mechanics.

In the development of quantum computers, some effort has been made to retain a link with conventional computer science. In particular, the notation used in depicting information and gates parallels that used in computer science. These parallels highlight similarities that do exist between the two fields as well as differences that distinguish them. This comparison of quantum computing with classical computing is not only a fairly standard approach, but bears significance in my attempt to use one to simulate the other. As such, I will use the following approach in my presentation of the information and operation models: the format for the classical approach will be presented first, followed by the format for the quantum approach.

3.A Quantum Bits Versus Classical Bits

3.A.1 Classical Bits

In classical computing the fundamental unit of information is the bit. The bit is binary; that is, it always has one of two distinct values, called on or off, labeled as one or zero. Information about a two-state system can be stored on one bit; larger pieces of information are stored on groups of bits. In particular, numbers are represented using the binary numbering system, with the range of possibilities being limited to integers within a range that is exponential in the number of bits. For instance, a set of eight bits, commonly called a "byte," has a range of 256 possible values. Since each component bit

in the computer can only be in one state at any given time, the computer's information, viewed as a whole, can consequently be in only one state at a given time.

3.A.2 Quantum Bits

The quantum computer stores its information through the use of quantum states. This model has some similarities to the classical computing model; the information is stored on basic units of storage called qubits. Each of these qubits has a finite number of basis states available; in most quantum computing models this is set at two available states to parallel the classical binary system. Again in parallel with the classical model, the available states are labeled as one and zero, or, less commonly, up and down. One qubit can hold information about a quantum two-state system; larger pieces of information are stored on groups of qubits.

3.B Notation

3.B.1 Dirac Notation

Several written representations of quantum states are available. The most general of these is Dirac notation, where a state N is written as $|N\rangle$. Computational states are generally represented in binary; for instance, a portion of memory representing decimal 13 would be 1101 in binary, and so the state would be written as $|1101\rangle$. This particular state would occupy four bits of memory. Individual bits are referred to by counting from the right, starting at zero. In the above example the first order bit would be in the $|0\rangle$ state, and the zeroeth, second, and third bits in the $|1\rangle$ state. In Dirac representation occupation coefficients are placed to the left of the state; the coefficient a_N of a state $|N\rangle$ would be written as $a_N|N\rangle$.

3.B.2 Matrix Representation

Most of the literature on quantum computing, however, chooses to use the matrix representation. In this representation, the entire state is depicted as being a column vector, with column elements corresponding to the available basis states. The occupation, or complex coefficient, of each of the basis states is found in the corresponding place. The available states themselves are given in binary. For instance, the numbers zero through seven could form the basis for a matrix representation; an arbitrary state

$$|\Psi\rangle = \alpha\,|000\rangle + \beta\,|001\rangle + \gamma\,|010\rangle + \delta\,|011\rangle + \varepsilon\,|100\rangle + \xi\,|101\rangle + \eta\,|110\rangle + \mu\,|111\rangle \quad (1)$$

would be, in vector form

$$|\Psi\rangle = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \\ \varepsilon \\ \xi \\ \eta \\ \mu \end{pmatrix} \quad (2)$$

If a piece of memory was in the state $|110\rangle$ (corresponding to decimal six), that state would look, in vector form

$$|110\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad (3)$$

3.C Superposition and Entanglement in Quantum Information

3.C.1 Superposition

The primary difference between the classical and quantum models, however, is in the quantum effect of superposition. Each qubit can occupy not only one of the two available states, but can occupy both states to a limited extent simultaneously. This occupation of states is analog; while it has a limited range, it may occupy any value within that range. In addition, the number specifying the occupation can be complex, with both a magnitude and a phase. If the occupation is of one state only, then the system is said to be in a basis state; if not, it is said to be in a superposition. For instance, a single qubit could be in one of several states:

$$\begin{array}{c} |0\rangle \\ \text{or } e^{i\phi}|0\rangle \\ \text{or } \alpha|0\rangle + \beta|1\rangle \end{array} \tag{4}$$

In the first instance the qubit is in a basis $|0\rangle$ state; in the second it is also in a basis $|0\rangle$ state, but with an added phase; in the third it is in an arbitrary superposition of the $|0\rangle$ and $|1\rangle$ states, with α and β complex numbers. A constraint on the coefficients is that the sum of their modulus squared be equal to unity; that is, for an arbitrary state

$$|\Psi\rangle = a_1|u_1\rangle + a_2|u_2\rangle + a_3|u_3\rangle + \ldots + a_n|u_n\rangle \tag{5}$$

the coefficients must satisfy

$$|a_1|^2 + |a_2|^2 + |a_3|^2 + \ldots + |a_n|^2 = 1 \tag{6}$$

3.C.2 Entanglement

In systems of more than one qubit, however, the quantum phenomenon of entanglement mandates the model used for viewing the information of the system. In a quantum system composed of subordinate elements, it is the state of the system as a whole that bears relevance. That is, in such a system it may not be meaningful to inquire after the state of a subordinate element; that element exists as a part of the whole rather than as an individual entity. The basis states that are available are those which cover the entire system; the degree of freedom is on the level of the system as a whole rather than on its constituent elements. For instance, in a system with three qubits the state vector might look like

$$\begin{pmatrix} 0 \\ 0.193 \\ 0.580i \\ 0 \\ 0.774 \\ 0 \\ 0.097 + 0.135i \\ 0 \end{pmatrix} \tag{7}$$

In this case, the state of the system is clearly defined, but the state of the three individual qubits is not; it is not possible to identify the state of, say, the first qubit.

3.C.3 Composition and Decomposition

The implication of this for the quantum computer is that the computer's state must be viewed as a whole, rather than by examining individual qubits. In the classical computer, both the state of the computer as a whole and the state of its individual bits bear relevance, the bit as having the degree of freedom and the system as being

constructed from constituent bits. In the quantum computer, the system carries the degree of freedom, and the individual qubits do not necessarily have a definable state.

In particular, the state of a quantum computer can be constructed from constituent qubit states, while the reverse process may not be possible. As conventional computers have only one state, composition and decomposition are trivial operations. For the quantum computer, the value of a system's basis state can be found by simply multiplying together the qubit basis states of which it is composed. In a simple two qubit system, the two qubits may occupy an arbitrary superposition of the up and down states:

$$\begin{aligned} &\alpha|0\rangle + \beta|1\rangle \\ &\gamma|0\rangle + \delta|1\rangle \end{aligned} \tag{8}$$

The state vector that defines this system is then:

$$\begin{pmatrix} \alpha\gamma \\ \beta\gamma \\ \alpha\delta \\ \beta\delta \end{pmatrix} \tag{9}$$

In light of the above constructions, it may occasionally be possible to decompose a state vector into constituent substate vectors. In some cases, however, this is clearly not possible, as in the apparently simple case below:

$$\frac{1}{\sqrt{3}}\begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{10}$$

Because of the zero in the $|00\rangle$ state, the status of individual bits cannot be determined. In other words, no α, β, γ, δ can be found such that $\alpha\gamma = 0$, $\beta\gamma = 1$, $\alpha\delta = 1$, $\beta\delta = 1$.

3.D Measurement in Classical and Quantum Systems

An additional distinction between the classical and the quantum information models is in the measurement of the state of the system. In a classical computer, the act of measuring a bit to determine its state has a negligible effect on the state of the bit. Therefore, the bits can be examined as often as necessary with no effect on the information as a whole.

In the quantum computer, however, as the information is stored in a quantum system, the act of measurement does interfere with the state of the system. In particular, the system will be measured as being in only one state. The probability of a given state being measured is the modulus squared of its (complex) occupation value, or coefficient. In this way individual qubits can be measured, although they would only be found to be in the $|0\rangle$ or $|1\rangle$ state. All information about other states the system was occupying is permanently lost; the system is said to have collapsed to one value. Thus, the system can only be measured once before losing the depth of its quantum characteristics. The implication of this for quantum computing is that a measurement of the system can only be made once, after the calculations are finished. However, while the inherent parallelism of the system is not directly accessible, the action of gates is not considered a measurement. Consequently, the system does maintain its superposition through series of operations, which gives the quantum computer its distinct qualities.

4 Operation Models

In both the classical and quantum computing models, the computer's operation involves performing some sort of defined transformation on some input information. Unlike the information part of the model, however, the parallel between quantum and classical computing with respect to the transformations, or operations, is not strong. By comparison, classical computing remains a subset of quantum computing with respect to operations as well as information. The operations are identical, but the representations of operations are vastly different.

4.A Operation Representation

In keeping with the matrix representation most commonly used for the quantum computer, the operations themselves are represented as matrices. As in the quantum formalism, the state is given as a column matrix and the operation is given as a square matrix multiplied to the left of the state. A sample one-bit operation that flips a bit might look like this:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix} \tag{11}$$

In a sense, the quantum computing operators can be seen as truth tables. Matrix elements select a source basis state and specify how that state will contribute to a selected destination basis state. Columns in the operator matrix specify how a given source basis state will contribute to the destination state as a whole. In the following example, an arbitrary operator acting on a basis state demonstrates the relationship between matrix elements and basis states

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_{02} \\ a_{12} \\ a_{22} \\ a_{32} \end{pmatrix} \tag{12}$$

4.B Operator Properties

With this notation, the classical operations can be represented in operator form, though some significant differences remain. Primarily, in classical operations there is usually a fundamental loss of information; in classical gates, two bits are often used as input for only one bit of output. Because of this loss of information, the process is irreversible. Classical gates can be adapted to the quantum formalism, however, by adding one or more output bits, thus preserving information and allowing reversibility. Reversible classical computing as developed by Fredkin and Toffoli [3] uses this approach to maintain reversibility. In addition, for a classical gate to be adapted to the quantum domain, it must meet criteria unique to quantum operators.

In order for a quantum computing operation to be implemented by a physical process, the operator falls under the same constraints as quantum mechanical operators. Reversibility is one of those properties; however, the matrix operator must also be unitary. In practice, this maintains normalization from the initial to the final state vector. As with the state vector elements, matrix elements can be complex.

The most distinct difference in the quantum and classical operators is in the type of state the two can operate on. Classical operators can only act on a basis state; they cannot operate on a superposition. Quantum operators, on the other hand, do operate on superpositions. When a quantum operator acts on a superposition, it automatically acts on each element of the superposition. Because it acts on multiple pieces of information simultaneously, the quantum operation is a truly parallel computation. While only one element of the superposition will eventually be measured, the parallel processing provides for the possibility of interference effects. These interference effects are what render

Shor's algorithms [7] significantly faster than any algorithms known for classical computers.

4.C Universal Gates

The XOR gate, although not universal to classical digital computing, can be made reversible by preserving one of the input bits. The reversible XOR gate can be represented in the $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ basis as follows

$$XOR = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{13}$$

The Toffoli gate [9], universal to reversible computing, is a three-bit variation on the reversible XOR gate. In effect, it flips the third bit if the first two are in the $|1\rangle$ state. In the $\{|000\rangle, |001\rangle, \ldots, |111\rangle\}$ basis it is

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{14}$$

The first gate universal to quantum computing was discovered by Deutsch [6]; it is a variation on the Toffoli gate, and is also a three-bit, or rather three qubit, gate. It is, in the $\{|000\rangle, |001\rangle, \ldots, |111\rangle\}$ basis

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & i\cos\theta & \sin\theta \\ 0 & 0 & 0 & 0 & 0 & 0 & \sin\theta & i\cos\theta \end{pmatrix} \tag{15}$$

with θ/π irrational. The first of the universal two-bit gates for quantum computing was discovered by Barenco [10], and is a variation on the Deutsch gate. In the $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ basis it is

$$A(\phi, \alpha, \theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{i\alpha}\cos\theta & -ie^{i(\alpha-\phi)}\sin\theta \\ 0 & 0 & -ie^{i(\alpha+\phi)}\sin\theta & e^{i\alpha}\cos\theta \end{pmatrix} \tag{16}$$

with ϕ, α, and θ are fixed irrational multiples of π and of each other.

5 Simulation

Our problem of interest is the simulation of a quantum computer on a classical computer. In particular, our goal is to write a program on an existing platform that simulates both the quantum computer information storage and operations. The result we hope to obtain is a program that executes this simulation within the limits of available computing resources, is flexible enough to allow some experimentation, and can give concrete data on errors and resource usage. In the programming of this simulation, however, several unique challenges present themselves. Most of the difficulties encountered involve the fact that the operations available to us on the classical computer are a subset of the operations we are attempting to simulate. The difficulty could be compared to attempting to perform multiplication or division using only addition; the simulation is possible but not necessarily efficient.

5.A Programming Approach

Writing the program can be separated into two significant tasks. The first is the most difficult, that of devising an algorithm, or set of instructions, to perform the desired operations. The second task is the actual coding of the algorithm, where the set of instructions is written out in computer language, in a format ready for compilation. As the actual programming language used is not as important as the algorithm, I will not make reference to any details specific to the language used. I refer the reader to Appendix A for a complete listing of the program, as well as a few notes specific to the programming language. Rather, general programming approaches serve our purpose in providing techniques for implementing concepts involved in the simulation. While the construction of the algorithm usually precedes the coding of the program, certain issues specific to the coding have a profound effect on the algorithm, and consequently will be discussed first.

As discussed in Chapters 3 and 4, the quantum computer contains both information and operators. The most direct method of representing these in a program is

through the use of arrays, which fortunately are a direct counterpart to matrices. Although the quantum matrices have complex elements, this again is not a problem, as arrays can be built of complex elements, and computing operations for complex numbers are available. In implementing complex numbers, however, the first of the limitations inherent to the simulation becomes apparent. Because of the digital nature of the computer, true analog numbers cannot be represented; rather, the numbers are digitized, and so have a limited accuracy. In particular, complex numbers are generally represented using floating point numbers, wherein an exponent and a specific number of significant figures are retained, with the remainder being discarded. The limited accuracy of these numbers results in rounding errors, which typically are small but have the potential for causing larger problems.

5.B Scaling the Simulation

While implementing matrices as arrays has no inherent problems, the size of the arrays is a potentially critical problem. The column matrices used to represent the quantum state grow exponentially with the number of qubits; in particular, the size of a column vector corresponding to N qubits is 2^N elements. The problem is that the program must use computer memory for each element of the array; with the number of elements growing exponentially in the number of qubits, the program will be capable of simulating only a very small number of qubits. Unfortunately, this problem is inherent in the simulation. The quantum computer uses superposition to carry an exponential amount of information on its qubits, while the classical computer is inherently linear in its memory model.

A more severe problem is in the representation of the operator matrices. Quantum computing operators are defined with respect to their action on a specified number of qubits, often three or less. Specifically, operator matrices are given in terms of the set of basis states corresponding to the number of qubits acted upon. When a larger number of qubits, and hence a different set of basis states, is used, the appearance of the operator matrix is not well defined. Therefore, before an algorithm for applying an operator can

be designed, a decision must be made as to how relatively small operators are to act upon relatively large sets of qubits.

The approach used by Damian Menscher [1] in his simulation is to create a new operator matrix appropriate to the new set of basis states, but based on the original matrix. In this approach, the new matrix is square in the size of the new set of basis states; the original matrix elements are repositioned in the expanded matrix so as to retain the effect of the original operation with respect to specific qubits (specific details on this transformation are given in [1]). The primary strength of this approach is that the algorithm for applying the matrix is fairly straightforward; the operator matrix and the state matrix are simply multiplied together to produce the result. In addition, since operator matrices can themselves be multiplied together, several operations can be performed at once. Its primary weakness is in the usage of memory. Since the size of the expanded operator array is quadratic in the size of the state array, the memory usage for one operation on N qubits is 2^{2N}. The time used for executing operations can also become a limiting factor, as each application of a matrix to a state requires a number of machine instructions on order of the size of the operation matrix (2^{2N} for N qubits).

Rather than take this approach, however, I chose to fix the size of the operator matrices. In particular, I chose to fix the gate size at two bits, as most quantum computing gates are two-bit gates or can be decomposed [2, 12] into two-bit gates. By fixing the operator, necessary adjustments for disparity in the size of the operation and the information would have to be made elsewhere, namely in the information.

A two-bit gate's action is defined only within the set of two-bit basis states. A basis state from a basis having a large number of qubits, however, can be decomposed into basis states involving fewer numbers of qubits. For instance, the basis state $|0110100\rangle$ from a seven qubit system can be decomposed into a three qubit basis state and a four qubit basis state to give $|011\rangle|0100\rangle$; it can also be decomposed into other combinations of basis states ($|01\rangle|10\rangle|100\rangle$, for instance). This is similar to the way in which other quantum states can be decomposed; the spin basis states for a system of particles, for instance, can be decomposed into spin basis states corresponding to individual particles. Since a two-bit gate's action is dependent only on the condition of

the two bits it applies to, the remainder of the bits in a larger set of information are irrelevant. If the basis states in the larger set of information were to be decomposed into smaller basis states, some of the smaller basis states would contain only bits unaffected by the operation, and could safely be set aside. It is possible to decompose these larger basis states in such a way that only two smaller basis states are formed: a two bit basis state that contains the two bits relevant to the operation, and a basis state containing the remainder of the bits. To apply a two-bit gate to a larger basis state, then, a two-bit basis state is decomposed from, or extracted from, the larger basis state, and the gate is applied to the extracted state.

5.C Executing the Gates

Using the above approach, a two-bit gate can be applied to a state involving more than two qubits. As explained in Chapter 3, the state can be expressed in terms of basis states with weighted coefficients. The gate is successively applied to each basis state, and the result is weighted by the appropriate coefficient. In reconstructing the result, however, some care must be taken. The gate in question may be operating on qubits that are not adjacent; in decomposing the basis state, then, the ordering of bits must be retained. For instance, say a gate was operating on the first and third bits of the basis state $|1101\rangle$; the relevant two bit state would then be $|1_3 0_1\rangle$, and the remainder state would be $|1_2 1_0\rangle$, where the subscripts designate position in the larger state. After acting on the extracted state, the gate will produce an output state, which must then be added to the remainder state, and the larger state is recomposed. For instance, if the gate in the above example were the XOR gate (from Chapter 4), the result would be $|1_3 1_1\rangle$, which, after recomposition with $|1_2 1_0\rangle$, would be $|1111\rangle$. In the case of more complex gates, however, one input state could result in an output as complex as a superposition of four basis states. In this case, each output basis state would be "spliced" into the remainder state, with the final result being a superposition.

To illustrate the overall approach a complete example follows. In this example, the input state

$$\Psi = \frac{1}{\sqrt{2}}(|1110\rangle + |0010\rangle) \tag{17}$$

is in an even superposition of the states $|1110\rangle$ and $|0010\rangle$. In this instance, the status of the individual qubits is discernible; the zeroeth and first qubits are in the $|0\rangle$ and $|1\rangle$ states respectively, while the second and third qubits are in an even superposition of $|0\rangle$ and $|1\rangle$. The input state is operated on by the gate

$$A(0,0,\tfrac{\pi}{4}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -i\frac{1}{\sqrt{2}} \\ 0 & 0 & -i\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \tag{18}$$

acting on the zeroeth and second input bits. Note that this gate [10] rotates the zeroeth qubit to the extent that the first qubit is in the $|1\rangle$ state. Note also that the gate is not symmetric with respect to input bits, so the ordering of the input bits is important. In this case, the zeroeth bit corresponds to the zeroeth input bit, and the second bit corresponds to the first input bit. First, the basis states are decomposed into relevant and remnant states:

$$\Psi = \frac{1}{\sqrt{2}}(|1_2 0_0\rangle|1_3 1_1\rangle + |0_2 0_0\rangle|0_3 1_1\rangle) \tag{19}$$

The two extract states are then separately operated on by the gate:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -i\frac{1}{\sqrt{2}} \\ 0 & 0 & -i\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ -i\frac{1}{\sqrt{2}} \end{pmatrix} \tag{20}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -i\frac{1}{\sqrt{2}} \\ 0 & 0 & -i\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{21}$$

Both results are then spliced into the remnant:

$$\Psi = \frac{1}{\sqrt{2}}\left(\left(\frac{1}{\sqrt{2}}|1_2 0_0\rangle - i\frac{1}{\sqrt{2}}|1_2 1_0\rangle\right)|1_3 1_1\rangle + (|0_2 0_0\rangle)|0_3 1_1\rangle\right) \tag{22}$$

to obtain the result.

$$\Psi = \frac{1}{2}|1110\rangle - i\frac{1}{2}|1111\rangle + \frac{1}{\sqrt{2}}|0010\rangle \tag{23}$$

The output state is, like the input state, a normalized superposition. However, the operation has transformed the state into an entangled one, where the status of the individual qubits is not discernible.

The primary advantage of this general approach is in the memory and time saved when relatively large amounts of qubits are simulated. Using this approach, a minimal amount of storage is required for the gates (sixteen array elements) and the number of machine instructions executed is on the order of the size of the state array (2^N for N qubits). Conversely, its primary weakness is in the extra time needed when small numbers of qubits are involved, as expanding the operator matrix may be quicker than splitting, evaluating, and recombining each of the individual basis states.

6 Results

In general, my simulation efforts were successful; in particular, the results obtained with my program were in harmony with theoretical predictions. In running the simulation, I was able to observe not only the desired quantum computation, but also the limitations inherent in my simulation. The most notable limitations observed were in program execution time, memory limitations, and rounding errors.

6.A Observed Quantum Effects

In running the program, the most directly observed quantum effect was that of superposition. The program was written specifically to handle a general state that could be any arbitrary combination of basis states. Even the initial state could be set as a superposition. The quantum effect of entanglement was observable as well, in that the final state could not be decomposed into component qubit states. For instance, consider the following operation (the gate used is identical to the one in the Chapter 5 example)

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & -i\frac{1}{\sqrt{2}} \\ 0 & 0 & -i\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 1 \\ -i\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} \tag{24}$$

The initial state contains equal amounts of $|01\rangle$ and $|11\rangle$; by observation, the zero order qubit is in the pure state $|1\rangle$ while the first order qubit is in an equal superposition of $|0\rangle$ and $|1\rangle$. In accordance with the mathematical prediction, the simulation produced a result of $0.7071|01\rangle + (-0.5i)|10\rangle + 0.5|11\rangle$. As can be seen by comparison with the example in section 3.C.3, the result cannot be decomposed into qubit basis states.

6.B Execution Time

I used standard programming procedures to record the execution time of the program. In particular, I used the computer's internal clock to time the processing cycle. The time starts before the program enters the loop used to process the gates, and is recorded after the program exits the loop. As an identical set of instructions is executed regardless of the gate in question, the time used for any gate operation is expected to be fixed. Some variable factors do exist in the computer's internal timing and in the operating system, however, so I used an average over several gates to get a standard time for a gate operation. Running under MS-DOS on a Pentium chip 90 MHz I recorded average gate times as shown in Table 1.

Table 1: Average Times for Q

Qubits	States	Average Time(ms)
2	4	0.430
3	8	0.930
4	16	1.75
5	32	3.36
6	64	6.59
7	128	13.3
8	256	26.3
9	512	53.5
10	1024	108
11	2048	215
12	4096	431

Between individual times a discrepancy of as much as 10% was observed, and under Windows 95 times were around 25% higher. The error in both cases comes from unknown variables in the operating system; the discrepancy in run times between operating systems comes from the larger operating system overhead under Windows 95. As can clearly be seen in Table 1, the time required per gate per basis state is on the order

of one millisecond for every ten states. By way of comparison, a similar computation involving twenty qubits (roughly a million basis states) and one gate would take roughly 100 seconds, or over a minute and a half. The average times for Damian Menscher's simulation QCompute [1] follow. As anticipated in Chapter 5, his algorithm is faster for small numbers of qubits but slower for large numbers; the two are about equal when simulating five qubits.

Table 2: Average Times for QCompute

Qubits	States	Average Time (ms)
2	4	0.394
3	8	0.792
4	16	1.60
5	32	3.27
6	64	6.69
7	128	13.7
8	256	27.8
9	512	58.8
10	1024	122
11	2048	250
12	4096	512

6.C Memory Limitations

The principal memory limitation on the program was from the storage used to record the basis states. Each qubit added to the simulation doubled the program's memory requirements. Since current computers have memory available to programs on the order of 2^{25} bytes, this sets the maximum practical number of qubits at around twenty-five. For functionality considerations, however, in particular functionality on older machines, the maximum number of qubits available was limited to fifteen, giving a total of 32768 basis states. Since the storage of each basis state used twelve bytes of storage, the overall storage requirement for the set of basis states was 384 kilobytes (a kilobyte

being 2^{10} bytes). By comparison, storage requirements for different numbers of qubits would be as set out in Table 2 (note that a gigabyte is 2^{30} bytes).

Table 3: Memory Requirements

Qubits	Storage Needed
5	384 bytes
10	12 kilobytes
15	384 kilobytes
20	12 megabytes
25	384 megabytes
30	12 gigabytes
35	384 gigabytes

Note that twenty-five or more qubits could be simulated by using hard disk drive storage (which currently is on the order of 4 gigabytes); as access to memory on the hard disk drive is significantly slower than access to RAM (memory traditionally used by programs), the program would run significantly, even prohibitively, slower. Memory usage for the gates themselves was much lower than for the states. As each gate is basically sixteen complex matrix elements, each gate took about the same room as sixteen basis states; specifically, each took 130 bytes per gate. In addition, instructions for a gating operation contained only an identifier of the gate to be used and identifiers for the lines operated on. Consequently each instruction occupied twelve bytes of memory, the same as one basis state.

6.D Rounding Errors

In order to obtain information on rounding errors, I allowed for the program to check for normalization over the set of basis states. In addition, I had the program automatically normalize the initial state vector. In the case of trivial input (a basis state, for instance) in conjunction with relatively trivial gates, no rounding error was detected.

With more complicated inputs and gates, however, rounding error appeared both before and after the gate operation. The initial normalization produced an error in the seventh significant figure; following a gate operation, the error appeared in the sixth significant figure. As the variable type used to store the coefficients carries approximately seven significant figures, the normalization error can most likely be attributed to limitations inherent in the type. On the other hand, the error that appeared after gating operations, being an order of magnitude higher, is likely to be a result of rounding off during calculations.

7 Conclusions

The correlation between the output of the program and the theoretical predictions indicates that the program correctly simulates the presence of quantum effects. In addition, the algorithm used to evaluate the quantum computing processes is not one that has been considered independent of this paper. The fact that the results achieved were consistent with theoretical predictions therefore indicates the validity of the approach used. These successes indicate that overall the program is a fair simulation of a quantum computer.

Errors are present in the simulation, but the errors encountered were not unexpected. The presence of these errors does limit the accuracy of the simulation on the one hand, but on the other it verifies the problems present in the transition. The errors present in the simulation can be countered but not completely overcome. For instance, rounding error can be countered by using a larger storage type and by introducing error correction code. While this would result in higher accuracy, precision would still be limited and rounding errors would still occur, though with less of an effect.

The simulation's central problem, memory limitations, is again a limitation that serves to highlight problems inherent in the transition. Memory problems can be eased by allocating more memory or using compression schemes, but memory usage would still rise exponentially with added bits, albeit at a somewhat slower rate. Time problems as well can be reduced by more efficient algorithms; checking for and handling trivial calculations, for instance. In all cases, however, fundamental problems remain.

The persistence of problems inherent in the simulation indicates that the difficulties are at a deeper level than the programming. Indeed, the difficulty is in the fact that the hardware is simply not suited for the calculations performed. As long as the hardware is digital, analog numbers will not be correctly represented; as long as the hardware uses a linear model for instructions, it will be unable to perform the needed calculations without a disproportionate use of time and resources. These limitations highlight the fact that only a true quantum computer will be capable of performing

quantum computing operations with the efficiency and accuracy expected in theoretical work.

Although the simulation fails to match the expected efficiency and accuracy of a quantum computer, it nonetheless has potential for work in quantum computing research. As much of the work done is quantum computing is relatively abstract, the simulation provides a way for more concrete observation of expected results. In addition, the simulation provides a potentially faster way of exploring quantum algorithms and networks than written calculations. The simulation could also be used as a demonstration tool in showing the unique properties and potential benefits of a quantum computer.

As noted above, the simulation could be expanded by adding code, available memory, and so forth; such expansions would provide moderate increases in efficiency and accuracy, but would not overcome fundamental errors. On the other hand, more significant progress could be made against these problems by using different hardware. For instance, analog computers have the potential of more accurately representing analog numbers, avoiding rounding error, and parallel processing could be used to reduce time limitations. These improvements, however, are no more than steps toward a true quantum computer. Only a computer based directly on quantum principles can make full use of the computing power available in the physical world. While such a computer has not yet been built, its advantages, both known and unknown, make its development an issue of paramount importance in both the fields of physics and computer science.

References

[1] D. Menscher, B.S. thesis, Brigham Young University, 1997 (unpublished).

[2] A. Barenco *et al.*, Phys. Rev. A **52**, 3457 (1995).

[3] T. Toffoli and E. Fredkin, Int. J. Theor. Phys. **21**, 219 (1982).

[4] H. F. Chau and F. Wilczek, Phys. Rev. Lett. **75**, 748 (1995).

[5] R. P. Feynman, Found. Phys. **16**, 507 (1986).

[6] D. Deutsch, Proc. R. Soc. London Ser. A **400**, 97 (1985).

[7] P. W. Shor, in *Proceedings of the 35th Annual Symposium on the Foundations of Computer Science* (IEEE Computer Society, Los Alamitos, 1994), p. 124.

[8] D. Deutsch, Proc. R. Soc. London Ser. A **425**, 73 (1989).

[9] T. Toffoli, Mathematics Systems Theory **14**, 13 (1981).

[10] A. Barenco, Proc. R. Soc. London Ser. A **449**, 679 (1995).

[11] D. Deutsch, A. Barenco, and A. Ekert, Proc. R. Soc. London Ser. A **449**, 669 (1995).

[12] D. P. DiVincenzo, (preprint).

[13] A. Barenco, Contemporary Phys. **37**, 375 (1996).

[14] D. Beckman *et al.*, Phys. Rev. A **54**, 1034 (1996).

[15] T. Sleator and H. Weinfurter, Phys. Rev. Lett. **74**, 4087 (1995).

[16] C. Monroe *et al.*, Phys. Rev. Lett. **74**, 4714 (1995).

[17] A. Barenco, D. Deutsch, and A. Ekert, Phys. Rev. Lett. **74**, 4083 (1995).

[18] I. L. Chuang, R. Laflamme, P. W. Shor, and W. H. Zurek, Science **270**, 1633 (1995).

[19] S. Haroche and J.-M. Raimond, Phys. Today **50**, 51 (August 1996).

[20] M. B. Plenio, V. Vedral, and P. L. Knight, (unprinted).

[21] Stony Brook Modula-2 (Stony Brook Software, Thousand Oaks, CA, 1996).

Appendix A : Program Code

My program Q was written in Modula-2 and was compiled on the Stony Brook Modula-2 compiler [21] for both MS-DOS and Microsoft Windows 95 (as a console application). It has three component modules:

Q.MOD (the main program)

QuantumGateLibrary.DEF (the definition file for the library module)

QuantumGateLibrary.MOD (a library of procedures used by Q)

A complete listing of the program's code follows. The program does use some standard code; specifically, some functions and procedures are imported from the Modula-2 ISO standard libraries, with the exception of the timing functions in the main program, which are imported from the Stony Brook library. Exceptions to the Modula-2 ISO standard syntax are noted where they occur. Note also that the program uses a *gatespec.txt* file for gate specifications; a sample follows the code listing.

```
MODULE Q;

(* File     : Q.MOD
   Author   : Z. Johnson
   Compiler : Stony Brook MODULA-2
   ISO      : Partly; see below
   Date     : May 21, 1997 *)

(* General Notes on the Operation of Q:
       This program simulates the operation of an ideal quantum
   computer using matrix format.  This program is written
   specifically for use with two-bit quantum gates; one-bit
   quantum gates can be represented fairly trivially, and larger
   gates can be decomposed into sets of two-bit and one-bit gates.
   The information is represented in terms of basis states; the
   maximum number of qubits is currently hard-coded at 10; the
   number of basis states is 2^(no. of qubits), but calculating the
   upper bound on the basis states is automatically handled, and
   the procedures are perfectly general.  To change the upper bound
   on the number of qubits, simply change the constant MaxStates.
   Note that the gates must be read in from "gatespec.txt"; format
   for this file can be found with the initialization code in the
   QuantumGateLibrary.MOD file.  The program will prompt for all
   relevant information; note that the coefficients for the basis
   states must be entered in standard REAL format.  The input can
   be read in from the "settings.txt" file, which follows a format
   identical to that of the expected input.  Output states are
   written using the WriteFixed procedure, which does round them
```

```
   off; if this is not a desired effect, WriteFloat can be
   substituted.  Output can also be dumped to any text file (which
   will be overwritten); output is as it would appear on the
   screen.  The input state is automatically normalized, and both
   it and the output state are checked for normalization; this is
   to check for rounding errors.  The gate matrices are not,
   however, automatically unitary, and non-unitary matrices will
   affect the normalization on the output state.
*)


FROM QuantumGateLibrary IMPORT Extract, Process, Restore,
                               Reset, State,
                               ExtractState, GateSpec,
                               Initialize, WriteResult;

FROM SWholeIO IMPORT WriteCard, ReadCard;

FROM RealMath IMPORT power;

FROM STextIO IMPORT WriteString, WriteLn, SkipLine;

FROM ElapsedTime IMPORT StartTime, GetTime;

CONST

    MaxStates = 32768;
    MaxGates = 128;

(* Note that the number of basis states is 2 to the Nth power,
   where N is the number of qubits.  The maximum number of gates
   is arbitrary *)

VAR
    initstate, finalstate  : ARRAY [0 .. MaxStates - 1] OF State;

    maxbits, maxstate      : CARDINAL;
    floatbits, floatstate  : REAL;

    gatein                 : ExtractState;
    gateout                : ARRAY [0 .. 3] OF ExtractState;
    gateset                : ARRAY [0 .. MaxGates - 1] OF GateSpec;
    doagain                : BOOLEAN;
    cycle, count, gatehigh : CARDINAL;
    time                   : CARDINAL32;

BEGIN

REPEAT

    (* perform various initializations *)

    Reset(initstate);

    (* get the number of bits, and set the upper bound on the
       array of states *)

    WriteLn;
    WriteString("Please enter the number of bits to work on:  ");
    ReadCard(maxbits); SkipLine; WriteLn;
    floatbits := FLOAT(maxbits);
    floatstate := power(2.0, floatbits);
    maxstate := (TRUNC(floatstate) - 1);

    Initialize(initstate[0..maxstate], gateset, gatehigh);
```

```
    (* process each gating action *)

    (* also, start timing *)

    StartTime;
    FOR cycle := 0 TO (gatehigh - 1) DO

        Reset(finalstate[0..maxstate]);

        (* evaluate the gating action for each basis state *)

        FOR count := 0 TO maxstate DO
            Extract(initstate[count], gateset[cycle], gatein);
            Process(gatein, gateset[cycle], gateout);
            Restore(gateout, initstate[count], finalstate[0..maxstate]);
        END; (* FOR Count *)

        initstate[0..maxstate] := finalstate[0..maxstate];

    END; (* FOR Cycles *)
    time := GetTime();

    (* write the output *)

    WriteString("Elapsed time is ");
    WriteCard(time, 6);
    WriteString(" in milliseconds."); WriteLn;
    WriteResult(finalstate[0..maxstate], doagain);
UNTIL (NOT doagain);

END Q.


DEFINITION MODULE QuantumGateLibrary;

(* File     : QuantumGateLibrary.DEF
   Author   : Z. Johnson
   Compiler : Stony Brook MODULA-2
   ISO      : Partly; see below
   Date     : May 8, 1997 *)

(* The following are useful type declarations; note that
   I have to use conditional compilation to secure my
   type casts.  This is non-ISO standard format,
   but should be trivial to adjust *)

%IF ThirtyTwoBit %THEN

TYPE Bitset       = SET OF CARDINAL[0..31];

%ELSE

TYPE Bitset       = BITSET;

%END

TYPE GateArray    = ARRAY [0..3],[0..3] OF COMPLEX;

                        (* [rows][columns] *)

TYPE GateSpec     = RECORD
                        ID : CARDINAL;
                        FirstBit : CARDINAL;
```

```
                        SecondBit : CARDINAL;
                    END;

TYPE State          = RECORD
                        Bits : Bitset;
                        Coefficient : COMPLEX;
                    END;

TYPE ExtractState = RECORD
                        Coefficient : COMPLEX;
                        Bits : Bitset;
                        FirstBit : CARDINAL;
                        SecondBit : CARDINAL;
                    END;

PROCEDURE Initialize(VAR initialstate : ARRAY OF State;
                     VAR gateset : ARRAY OF GateSpec;
                     VAR gatesetmax : CARDINAL);

PROCEDURE WriteResult(output : ARRAY OF State;
                      VAR repeat : BOOLEAN);

PROCEDURE Extract(instate : State; gate : GateSpec;
                  VAR result : ExtractState);

PROCEDURE Restore(VAR outstates : ARRAY OF ExtractState;
                  archive : State; VAR target : ARRAY OF State);

PROCEDURE Process(instate : ExtractState; gate : GateSpec;
                  VAR outstates : ARRAY OF ExtractState);

PROCEDURE Reset(VAR target : ARRAY OF State);

END QuantumGateLibrary.


IMPLEMENTATION MODULE QuantumGateLibrary;

(* File     : QuantumGateLibrary.MOD
   Author   : Z. Johnson
   Compiler : Stony Brook MODULA-2
   ISO      : Yes
   Date     : May 8, 1997 *)

FROM STextIO IMPORT ReadChar, ReadString,
                    ReadToken, SkipLine,
                    WriteLn, WriteString;

FROM SWholeIO IMPORT ReadCard, WriteCard;

FROM SRealIO IMPORT WriteFixed, WriteFloat, ReadReal;

FROM ComplexMath IMPORT conj, zero;

FROM RealMath IMPORT sqrt;

FROM SeqFile IMPORT ChanId, OpenResults,
                    OpenRead, OpenWrite,
                    Close, old;

FROM StdChans IMPORT StdInChan, StdOutChan,
                     NullChan, SetInChan,
                     SetOutChan;
```

```
IMPORT SeqFile, ChanConsts;

FROM SYSTEM IMPORT CAST;

(* This procedure gets all the pertinent information from
   the user; in particular, it prompts for the number and
   type of gates and for the initial states. *)

(* It is assumed in this procedure that the initial
   state has been properly prepared, so Reset needs to be
   called first.  The set of gates, on the other hand,
   does not need to be initialized, as it will be fully
    specified by this procedure. *)

(* This procedure now takes input from the external file
   "settings.txt" and performs automatic normalization *)

(* Note that the format for "settings.txt" follows the input
   exactly; that is, the set of commands that would normally be
   manually entered in is typed into the file beforehand with
   no changes *)

PROCEDURE Initialize(VAR initialstate : ARRAY OF State;
                     VAR gateset : ARRAY OF GateSpec;
                     VAR gatesetmax : CARDINAL);

    VAR
        j, k, temp         : CARDINAL;
        yesno              : CHAR;
        high               : CARDINAL;
        real, imag, check : REAL;
        readingfile        : BOOLEAN;

    BEGIN

    WriteString("Would you like to read in the settings from a text
    file?");
    WriteLn;
    WriteString("(The settings will be read in from settings.txt)
(y/n):");
    WriteLn;
    ReadChar(yesno); SkipLine;
    IF ((yesno = 'y') OR (yesno = 'Y')) THEN
        readingfile := TRUE;
        OpenRead(filecid, "settings.txt", old, fileresult);
        IF (fileresult = ChanConsts.noSuchFile) THEN
            WriteString("Cannot find settings.txt!  Aborting program . .
. ");
            HALT;
        END;
        SetInChan(filecid);
        dumpcid := NullChan();
        SetOutChan(dumpcid);
    ELSE readingfile := FALSE;
    END;

    WriteString("Please enter the number of gates."); WriteLn;
    ReadCard(gatesetmax); SkipLine; WriteLn;
    high := HIGH(gateset);
    WHILE (gatesetmax > high) DO
        WriteString("Sorry, you can only have ");
        WriteCard(high, 0); WriteString(" gates."); WriteLn;
        WriteString("Try again, please.");
        ReadCard(gatesetmax); SkipLine; WriteLn;
    END;
```

```
    WriteString("Please enter the gate as follows: "); WriteLn;
    FOR k := 0 TO (numberofgates - 1) DO
        WriteString("  ("); WriteCard(k, 0); WriteString(")  ");
        WriteString(possiblegates[k].GateName); WriteLn;
    END;
    WriteLn;

    FOR j := 0 TO (gatesetmax - 1) DO
        WriteString("Please enter the type for gate ");
        WriteCard(j, 0); WriteString(": "); WriteLn;
        ReadCard(gateset[j].ID); SkipLine;
        WriteString("Please enter the first bit of the gate: ");
        WriteLn;
        ReadCard(temp); SkipLine;
        gateset[j].FirstBit := temp;
        WriteString("Please enter the second bit of the gate: ");
        WriteLn;
        ReadCard(temp); SkipLine;
        gateset[j].SecondBit := temp;
    END; (* for *)

    WriteString("The registers will initially be set to 0");
    WriteLn;
    WriteString("unless you specify otherwise."); WriteLn;
    WriteString("Would you like to specify any values? (y/n):");
    ReadChar(yesno); SkipLine;
    WHILE ((yesno = 'y') OR (yesno = 'Y')) DO
        WriteString("Which state would you like to specify?");
        WriteLn;
        WriteString("(Please enter the state in base 10):");
        WriteLn;
        ReadCard(temp); SkipLine;
        IF (temp <= HIGH(initialstate)) THEN
            WriteString("What would you like for its real
coefficient?");
            WriteLn;
            ReadReal(real); SkipLine;
            WriteString("What would you like for its imaginary
coefficient?");
            WriteLn;
            ReadReal(imag); SkipLine;
            initialstate[temp].Coefficient := CMPLX(real, imag);
        ELSE WriteString("That state is out of range."); WriteLn;
        END; (* if *)
        WriteString("Would you like to specify another state?  (y/n)");
        WriteLn;
        ReadChar(yesno); SkipLine;
    END; (* while *)

    IF readingfile THEN
        SetInChan(instdcid);
        SetOutChan(outstdcid);
        Close(filecid);
    END;

    WriteString("Normalizing . . . "); WriteLn;
    Normalize(initialstate);
    WriteString("Checking initial normalization . . . ");
    check := NormalizeCheck(initialstate);
    WriteFloat(check, 4, 6); WriteLn;
    WriteString("Executing, please wait."); WriteLn;

    END Initialize;
```

```
(* This outputs the result of the calculations.  It takes in
   the final state and writes out, about a screen at a time,
   the results.  It currently skips over any 0 entries.  This
   procedure now also checks the normalization; note that if
   the operating matrices are not unitary, normalization will
   be off *)

PROCEDURE WriteResult(output : ARRAY OF State;
                      VAR repeat : BOOLEAN);

    VAR
        j, count, max, which : CARDINAL;
        check                : REAL;
        more, yesno          : CHAR;
        filename             : ARRAY[0..11] OF CHAR;
        filewrite            : BOOLEAN;

    BEGIN

    WriteString("Would you like to dump the outputs to a text file?
(y/n):");
    WriteLn;
    ReadChar(yesno); SkipLine;

    WriteString("Checking final normalization . . . ");
    check := NormalizeCheck(output);
    WriteFloat(check, 4, 6); WriteLn;

    IF ((yesno = 'y') OR (yesno = 'Y')) THEN
        filewrite := TRUE;
        WriteString("Please enter the name of the target file:");
        WriteLn;
        ReadString(filename); SkipLine;
        OpenWrite(filecid, filename, old, fileresult);
        IF fileresult <> opened THEN
            filecid := outstdcid;
            WriteString("Something went wrong . . . reverting to
terminal output.");
            WriteLn;
            filewrite := FALSE;
        END;
        SetOutChan(filecid);
    ELSE filewrite := FALSE;
    END;

    max := HIGH(output);
    count := 0;
    WriteString("Q Output"); WriteLn;
    FOR j := 0 TO max DO
        IF (NOT filewrite AND (count = 23)) THEN
            WriteLn; WriteString("Hit a key for more."); WriteLn;
            ReadChar(more); SkipLine;
            count := 0;
        END;
        IF (output[j].Coefficient = zero)
        THEN (* do nothing *)
        ELSE
            which := CAST(CARDINAL, output[j].Bits);
            WriteString("State :  ");
            WriteCard(which, 6);
            WriteString("  Coefficient :  ");
            WriteFixed(RE(output[j].Coefficient), 4, 6);
            WriteString(" + ");
            WriteFixed(IM(output[j].Coefficient), 4, 6);
            WriteString("i");
```

```
            WriteLn;
            INC(count);
        END;
    END;

    IF filewrite THEN
        SetOutChan(outstdcid);
        Close(filecid);
    END;

    WriteString("Would you like to run the program again? (y/n)");
    WriteLn;
    ReadChar(yesno); SkipLine;
    IF ((yesno = 'y') OR (yesno = 'Y'))
    THEN repeat := TRUE
    ELSE repeat := FALSE
    END;

    END WriteResult;


(* This procedure is what makes this program unique; this is where
   the state information pertinent to a specific gate is extracted;
   as inputs it takes an individual state and information about
   which lines are being affected; it then extracts the pertinent
   information into an extract state *)

PROCEDURE Extract(instate : State; gate : GateSpec;
                  VAR result : ExtractState);
    BEGIN

    WITH result DO
        Coefficient := instate.Coefficient;
        FirstBit := gate.FirstBit;
        SecondBit := gate.SecondBit;
        Bits := Bitset{};
        IF (gate.FirstBit IN instate.Bits)
        THEN INCL(Bits, 0)
        ELSE EXCL(Bits, 0)
        END;
        IF (gate.SecondBit IN instate.Bits)
        THEN INCL(Bits, 1)
        ELSE EXCL(Bits, 1)
        END;
    END; (* WITH Result *)

    END Extract;

(* This procedure undoes what the Extract procedure does; it takes
   an array of input states, assumed to be four, and puts them back
   onto the final state array *)

PROCEDURE Restore(VAR outstates : ARRAY OF ExtractState;
                  oldcopy : State; VAR target : ARRAY OF State);

    VAR
        count : CARDINAL;
        bitstate : Bitset;
        whichstate : CARDINAL;

    BEGIN

    FOR count := 0 TO 3 DO
        bitstate := oldcopy.Bits;
        IF (0 IN outstates[count].Bits)
```

```
        THEN INCL(bitstate, outstates[count].FirstBit)
        ELSE EXCL(bitstate, outstates[count].FirstBit)
        END;
        IF (1 IN outstates[count].Bits)
        THEN INCL(bitstate, outstates[count].SecondBit)
        ELSE EXCL(bitstate, outstates[count].SecondBit)
        END;
        whichstate := CAST(CARDINAL, bitstate);
        target[whichstate].Coefficient :=
            target[whichstate].Coefficient
            + outstates[count].Coefficient;
    END; (* FOR Count *)

    END Restore;

(* This is the critical procedure.  It takes the extracted states
   and applies the gate to them, resulting in a set of states
   which are then sent to the Restore procedure.  Note that it is
   assumed that "outstates" has exacty four states; this is
   critical to the operation of the procedure *)

PROCEDURE Process(instate : ExtractState; gate : GateSpec;
                  VAR outstates : ARRAY OF ExtractState);

    VAR
        thisgate : GateArray;
        which, j : CARDINAL;

    BEGIN

    thisgate := possiblegates[gate.ID].Matrix;
    which := CAST(CARDINAL, instate.Bits);
    FOR j := 0 TO 3 DO
        outstates[j] := instate;
        outstates[j].Bits := CAST(Bitset, j);
        outstates[j].Coefficient := (thisgate[j][which] *
                                     instate.Coefficient);
    END; (* FOR *)

    END Process;

(* This procedure cleans out a set of states, resetting them to 0;
   it also sets the bitsets correctly *)

PROCEDURE Reset(VAR target : ARRAY OF State);

    VAR
        j : CARDINAL;

    BEGIN

    FOR j := 0 TO HIGH(target) DO
        target[j].Bits := CAST(Bitset, j);
        target[j].Coefficient := zero;
    END;

    END Reset;

(* This procedure normalizes the state vector; it is internal,
   being used only by the Initialize procedure *)

PROCEDURE Normalize(VAR states : ARRAY OF State);

    VAR
        runningsum, divisor, absolute   : REAL;
```

```
        complexdivisor, complexabsolute : COMPLEX;
        i                               : CARDINAL;

    BEGIN
        runningsum := 0.0;
        FOR i := 0 TO HIGH(states) DO
            complexabsolute := states[i].Coefficient *
                               conj(states[i].Coefficient);
            absolute := RE(complexabsolute);
            runningsum := runningsum + absolute;
            divisor := sqrt(runningsum);
        END;
        FOR i := 0 TO HIGH(states) DO
            complexdivisor := CMPLX(divisor, 0.0);
            states[i].Coefficient := states[i].Coefficient
                                     / complexdivisor;
        END;
    END Normalize;

(* This another internal procedure, used by both Initialize and
   WriteResult for checking normalization *)

PROCEDURE NormalizeCheck(states : ARRAY OF State) : REAL;

    VAR
        runningsum, divisor, absolute   : REAL;
        complexabsolute                 : COMPLEX;
        i                               : CARDINAL;

    BEGIN
        runningsum := 0.0;
        FOR i := 0 TO HIGH(states) DO
            complexabsolute := states[i].Coefficient
                               * conj(states[i].Coefficient);
            absolute := RE(complexabsolute);
            runningsum := runningsum + absolute;
            divisor := sqrt(runningsum);
        END;
        RETURN 1.0 - divisor;
    END NormalizeCheck;

(* This initialization code sets up the set of available gates;
   gates are read in from "gatespec.txt", which follows the
   following format:

   (number of gates)

   (name of gate)
   REAL +i REAL  REAL +i REAL  REAL +i REAL  REAL +i REAL
   REAL +i REAL  REAL +i REAL  REAL +i REAL  REAL +i REAL
   REAL +i REAL  REAL +i REAL  REAL +i REAL  REAL +i REAL
   REAL +i REAL  REAL +i REAL  REAL +i REAL  REAL +i REAL

   . . . [other gates]

       where "REAL" is a real number in standard format, and
   the items in parentheses are strings; no checking on
   the matrix is performed, so the gates must be verified as
   unitary to achieve correct results *)

TYPE GateSpecifications = RECORD
                              GateName : ARRAY[0..15] OF CHAR;
                              Matrix : GateArray;
                          END;
```

```
VAR
    possiblegates           : ARRAY[0..15] OF GateSpecifications;
    instdcid, outstdcid,
    filecid, dumpcid        : ChanId;
    fileresult              : OpenResults;
    j, k, l, numberofgates  : CARDINAL;
    real, imag              : REAL;
    dummy                   : ARRAY [0..7] OF CHAR;

BEGIN

OpenRead(filecid, "gatespec.txt", old, fileresult);
IF (fileresult = ChanConsts.noSuchFile) THEN
    WriteString("Cannot find gatespec.txt!  Aborting program . . . ");
    HALT;
END;

WriteString("Quantum Gate Processing"); WriteLn;
WriteString("  by Z. Johnson"); WriteLn; WriteLn;
WriteString("Reading in gates . . . "); WriteLn;

instdcid := StdInChan();
outstdcid := StdOutChan();
SetInChan(filecid);
ReadCard(numberofgates); SkipLine; SkipLine;

FOR j := 0 TO (numberofgates - 1) DO
    ReadString(possiblegates[j].GateName); SkipLine;
    FOR k := 0 TO 3 DO
        FOR l := 0 TO 3 DO
            ReadReal(real);
            ReadToken(dummy);
            ReadReal(imag);
            possiblegates[j].Matrix[k][l] := CMPLX(real, imag);
        END;
        SkipLine;
    END;
    SkipLine;
END;

SetInChan(instdcid);
Close(filecid);

END QuantumGateLibrary.
```

The following is a sample of *gatespec.txt*; the file starts at the "6".

```
6

Identity
1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0

ControlledNot
1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0

OneBitFlip
0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0

U_A
0.70710678 +i 0.0  0.70710678 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.70710678 +i 0.0  -0.70710678 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.70710678 +i 0.0  0.70710678 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.70710678 +i 0.0  -0.70710678 +i 0.0

U_B(pi/4)
1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  0.70710678 +i 0.70710678

A(0,0,pi/4)
1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  1.0 +i 0.0  0.0 +i 0.0  0.0 +i 0.0
0.0 +i 0.0  0.0 +i 0.0  0.70710678 +i 0.0  0.0 +i -0.70710678
0.0 +i 0.0  0.0 +i 0.0  0.0 +i -0.70710678  0.70710678 +i 0.0
```

Appendix B: Sample Program Session

The following are two sample sessions of Q. In the first run the following computation is performed:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\pi/4} \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 0 \\ 0 \\ 1 \\ e^{i\pi/4} \end{pmatrix} \tag{25}$$

Note that in this session the initial state is not entered normalized; rather, since the initial state is an even superposition of $|10\rangle$ and $|11\rangle$, both states are entered as 1.0 and the program automatically normalizes them. The error initially reported is from rounding error in the normalization computation. Rounding error is similarly responsible for the reported lack of error in the final normalization check. The gate used is one that is used by Barenco [13]; it performs a phase shift if both qubits are in the $|1\rangle$ state. Note also that, for the sake of brevity, the program does not report output states with a zero coefficient.

```
Quantum Gate Processing
  by Z. Johnson

Reading in gates . . .

Please enter the number of bits to work on:  2

Would you like to read in the settings from a text file?
(The settings will be read in from settings.txt) (y/n):
n
Please enter the number of gates.
1

Please enter the gate as follows:
   (0)   Identity
   (1)   ControlledNot
   (2)   OneBitFlip
   (3)   U_A
   (4)   U_B(pi/4)
   (5)   A(0,0,pi/4)


Please enter the type for gate 0:
```

```
4
Please enter the first bit of the gate:
0
Please enter the second bit of the gate:
1
The registers will initially be set to 0
unless you specify otherwise.
Would you like to specify any values? (y/n):y
Which state would you like to specify?
(Please enter the state in base 10):
2
What would you like for its real coefficient?
1.0
What would you like for its imaginary coefficient?
0.0
Would you like to specify another state? (y/n)
y
Which state would you like to specify?
(Please enter the state in base 10):
3
What would you like for its real coefficient?
1.0
What would you like for its imaginary coefficient?
0.0
Would you like to specify another state? (y/n)
n
Normalizing . . .
Checking initial normalization . . . 5.960E-08
Executing, please wait.
Elapsed time is      0 in milliseconds.
Would you like to dump the outputs to a text file? (y/n):
n
Checking final normalization . . . 0.000E+00
Q Output
State :       2  Coefficient :  0.7071 + 0.0000i
State :       3  Coefficient :  0.5000 + 0.5000i
Would you like to run the program again? (y/n)
n
```