Technical University of Crete
Department of Electronic and Computer Engineering

Techniques for Simulating Quantum Computers

by
Koufogiannakis Christos

This thesis is submitted in partial fulfillment of requirements for the
Diploma Degree in Electronic and Computer Engineering

June 2004

Abstract

The first algorithm of practical interest that takes advantage of quantum mechanics was proposed by Peter Shor in 1994. Shor described a polynomial time quantum algorithm for factoring integers. Factoring is considered to be a hard problem for classical computers. Indeed, the efficiency of the famous public-key cryptosystem *RSA* is based on the assumption that classical computers cannot factor big integers fast. On the other hand a quantum computer would be able to break the *RSA* system relatively easy. A number of quantum algorithms have been proposed since 1994, like Grover's algorithm for database search, quantum cryptography and quantum teleportation.

Unfortunately, current technology makes it impossible to build large scale quantum computers to run these algorithms. This is a great roadblock for those who want to test their algorithms or those who want to write new quantum algorithms. Therefore, the only way to test quantum algorithms is by simulating them on classical computers. However, the exponential state explosion makes quantum simulation a difficult task for classical computers. In this thesis we describe how *Binary/Algebraic Decision Diagrams* (*BDDs*, *ADDs*) can be used to simulate quantum circuits, and especially Shor's algorithm.

Acknowledgements

I would like to thank,

my supervisor, Prof. Vasilis Samoladas, for introducing me to the subject of quantum computing and providing me with many constructive conversations.

my professors and my friends at the Technical University of Crete for their assistance and their comments during my studies.

Contents

Chapter 1

Introduction to Quantum Computing

Richard Feynman [1] observed in 1982 that it seems to be extremely difficult for a classical computer to simulate efficiently how a quantum system evolves in time. He also noted that, if we had a computing device that uses quantum effects, then this simulation could be made efficiently. Thus, he indirectly suggested that a *quantum computer* may be more efficient than any classical one.

It wasn't until 1994, when Peter Shor [2] described a polynomial time quantum algorithm for factoring integers, that Feynman's suggestion became stronger than ever. Two years later, Lov Grover [3] developed a technique for searching an unstructured list of items, which gives a polynomial speedup over classical computers. These two algorithms are the most remarkable in the pile of quantum algorithms, which has been developed during the last decade.

This chapter targets to introduce the reader in the fundamental principles of quantum computation. We do not deal with the details of how a quantum computer is physically constructed. The reader is supposed to have just a basic knowledge of linear algebra and boolean logic.

1.1 The Bra/Ket Notation

As we will see shortly, quantum states are represented by vectors; therefore, we need a compact notation for state vectors. Dirac [4] introduced the so called Bra/Ket notation where the ket $|x\rangle$ is used to describe the column vectors and the bra $\langle x|$ denotes the complex conjugate transpose of $|x\rangle$. The most commonly used vectors are

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{1.1}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{1.2}$$

The *inner* product of vectors $|x\rangle$ and $|y\rangle$ is represented by

$$\langle x\|y\rangle \text{ or } \langle x|y\rangle \tag{1.3}$$

The *inner* product of two vectors is a complex number. Two nonzero vectors are orthogonal if and only if the result of their inner product is 0. For example vectors $|0\rangle$ and $|1\rangle$ are orthogonal

$$\langle 0|1\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1\cdot 0 + 0\cdot 1 = 0 \tag{1.4}$$

The *outer* product of vectors $|x\rangle$ and $|y\rangle$ is represented by

$$|x\rangle\langle y| \tag{1.5}$$

The *outer* product of two *n*-element vectors is an *n* by *n* matrix, i.e.

$$|0\rangle\langle 1| = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \tag{1.6}$$

1.2 Tensor Product

Now, we describe the tensor product ($\otimes$), a matrix operation which is primitive to describe quantum systems mathematically. Suppose we have an *m* by *n* matrix *A* and a *k* by *p* matrix *B*. Then, the tensor product, also called the Kronecker product, of *A* and *B* is defined as

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \dots & a_{1n}B \\ a_{21}B & a_{22}B & \dots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \dots & a_{mn}B \end{bmatrix} \tag{1.7}$$

We see that the result is an *mk* by *np* matrix. We can give an example for (1.7)

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \otimes \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} & 2\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \\ 3\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} & 4\begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1\cdot 5 & 1\cdot 6 & 2\cdot 5 & 2\cdot 6 \\ 1\cdot 7 & 1\cdot 8 & 2\cdot 7 & 2\cdot 8 \\ 3\cdot 5 & 3\cdot 6 & 4\cdot 5 & 4\cdot 6 \\ 3\cdot 7 & 3\cdot 8 & 4\cdot 7 & 4\cdot 8 \end{bmatrix} = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}$$

If $n = 1$ and $p = 1$, then A and B are vectors, therefore, the tensor product is defined for vectors as well. We can give a simple example

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \otimes \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1\cdot\begin{bmatrix} 3 \\ 4 \end{bmatrix} \\ 2\cdot\begin{bmatrix} 3 \\ 4 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1\cdot 3 \\ 1\cdot 4 \\ 2\cdot 3 \\ 2\cdot 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \\ 6 \\ 8 \end{bmatrix}$$

If we use the Bra/Ket notation for vectors, then the symbol $\otimes$ can be omitted

$$|0\rangle|0\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

For the tensor product, the following hold

$$(A \otimes B)(C \otimes D) = AC \otimes BD \tag{1.8}$$
$$(A + B) \otimes C = A \otimes C + B \otimes C \tag{1.9}$$
$$A \otimes (B + C) = A \otimes B + A \otimes C \tag{1.10}$$
$$aA \otimes bB = ab(A \otimes B) \tag{1.11}$$

Notice that the tensor product does not commute i.e.

$$|0\rangle|1\rangle \neq |1\rangle|0\rangle .$$

1.3 Quantum Bit

The bit is the fundamental component of classical computation and classical information. We can describe the properties of a bit either by adopting a mathematical point of view or by describing it as a real system. The former perspective indicates that a bit has a state - either 0 or 1-, while the latter treats the bit as a voltage, where 0 Volts may correspond to 0 and 5 Volts may correspond to 1, supposed we are talking for TTL logic. The description of bits as mathematical objects gives as the freedom to build a general theory of computation and information which is independent of any specific physical realization. Thus, at the rest of this paper we are not going to deal with the way a classical or a quantum computer is physically implemented.

The quantum analogue of the classical bit is called *quantum bit*, or *qubit* for short. Just as a classical bit has a state, a qubit has also a state. Two possible states for a qubit are the states $|0\rangle$ and $|1\rangle$, which correspond to the states 0 and 1 for a classical bit. The difference between bits and qubits is that a qubit can be in a state which is a linear combination of $|0\rangle$ and $|1\rangle$, often called superposition

$$|\psi\rangle = a|0\rangle + \beta\,|1\rangle \tag{1.12}$$

We can rewrite (1.12) using (1.1) and (1.2)

$$|\psi\rangle = \begin{bmatrix} a \\ \beta \end{bmatrix} \tag{1.13}$$

The numbers a and β are called *amplitudes* and are complex numbers such that

$$\langle\psi|\psi\rangle = |a|^2 + |\beta|^2 = 1 \tag{1.14}$$

We know that we can examine a bit at any time to determine whether it is in the state 0 or 1. Remarkably, we cannot examine a qubit to determine its quantum state, that is, the values of a and β. Instead, when we examine a qubit we get either the result 0 with probability $|a|^2$, or the result 1 with probability $|\beta|^2$. This explains why (1.14) holds, since probabilities must sum to one. The action of examining a qubit is called *measurement* and after it, the qubit collapses to either the state $|0\rangle$ or $|1\rangle$ depending on the result of the measurement. For example, suppose that we have a qubit in the state

$$\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

Measuring this state will give the result 0 with probability $\left|\frac{1}{\sqrt{2}}\right|^2 = 50\%$ or the result 1 with equal probability. Now let's say that the outcome of the measurement was 1, then, the qubit collapses to the state $|1\rangle$.

The quantum bit can be defined more formally as a *two-dimensional Hilbert space H_2*. The space H_2 is equipped with a fixed basis $B = \{|0\rangle,\ |1\rangle\}$, a so-called *computational basis*. States $|0\rangle$ and $|1\rangle$ are called *basis states*. A general state of a single quantum bit is a vector $\begin{bmatrix} a \\ \beta \end{bmatrix}$ having unit length, i.e. $|a|^2 + |\beta|^2 = 1$.

1.4 Multiple Qubits

Suppose we have two classical bits, then there would be four possible states, 00, 01, 10 and 11. A system of two quantum bits is a four dimensional Hilbert space $H_4 = H_2 \otimes H_2$ with computational basis $B = \{|0\rangle|0\rangle,\ |0\rangle|1\rangle,\ |1\rangle|0\rangle,\ |1\rangle|1\rangle\}$. We can write $|0\rangle|0\rangle = |0,0\rangle = |00\rangle$, $|0\rangle|1\rangle = |0,1\rangle = |01\rangle$, etc. The state vector describing the two qubits is

$$|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle \tag{1.15}$$

Similar to the case for one qubit, the measurement result x (= 00, 01, 10, 11) occurs with probability $|a_x|^2$, with the state of the qubits after the measurement being $|x\rangle$. The condition that probabilities sum to one is expressed by the *normalization* condition

$$|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1 \tag{1.16}$$

For a multiple qubit system, we could measure just a subset of the qubits. Let's say that we want to measure the first qubit of the two qubit system described by (1.15). Then the probability to measure 0 is $|a_{00}|^2 + |a_{01}|^2$, leaving the post-measurement state

$$|\psi\rangle = \frac{a_{00}|00\rangle + a_{01}|01\rangle}{\sqrt{|a_{00}|^2 + |a_{01}|^2}} \tag{1.17}$$

The probability to measure 1 is $|a_{10}|^2 + |a_{11}|^2$ and the post-measurement state is

$$|\psi\rangle = \frac{a_{10}|10\rangle + a_{11}|11\rangle}{\sqrt{|a_{10}|^2 + |a_{11}|^2}} \tag{1.18}$$

The denominators in both (1.17) and (1.18) are used so that the *normalization* condition still holds for the post-measurement states.

Finally, we can make the notation even simpler if we write $|00\rangle = |0\rangle$, $|01\rangle = |1\rangle$, $|10\rangle = |2\rangle$, $|11\rangle = |3\rangle$. We can now rewrite (1.15)

$$|\psi\rangle = a_0|0\rangle + a_1|1\rangle + a_2|2\rangle + a_3|3\rangle = \sum_{x=0}^{3} a_x|x\rangle \tag{1.19}$$

Using this notation, an *n*-qubit system is a Hilbert space $H_{2^n} = \underbrace{H_2 \otimes \cdots \otimes H_2}_{n} = H_2^{\otimes n}$ having computational basis $B = \{|0\rangle, |1\rangle, ..., |2^n - 1\rangle\}$. The state of an *n*-qubit system is the unit-length vector

$$|\psi\rangle = \sum_{x=0}^{2^n-1} a_x |x\rangle \tag{1.20}$$

so it is required that

$$\sum_{x=0}^{2^n-1} |a_x|^2 = 1 \tag{1.21}$$

A group of qubits is called quantum register, just like a group of classical bits.

1.5 Entanglement

Suppose we have a quantum system in state $|\psi\rangle$ and another quantum system in state $|\phi\rangle$. Then the state of the combined system is given by the tensor product $|\psi\rangle \otimes |\phi\rangle$. For example if we have two qubits, each in the state $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$, then the state of the combined two qubit system is

$$\left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) = \frac{1}{\sqrt{2}}|0\rangle \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) + \frac{1}{\sqrt{2}}|1\rangle \otimes \left(\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle\right) =$$

$$= 1/2\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$$

We say that a system in the state $\frac{1}{2}\left(|00\rangle + |01\rangle + |10\rangle + |11\rangle\right)$ is decomposable, since it can be expressed as the tensor product of its subsystems. However, there exist quantum systems, which are in a state that cannot be written as the tensor product of their subsystems. For example the two qubit state $\frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right)$ is not decomposable. To see this, assume on the contrary, that

$$\frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right) = \left(a_0|0\rangle + a_1|1\rangle\right) \otimes \left(b_0|0\rangle + b_1|1\rangle\right) = a_0 b_0|00\rangle + a_0 b_1|01\rangle + a_1 b_0|10\rangle + a_1 b_1|11\rangle$$

for some complex numbers a_0, a_1, b_0, b_1. But then

$$\begin{cases} a_0 b_0 = 1/\sqrt{2} \\ a_1 b_1 = 1/\sqrt{2} \\ a_1 b_0 = 0 \\ a_0 b_1 = 0 \end{cases}$$

which is impossible.

Such states are called *entangled* and are responsible for many surprises in quantum computation and quantum information. In fact these states make quantum systems differ from classical ones. To see a characteristic property of entangled states, let's look to what happens if we measure the first qubit of the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, which is known as Bell state. There are two possible results: 0 with probability ½, leaving the post-measurement state $|\varphi\rangle = |00\rangle$, and 1 with probability ½, leaving the post-measurement state $|\varphi\rangle = |11\rangle$. Measuring now the second qubit will always give the same result as the measurement of the first qubit. It was John Bell, who first noticed that *measurement correlations* in a Bell state are stronger than those in a classical system.

1.6 Single Qubit Gates

We have seen how a quantum state is described, but we have not yet mentioned how we construct a desirable state. First, we must accept that a quantum computer is capable of initializing qubits to either the state $|0\rangle$ or $|1\rangle$. The simpler operation on a classical bit is the *NOT* gate which flips the value of the bit. We can define an analogous operation for a quantum bit. A perfect candidate is an operation which takes the state $|0\rangle$ to $|1\rangle$ and vice versa. Suppose we define a two by two matrix X

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.22}$$

Now let's multiply X with $|0\rangle$ and $|1\rangle$ separately

$$X|0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle \tag{1.23}$$

$$X|1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = |0\rangle \tag{1.24}$$

From equations (1.23) and (1.24) we see that matrix X represents the operation we want, and thus it is known as the *quantum NOT* gate. In fact this operation acts linearly, that is, it takes the state $a|0\rangle + b|1\rangle$ to the state $a|1\rangle + b|0\rangle$

$$X\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} b \\ a \end{bmatrix} \tag{1.25}$$

We can symbolize the above transformation as

$$a|0\rangle + b|1\rangle \xrightarrow{X} a|1\rangle + b|0\rangle \tag{1.26}$$

It follows that since the state of a single qubit is represented by a two row vector, the operators must be represented by 2x2 matrices. Are there any other constraints on what matrices can be used as quantum gates? We should consider that in the resulting state, the *normalization* condition $|a|^2 + |\beta|^2 = 1$ should still hold. It turns out that this property is satisfied when the matrix U describing a single qubit quantum gate is *unitary,* that is

$$U^\dagger U = UU^\dagger = I \tag{1.27}$$

where $U^\dagger$ is the adjoint of U (obtained by transposing and then complex conjugating U) and I is the two by two identity matrix. We can easily confirm that X is unitary

$$X^\dagger X = XX^\dagger = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

This *unitarity* constraint is the only constraint on quantum gates. The fact that each operator U is unitary means that we can construct another operator $U^\dagger$ which performs the reverse operation. To make clear, suppose that we apply a quantum gate U to a state $|\psi\rangle$ taking the state $|\phi\rangle$. We can go back to state $|\psi\rangle$ if we apply $U^\dagger$ to $|\phi\rangle$

$$|\psi\rangle \xrightarrow{U} \underbrace{U|\psi\rangle}_{|\phi\rangle} \xrightarrow{U^\dagger} U^\dagger U|\psi\rangle = I|\psi\rangle = |\psi\rangle \tag{1.28}$$

Another very important quantum gate is the *Hadamard* gate

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{1.29}$$

which is unitary, since $H = H^\dagger$ and $H^2 = I$. This gate creates a state of equal superposition when it acts on the *basis* states

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \tag{1.30}$$

$$H|1\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle \tag{1.31}$$

The resulting states are said to be in an equal superposition because the probability to measure 0 is the same as the probability to measure 1. A brief description of some important quantum gates follows

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad a|0\rangle + b|1\rangle \xrightarrow{I} a|0\rangle + b|1\rangle \tag{1.32}$$

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad a|0\rangle + b|1\rangle \xrightarrow{Z} a|0\rangle - b|1\rangle \tag{1.33}$$

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad a|0\rangle + b|1\rangle \xrightarrow{Y} -bi|0\rangle + ai|1\rangle \tag{1.34}$$

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \qquad a|0\rangle + b|1\rangle \xrightarrow{S} a|0\rangle + bi|1\rangle \tag{1.35}$$

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix} \qquad a|0\rangle + b|1\rangle \xrightarrow{T} a|0\rangle + be^{i\pi/4}|1\rangle \tag{1.36}$$

We can use the outer product to describe quantum gates. For example, $|0\rangle\langle 1|$ is the transformation that maps $|1\rangle$ to $|0\rangle$ and $|0\rangle$ to $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Similarly, $|1\rangle\langle 0|$ maps $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$. For example

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{1.37}$$

$$S = |0\rangle\langle 0| + i \cdot |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \tag{1.38}$$

1.7 Controlled Gates

Suppose we have a two qubit system. We want to apply a quantum gate to the second qubit, depending on the value of the first qubit. For example, we want to apply the quantum *NOT* gate on the second qubit only if the first qubit is in the state $|1\rangle$. In fact, such a gate does exist and is called *controlled-Not* or *CNOT* for short. *CNOT* is described by the matrix

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.39}$$

It is easy to verify that CNOT performs the transformation

$$|00\rangle \to |00\rangle,\ |01\rangle \to |01\rangle,\ |10\rangle \to |11\rangle,\ |11\rangle \to |10\rangle \tag{1.40}$$

We see that the first qubit remains unchanged, while, the second qubit is flipped only when the first qubit is in the state $|1\rangle$. The above action can be summarized as

$$|A,B\rangle \to |A,B\oplus A\rangle \tag{1.41}$$

where $A,B \in \{0,1\}$ and $\oplus$ is addition modulo two, or else the *XOR* of *A* and *B*. The first qubit, $|A\rangle$, is called *control* qubit and the second qubit, $|B\rangle$, is called *target* qubit.

Generally, if we have a single qubit quantum gate *U* we can construct the *Controlled-U* gate, that is, a gate that transforms the second qubit according to *U* only when the first qubit is in the state $|1\rangle$. The *Controlled-U* gate is described by the matrix

$$CU = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix} \tag{1.42}$$

where *I* is the two by two identity matrix, *U* is the two by two gate matrix and "0" denotes the two by two matrix with all elements zero. Since *U* is unitary, *Controlled-U* is also unitary.

We can construct gates with more than one control qubit. We will mention only one multi-controlled quantum gate, a *controlled-NOT* gate with two control qubits. This gate is known as *Toffoli* gate and is described by the matrix

$$Toffoli = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{1.43}$$

When we apply the *Toffoli* gate to a three qubit quantum register, it flips the target qubit only if both control qubits are in the state $|1\rangle$

$$|A,B,C\rangle \to |A,B,C\oplus AB\rangle \tag{1.44}$$

The *Toffoli* gate can be used to simulate the classical *NAND* gate. Suppose that the input bits to the classical *NAND* gate have the values A and B, then if C is 1 in equation (1.44) then we take the *NAND* of A and B in the third qubit

$$|A,B,1\rangle \to |A,B,1\oplus AB\rangle = |A,B,\neg AB\rangle \tag{1.45}$$

The main difference of *Toffoli* from *NAND* is that, like every other quantum gate, *Toffoli* gate is reversible; therefore, we can "undo" the computation back. Recalling that the *NAND* gate is *universal,* we understand that the *Toffoli* gate can be used to simulate any classical gate. Intuitively, the ability of a quantum computer to simulate any classical gate means that they are at least equivalent to classical computers, in that they can perform the same computations.

1.8 No-cloning Theorem

We have said that any quantum gate must be *unitary*. In 1982, Wootters and Zurek [5] proved that this property implies that unknown quantum states cannot be copied or cloned. To prove the no cloning theorem we will assume that U is a unitary transformation that clones, that is

$$U|\psi,0\rangle = |\psi,\psi\rangle \tag{1.46}$$

for all quantum states $|\psi\rangle$. Consider $|c\rangle = a|0\rangle + b|1\rangle$, then by linearity

$$U|c,0\rangle = U\big(a|00\rangle + b|10\rangle\big) = aU|00\rangle + bU|10\rangle = a|00\rangle + b|11\rangle \tag{1.47}$$

But U is a cloning transformation, that means

$$U|c,0\rangle = |c,c\rangle = \big(a|0\rangle + b|1\rangle\big)\big(a|0\rangle + b|1\rangle\big) = a^2|00\rangle + ab|01\rangle + ab|10\rangle + b^2|11\rangle \tag{1.48}$$

Equations (1.47) and (1.48) are not equal, unless $ab = 0$. This means than we cannot copy an unknown state $a|0\rangle + b|1\rangle$, but we can copy the *basis* states $|0\rangle$ and $|1\rangle$. For example, the *CNOT* gate can copy the *basis* states, but it cannot copy an unknown state.

1.9 Measurement

We pointed that the evolution of a quantum system is described by unitary transformations. However, there must also be times, when the quantum system interacts with the classical world in a way that we can observe the state of the quantum system. Such an interaction makes the system no longer closed, which entails that it is not necessarily subject to unitary evolution. *Measuring* a quantum state is a *non-unitary* transformation indeed.

Quantum measurements are described by a collection $\{M_\text{m}\}$ of *measurement operators*. The index m refers to the outcome of the measurement that may occur. If the state of the quantum system is $|\psi\rangle$ before the measurement, then the probability that result *m* occurs is

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle \tag{1.49}$$

and the post-measurement state is

$$\frac{M_m|\psi\rangle}{\sqrt{p(m)}} = \frac{M_m|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}} \tag{1.50}$$

The constraint that probabilities must sum to one is translated into the *completeness equation*

$$\sum_m M_m^\dagger M_m = I \tag{1.51}$$

A simple but very important example is the measurement of a single qubit system which is in the state $|\psi\rangle = a|0\rangle + b|1\rangle$. We can define the *measurement operators* using the outer product

$$M_0 = |0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{1.52}$$

$$M_1 = |1\rangle\langle 1| = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \tag{1.53}$$

It is easy to see that M_0 and M_1 satisfy the *completeness equation*. The probability of obtaining the measurement result 0 is

$$p(0) = \langle\psi|M_0^\dagger M_0|\psi\rangle = \begin{bmatrix} a^* & b^* \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = |a|^2 \tag{1.54}$$

and the probability to measure 1 is

$$p(1) = \langle\psi|M_1^\dagger M_1|\psi\rangle = \begin{bmatrix} a^* & b^* \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = |b|^2 \tag{1.55}$$

The post-measurement states for these two cases are

$$\frac{M_0|\psi\rangle}{\sqrt{p(0)}} = \frac{1}{|a|} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{|a|} \begin{bmatrix} a \\ 0 \end{bmatrix} = \frac{a}{|a|}|0\rangle \tag{1.56}$$

$$\frac{M_1|\psi\rangle}{\sqrt{p(1)}} = \frac{1}{|b|} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \frac{1}{|b|} \begin{bmatrix} 0 \\ b \end{bmatrix} = \frac{b}{|b|}|1\rangle \tag{1.57}$$

It can be proved that multipliers like $a/|a|$ can be effectively ignored, so states (1.56) and (1.57) are equivalent to $|0\rangle$ and $|1\rangle$ respectively.

When measuring on the computational basis, the *measurement operators* for measuring the *n* qubits of an *n*-qubit system are

$$M_x = |x\rangle\langle x|, \quad x \in [0,\ 2^n - 1] \tag{1.58}$$

A more interesting situation is when we want to measure a subset of the qubits of a quantum register. Suppose we have a register that is the result of combining three smaller registers, and its state is

$$|\psi\rangle = \underbrace{|m\rangle}_{\substack{p \\ qubits}} \overbrace{|y\rangle}^{\substack{n \\ qubits}} \underbrace{|z\rangle}_{\substack{q \\ qubits}} \tag{1.59}$$

If we want to measure the *n* qubits of register $|y\rangle$, the *measurement operators* are

$$M_x' = I_{2^p} \otimes |x\rangle\langle x| \otimes I_{2^q} \tag{1.60}$$

where I_{2^k} is the identity matrix of dimension 2^k.

1.10 Quantum Circuits

People who design classical circuits use special symbols for the basic boolean gates. Moreover they use these symbols to design more complex circuits on the paper. The idea is adopted for quantum computers too. First, we have to create symbols for the basic gates, the single qubit gates described above, and then we will use them to build bigger circuits. A single qubit gate is denoted by a box which has the name of the gate. For example, if we apply the gate *U* to a qubit which has initial state $|\psi\rangle$, we can symbolize the process like

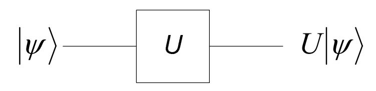

Figure 1.1: Applying the gate U to a qubit which has initial state $|\psi\rangle$

where the gate *U* is applied from left to right.

We are ready to give a symbol for the *controlled-U* gate which acts on a two qubit system which is in state $|\psi\rangle$

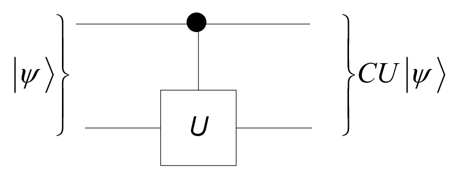

Figure 1.2: Applying the controlled-U gate to a two qubit state $|\psi\rangle$

The qubit with the black dot is the *control* qubit, while the other is the *target* qubit. Because of the importance of the *controlled-NOT* gate, it has a special symbol

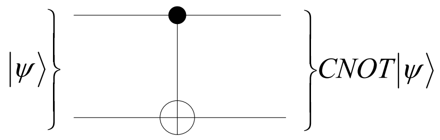

Figure 1.3: Controlled-NOT gate

We can introduce a new two qubit gate which swaps the states of the two bits,

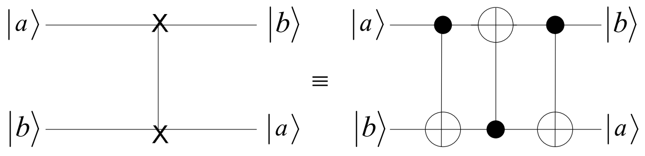

Figure 1.4: SWAP gate

The corresponding matrix for the *SWAP* gate is

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.61}$$

We need a way to represent the action of measuring a qubit in a circuit. This operation converts a single qubit into a probabilistic classical qubit *M*. The classical bit is distinguished from a qubit by drawing it as a double line wire,

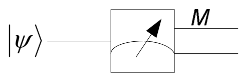

Figure 1.5: Measurement symbol

We can make the circuit representation more compact adopting the following equivalences

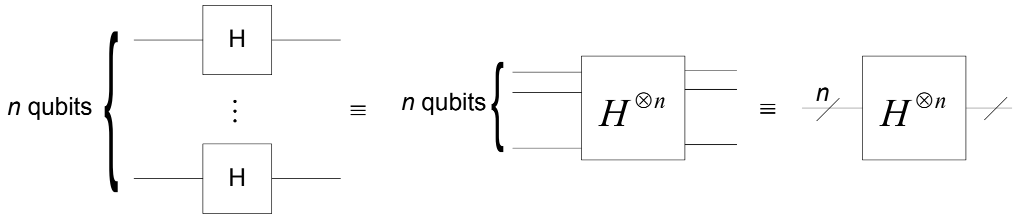

Figure 1.6: Shorter representation for multiple Hadamard gates

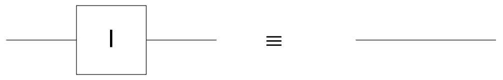

Figure 1.7: Shorter representation for Identity gate

In a circuit with gates acting on more than one qubit, the corresponding gate matrix results from the tensor product of the single qubit gates. For example, the matrix for the overall circuit of *Figure 1.6* is

$$H^{\otimes n} = \underbrace{H \otimes \cdots \otimes H}_{n} \tag{1.62}$$

We may have more than one gate applied on one qubit. In this case we put all the gates on the same "wire", with the one on the left acting first. The corresponding matrix is the product of all matrices that act on the qubit. Let see a small example,

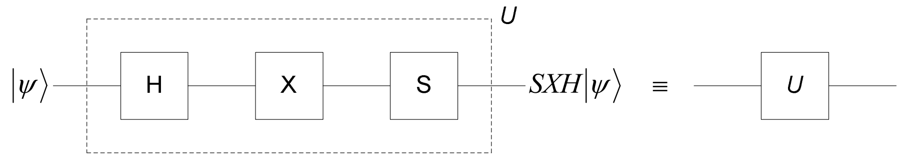

Figure 1.8: A series of gates acting on the same qubit

$$U = SXH \tag{1.63}$$

Our final example includes three qubits and a number of quantum gates, in an attempt to present all the cases which have been discussed previously. We give the circuit and then the corresponding matrix,

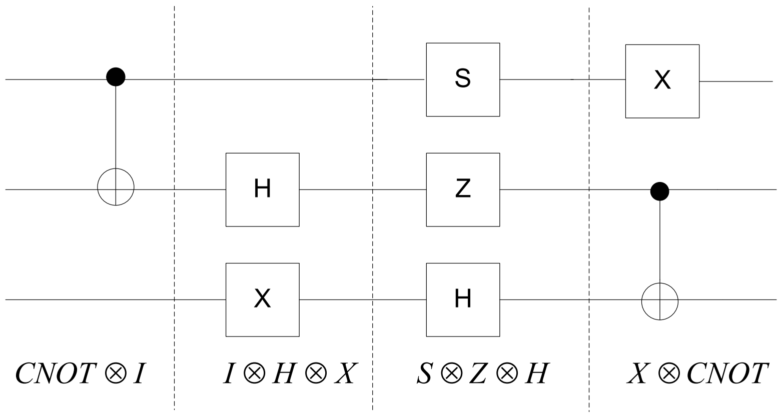

Figure 1.9: A three qubit circuit

$$U = (X \otimes CNOT)(S \otimes Z \otimes H)(I \otimes H \otimes X)(CNOT \otimes I) \tag{1.64}$$

In (1.64) U is an 8x8 matrix. Generally speaking, a gate for n qubits has dimension 2^n by 2^n.

There are some features of classical circuits that are not allowed in quantum circuits. First, there is no feedback form one part of the circuit to another; quantum circuits are *acyclic.* Second, the operation of classical circuits known as *FANIN* is not allowed for quantum circuits. Third, the inverse operation, *FANOUT*, is also not allowed, for, quantum mechanics forbid the copying of a qubit.

1.11 Reversibility and Scratch space

We can construct a quantum circuit to compute the function *f*(*x*),

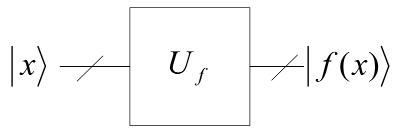

Figure 1.10: A possible transformation to compute f(*x*)

Unfortunately, this approach doesn't work for all functions. The constraint that quantum transformations are reversible makes the circuit of *Figure 1.10* valid only when *f*(*x*) is one to one, that is, function $f^{-1}(x)$ exists. In simpler words, we must be able to find the original input $|x\rangle$, by "uncomputing" the result $|f(x)\rangle$. Of course, only a few functions are one to one, so we must find a way to evaluate all the functions reversibly. The idea is to have two distinct registers. One should store the input and the other should store the output. Indeed, this is the way to compute functions using quantum circuits,

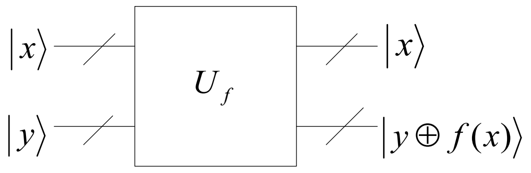

Figure 1.11: Computing f(*x*) *reversibly*

If we initialize $|y\rangle$ in the state $|0\rangle$, then $|y \oplus f(x)\rangle = |0 \oplus f(x)\rangle = |f(x)\rangle$, which means that the second register holds the result *f*(*x*). This transformation is summarized as

$$U_f|x,0\rangle = |x, f(x)\rangle \tag{1.65}$$

It is common for transformations U_f to use some temporary qubits as scratch space for their calculations. These temporary qubits, also known as *ancilla qubits*, are initialized to $|0\rangle$, but may end in any state $|g(x)\rangle$ (*g* for garbage) after U_f is applied. In the case that U_f uses *ancilla qubits*, *Figure 1.11* can be changed to *Figure 1.12*

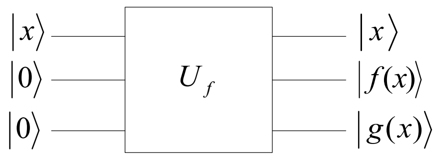

Figure 1.12: Computing f(*x*) *using scratch space*

In a chain of transformations that all use scratch space, we will end with a big number of qubits that store garbage. Fortunately, C. Bennett ([6, 7]) proposed two tricks to "erase" these qubits, in order to reuse them later. The first trick requires one more register,

$$|x,0,0,0\rangle \xrightarrow{U_f} |x, f(x), g(x), 0\rangle \xrightarrow{CNOT\,2,4} |x, f(x), g(x), f(x)\rangle \xrightarrow{U_f^\dagger} |x,0,0, f(x)\rangle \tag{1.66}$$

The second step is a form of copying, by applying bitwise *controlled-NOT* gates between the qubits of the second and the fourth register. The last step is the application of the inverse transformation of U_f.

The second trick introduces a way to write the output of the function $f(x)$ in the input register, provided that it is an invertible function. Then there exists the transformation $U_{f^{-1}}$,

$$|x,0\rangle \xrightarrow{U_{f^{-1}}} |x, f^{-1}(x)\rangle \tag{1.67}$$

which can be used in the following way

$$|x,0\rangle \xrightarrow{U_f} |x, f(x)\rangle \xrightarrow{Swap} |f(x), x\rangle \xrightarrow{U^{\dagger}_{f^{-1}}} |f(x),0\rangle \tag{1.68}$$

Thus, we can save even more space in the case $f(x)$ is invertible.

1.12 Quantum Parallelism

Suppose we have a transformation U_f that computes the function $f(x)$. This transformation is linear, that is, if the input is in a superposition then it is applied to all *basis states*. In this way, quantum computers can evaluate the function $f(x)$ for many different values of x simultaneously. This unique feature of quantum computers is called *quantum parallelism* and together with *entanglement* is exploited by almost all quantum algorithms. Only two steps are required to take the advantage of *quantum parallelism*. Starting with all the input qubits in the state $|0\rangle$, we apply the *Hadamard* gate to each of the n input qubits, to create an equal superposition in the input register

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \tag{1.69}$$

Then we apply U_f to get a superposition of all 2^n possible results of $f(x)$

$$U_f\left(\left(\frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle\right) \otimes |0\rangle\right) = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x, f(x)\rangle \tag{1.70}$$

Figure 1.13 depicts the circuit that performs these two steps

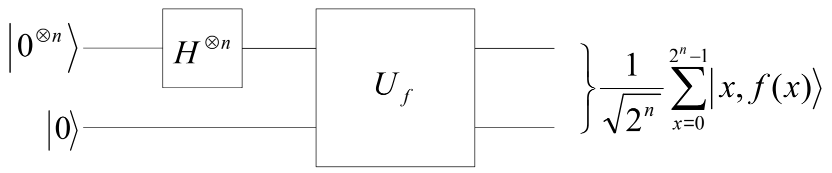

Figure 1.13: Computing f(x) for all possible inputs

We will use the *Toffoli* gate to simulate the classical *AND* gate as a trivial example for *quantum parallelism*. *Figure 1.14* shows the quantum circuit for the *AND* function.

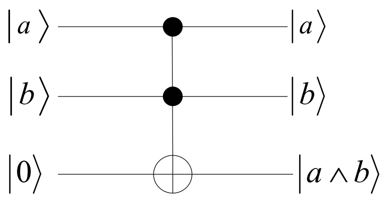

Figure 1.14: Simulating the classical AND gate using a Toffoli gate

First, we create an equal superposition in the two input qubits

$$(H\otimes H\otimes I)|0\rangle|0\rangle|0\rangle=\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes|0\rangle=\frac{1}{2}(|000\rangle+|010\rangle+|100\rangle+|110\rangle) \quad (1.71)$$

We are going to apply the *Toffoli* gate to the state (1.71),

$$\mathit{Toffoli}\,\frac{1}{2}(|000\rangle+|010\rangle+|100\rangle+|110\rangle)=\frac{1}{2}(|000\rangle+|010\rangle+|100\rangle+|111\rangle) \quad (1.72)$$

The resulting superposition contains the result of the *AND* gate for all possible inputs. In other words, it can be view as the truth table for the conjugation. In the final state, the input registers are entangled with the output register. Measuring the third qubit, would project the input qubits in the states for which the function produces a result equal to the result of measurement. For example, the measurement outcome is 1, then the post-measurement state would be $|111\rangle$.

To sum up, *quantum parallelism* gives us the opportunity to compute all the possible values of a function *f*(*x*) by evaluating the function only once. This is an amazing feature indeed, but how can we take advantage of the final state which is in a superposition of all possible results? Measuring the final state gives only one result, which is equivalent to what a classical computer can do. We need something cleverer if we want to score more, using *quantum parallelism*. Such an idea is presented in the next chapter, where we describe an algorithm for fast factorization.

Chapter 2

Fast Factorization

Many people have tried to find efficient algorithms for integer factorization during the last three decades. The most efficient classical algorithm known today is called *number field sieve*. To find the prime factorization of an n-bit integer, this algorithm requires $\exp(\Theta(n^{1/3}\log^{2/3} n))$ operations, which is exponential in the size of the number being factored. It is assumed that it would be impossible to find a fast classical algorithm for factorization. Indeed, the safety of the famous public key cryptographic system *RSA* is based on this assumption. In 1994, Peter Shor [2] described an algorithm, which can factor an integer using $\mathrm{O}(n^2 \log n \log\log n)$ operations. That is, a quantum computer can factor a number *exponentially faster* than the best know classical algorithm.

In this chapter we describe the *quantum Fourier transform*, which is the main gear of Shor's algorithm. Then we describe how factoring is reduced to *order finding* and how a quantum computer can solve this problem. A short description of the appropriate quantum circuits follows. Finally, we show a way to implement *the quantum Fourier transform* semi-classically.

2.1 Quantum Fourier Transform

It is common in mathematics and computer science to solve a problem by transforming it into another problem. One such transformation is the *discrete Fourier transform* (*DFT*), which takes as input a vector of complex numbers, $x_0, x_1, \ldots, x_{N-1}$, where N is a fixed parameter, and it outputs a vector of complex numbers $y_0, y_1, \ldots, y_{N-1}$, defined by

$$y_k \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi \cdot i \cdot j \cdot k / N} \tag{2.1}$$

The *quantum Fourier transform* (*QFT*) is a variant of *DFT* where N is a power of 2. The *QFT* on a computational basis $B = \{|0\rangle,\ |1\rangle,\ \ldots,\ |N-1\rangle\}$ is defined to be a linear operator with the following action on the basis states,

$$|j\rangle \xrightarrow{QFT} \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi \cdot i \cdot j \cdot k / N} |k\rangle \tag{2.2}$$

The action of *QFT* on an arbitrary state may be written

$$\sum_{j=0}^{N-1} x_j |j\rangle \xrightarrow{QFT} \sum_{k=0}^{N-1} y_k |k\rangle \tag{2.3}$$

where the amplitudes y_k are the *DFT* of the amplitudes x_j.

When acting on an n-qubit state the *QFT* can be implemented efficiently by the following quantum circuit

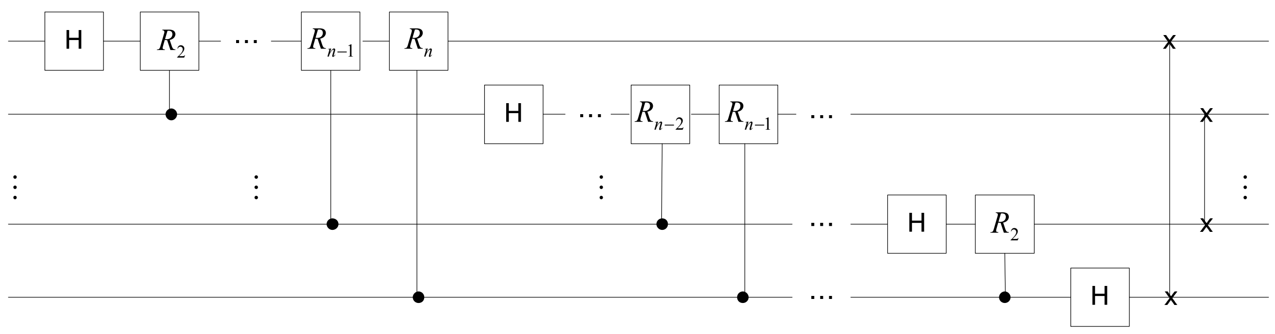

Figure 2.1: The circuit which implements the QFT

where the top wire acts on the most significant qubit and the gate R_k denotes the unitary rotation transformation

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi\, i/2^k} \end{bmatrix} \tag{2.4}$$

The corresponding 2^n by 2^n matrix for the *QFT* has elements

$$F(k, j) = \frac{1}{\sqrt{2^n}} e^{2\pi \cdot i \cdot k \cdot j / 2^n}, \text{ where } \quad k, j = 0, 1, \ldots, 2^n\text{-}1 \tag{2.5}$$

We can count the number of gates that is used by the circuit. We apply a *Hadamard* and n-1 conditional rotations on the first qubit - a total of n gates. On the second qubit we apply one Hadamard and *n*-2 conditional rotations – a total of *n*-1 gates. Continuing in this way and counting the *n/2 Swap* gates, we see that the total number of gates is

$$\frac{n}{2}+\sum_{i=1}^{n} i=\frac{n+n(n+1)}{2}=\frac{n^2+2n}{2}=\Theta(n^2) \tag{2.6}$$

The best classical algorithm for computing the *DFT* on 2^n elements is the *Fast Fourier Transform* (*FFT*) which uses $\Theta(n2^n)$ gates. That is, a classical computer needs exponentially more operations to compute the Fourier transform than the quantum computer. Fourier transform is a key ingredient for many applications, like signal processing, so the exponential speedup sounds terrific. However, the amplitudes in a quantum computer cannot be directly accessed by measurement. This is more or less the same problem that prevents us from taking advantage from *quantum parallelism*. In the next session we describe how *quantum parallelism* and *QFT* can be combined to create a polynomial time factoring algorithm.

In fact this algorithm uses the *inverse QFT* which is implemented by the following circuit

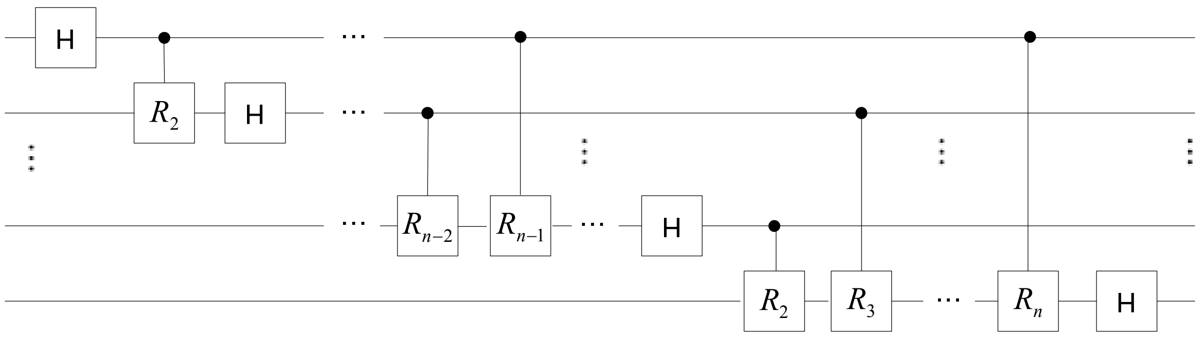

Figure 2.2: The circuit which implements the inverse QFT

The qubit swapping is not necessary, for, at the end of the algorithm all the qubits are measured and we can swap the bits of the measured value classically.

2.2 Reduction of Factoring to Order Finding

For positive integers *a* and *N*, *a*<*N*, with no common factors, the *order* of *a* modulo *N* is defined to be the least positive integer, *r*, such that a^r = 1(*mod N*). In other words, the *order* of *a* modulo *N* is the period of the function

$$f(x)=a^x \bmod N \tag{2.7}$$

We are now going to see how we can factor an integer *N* when we know the period of function (2.7). First, we suppose that *N* is the product of powers of distinct prime integers

$$N = p_1^{e_1} p_2^{e_2} \cdots p_k^{e_k} \tag{2.8}$$

Then we randomly choose an integer $a \neq 1$, $a<N$. If $m = \gcd(a, N) > 1$ then m is a nontrivial factor of *N*, therefore we can find the rest of the factors by dividing *N* by *m*. If $m = 1$, assume that *r* is the *order* of *a* modulo *N*. Then

$$a^r \equiv 1 (\operatorname{mod} N) \tag{2.9}$$

which means that *N* divides $a^r - 1$. If *r* is *even*, we can factorize $a^r - 1$ as

$$a^r - 1 = (a^{r/2} - 1)(a^{r/2} + 1) \tag{2.10}$$

Since *N* divides $a^r - 1$, it should share a factor with $a^{r/2} - 1$ or with $a^{r/2} + 1$ or with both. We can easily extract this factor by using Euclid’s algorithm for finding the greatest common divisor of two numbers.

Of course, we must guarantee that this is not a trivial factor. This happens when $a^{r/2} \pm 1$ are multiples of *N*, so the greatest common divisor of these numbers with *N* is *N*. Fortunately, it is not so likely that *N* divides $a^{r/2} \pm 1$, as we will demonstrate soon. First, *N* cannot divide $a^{r/2} - 1$, since this would imply that

$$a^{r/2} \equiv 1 (\operatorname{mod} N) \tag{2.11}$$

which is impossible, because it redefines the *order* of *a* to be *r*/2. However, it can still happen that *N* divides $a^{r/2} + 1$ and does not share any factor with $a^{r/2} - 1$, that is

$$a^{r/2} \equiv -1 (\operatorname{mod} N) \tag{2.12}$$

To sum up, the reduction of factoring fails only when the *order* of *a* modulo *N* is odd or when equation (2.12) holds. It can be shown that if the procedure is applied with a random *a*, yields a factor of *N* with probability at least $1 - 1/2^{k-1}$, where *k* is the number of different prime divisors of *N* in equation (2.8). A brief sketch of the proof follows.

If r_i is the order of $a(\operatorname{mod} p_i^{e_i})$, then *r* is the least common multiple of all r_i. Consider the largest power of 2 dividing each r_i. If they are all 1, then *r* is odd, so the algorithm fails. The same happens when they are all equal and larger than 1, because (2.12) holds since $a^{r/2} \equiv -1(\operatorname{mod} p_i^{e_i})$ for every *i*. These are the only two cases that the algorithm fails. The Chinese remainder theorem [12] says that choosing an $a(\operatorname{mod} N)$ at random is the same as choosing for each *i* a number $a_i(\operatorname{mod} p_i^{e_i})$ at random. The multiplicative group $(\operatorname{mod} p^e)$ for any odd prime power p^e is cyclic [12], so for any odd prime power $p_i^{e_i}$, the probability is at most ½ of choosing an a_i, having any particular power of two as the largest divisor of its order r_i. Therefore, each of the powers of 2 has at most probability ½ to agree with the previous ones, so all *k* of them agree with probability at most $1/2^{k-1}$. This means that there is at least a $1 - 1/2^{k-1}$ chance that we have chosen a good $a(\bmod N)$. The only constraint is that *N* is odd and not a prime power, that is

$$N \neq p^{\lambda} \tag{2.13}$$

If *N* is even, then we can immediately say that 2 is a factor of *N*. If *N* is a prime power, there exist efficient classical methods to find p and λ in (2.13).

What about if the numbers $a^{r/2} \pm 1$ are so large that they cannot be manipulated efficiently? Fortunately, divisibility by *N* is a periodic property with period *N*, so upon estimation of *r* we can use $(a^{r/2} \pm 1) \bmod N$ instead of $a^{r/2} \pm 1$. Furthermore, very rapid algorithms for modular exponentiation are known.

2.3 Finding the order

We are going to describe the quantum part of Shor's algorithm which is used to find the *order* of *a* modulo *N*. First we find $M = 2^m$, such that $N^2 \leq M < 2N^2$. It is common to use $M = 2^{2L}$, where *L* is the number of bits of *N*. The algorithm uses two quantum registers one of size *m* qubits and the other of size *L* qubits. We initialize the two registers in the state

$$|0\rangle|1\rangle \tag{2.14}$$

Initializing the second register in $|1\rangle$ helps us to compute $a^x \bmod N$ as we describe in *section 2.4*. Next, we apply the *Hadamard* gate to each qubit of the first register to create the superposition

$$\frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle|1\rangle \tag{2.15}$$

Then we compute the function $f(x) = a^x \bmod N$, using as input the first register and storing the result in the second

$$\frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle\left|a^x \bmod N\right\rangle \tag{2.16}$$

Since this function is periodic, with period *r*, we can write (2.16) again as

$$\frac{1}{\sqrt{M}} \sum_{l=0}^{r-1} \sum_{k=0}^{s} |kr + l\rangle\left|a^l\right\rangle \tag{2.17}$$

where *s* is the greatest integer for which $s \cdot r + l < M$. The next step is to apply the *inverse QFT* to the first register to get

$$\frac{1}{\sqrt{M}}\sum_{l=0}^{r-1}\sum_{k=0}^{s}\frac{1}{\sqrt{M}}\sum_{p=0}^{M-1}e^{\frac{2\pi\cdot i\cdot p(k\cdot r+l)}{M}}\left|p\right\rangle\left|a^{l}\right\rangle$$

$$=\frac{1}{M}\sum_{l=0}^{r-1}\sum_{p=0}^{M-1}e^{\frac{2\pi\cdot i\cdot p\cdot l}{M}}\sum_{k=0}^{s}e^{\frac{2\pi\cdot i\cdot p\cdot k\cdot r}{M}}\left|p\right\rangle\left|a^{l}\right\rangle \tag{2.18}$$

In fact, what *QFT* does is to leave nonzero amplitudes only to integers that are multiples of $\frac{M}{r}$. If *r* doesn't divide *M*, then most of the amplitude is attached to integers close to multiples of $\frac{M}{r}$. This means that the first register is in a state $\sum_{q} c_q \left| q\frac{M}{r}\right\rangle$.
Following, we measure the first register to get an outcome

$$p = q\frac{M}{r}\text{, for some } q. \tag{2.19}$$

Most of the time, q and r will be relatively prime, which means that reducing the fraction $\frac{p}{M}\left(=\frac{q}{r}\right)$ to its lower terms will yield a fraction whose denominator is the period *r*. For this purpose, we perform the *continued fraction expansion* of $\frac{p}{M}$ to find the convergents $\frac{p_i}{q_i}$. The smallest q_i for which $a^{q_i} \equiv 1(\operatorname{mod} N)$, if such q_i exists, is candidate to be the *order* of *a* modulo *N*.

The *continued fraction expansion* of a real number is a way to describe it in terms of integers, using expressions of the form

$$[a_0, a_1, \ldots, a_n] \equiv a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ldots + \cfrac{1}{a_n}}}} \tag{2.20}$$

For example,

$$\frac{31}{13} = 2+\frac{5}{13} = 2+\cfrac{1}{\cfrac{13}{5}} = 2+\cfrac{1}{2+\cfrac{3}{5}} = 2+\cfrac{1}{2+\cfrac{1}{\cfrac{5}{3}}} = 2+\cfrac{1}{2+\cfrac{1}{1+\cfrac{2}{3}}} = 2+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{\cfrac{3}{2}}}} = 2+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{2}}}}$$

The *mth* convergent, $m < n$, to (2.20) is defined to be $\frac{p_m}{q_m} = [a_0, a_1, \ldots, a_m]$. The continued fractions algorithm can be computed efficiently by a classical computer using the following equations

$$\begin{aligned} p_0 &= a_0 \\ q_0 &= 1 \\ p_1 &= a_0 a_1 + 1 \\ q_1 &= a_1 \\ p_n &= a_n p_{n-1} + p_{n-2} \\ q_n &= a_n q_{n-1} + q_{n-2} \end{aligned} \tag{2.21}$$

The algorithm finishes after a finite number of steps for a rational number. This is the case for the number $\frac{p}{M}$. If p and M are k bit integers the continued fraction expansion can be computed using $O(k^3)$ operations.

2.4 Example: Factoring 15

An example is necessary, so we are going to use Shor's algorithm to factor $N = 15$. First, we have to choose a random number smaller than N that has no common factors with N, for example $a = 7$. We initialize two registers of 8 and 4 bits respectively. Then we apply the Hadamard transformation to the first register to get the state

$$\frac{1}{\sqrt{256}} \sum_{x=0}^{256-1} |x\rangle|1\rangle = \frac{1}{16} \sum_{x=0}^{255} |x\rangle|1\rangle = \frac{1}{16}[|0\rangle + |1\rangle + \ldots + |255\rangle]\ |1\rangle$$

The next step is to compute the modular exponentiation

$$\frac{1}{16} \sum_{x=0}^{255} |x\rangle|7^x \bmod 15\rangle = \frac{1}{16}[|0\rangle|1\rangle + |1\rangle|7\rangle + |2\rangle|4\rangle + |3\rangle|13\rangle + |4\rangle|1\rangle + |5\rangle|7\rangle + \ldots]$$

We can now apply the *inverse QFT* to the first register, but for clarity we will do an extra step which is not necessary. Since the second register is not used again, we can measure it. The measurement gives 1, 7, 4 or 13. Suppose that it gives 4. *Entanglement* indicates that the measurement leaves the first register in a superposition of basis vectors $|k\rangle$ for any $k < 255$ such that 7^k *mod 15 = 4*. The state is now

$$\frac{1}{8}[|2\rangle + |6\rangle + |10\rangle + |14\rangle + \ldots]\ |4\rangle$$

The sharp-eyed reader may have noticed that the order r = 4 is "stored" in the superposition of the first register. However, as mentioned before, we cannot just measure it, because that will only give a single number. We cannot even copy the state, to measure it several times, so we need something cleverer. What we need is the *inverse QFT* which when applied to the first register produces the state

$$\tfrac{1}{2}\left[|0\rangle + |64\rangle + |128\rangle + |192\rangle\right]$$

Measuring the first register will give 0, 64, 128 or 192 with probability ¼. Suppose we obtain 64. Then the continued fraction expansion of $64/256$ gives $1/4$, so r = 4 is the order of 7 modulo 15. We can apply the ideas mention in *section 2.2*, to get

$$\textit{Possible factor} = \gcd(7^{4/2} - 1,\ 15) = 3$$
$$\textit{Possible factor} = \gcd(7^{4/2} + 1,\ 15) = 5$$

Measuring 0, means that we cannot apply the continued fraction algorithm. Measuring 128 gives r = 2, which is not the correct *order*. Thus, it is a good idea to try some small multiples of r like $2r$ and $3r$. Measuring 192 gives the correct order 4.

2.5 Quantum Circuits for Modular Exponentiation

We have purposely avoided mentioning how we can evaluate the modular exponential $a^x \bmod N$ using quantum circuits, which is the dominant part of Shor's algorithm. The simplest way is to have a circuit that multiplies by a a total of x-1 times and then taking the modulo N. Fortunately, there is a trick called *repeated squaring*, that speeds up the computation. If x is an L-bit number, we can use its binary expansion to write

$$a^x = a^{\sum_{i=0}^{L-1} x_i 2^i} = \prod_{i=0}^{L-1} \left(a^{2^i}\right)^{x_i} \tag{2.22}$$

Moreover

$$a^{2^i} = \left(a^{2^{i-1}}\right)^2 \tag{2.23}$$

which means that $a^{2^i} \bmod N$ can be computed by squaring $a^{2^{i-1}}$. The pseudo-code describing the process of computing $a^x \bmod N$ is

$$\textit{For } i = 0 \textit{ to } L\text{-}1, \textit{ if } x_i = 1 \textit{ then Multiply } a^{2^i} \bmod N \tag{2.24}$$

The *if... then* statement can be computed using controlled gates. If have a gate $OMULN(k, N)$ that multiplies with k and then takes the modulo N

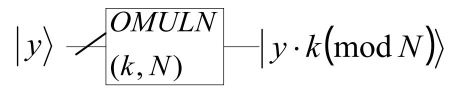

Figure 2.3: Operator for modulo N multiplication

then the circuit for Shor's algorithm is depicted by the following figure

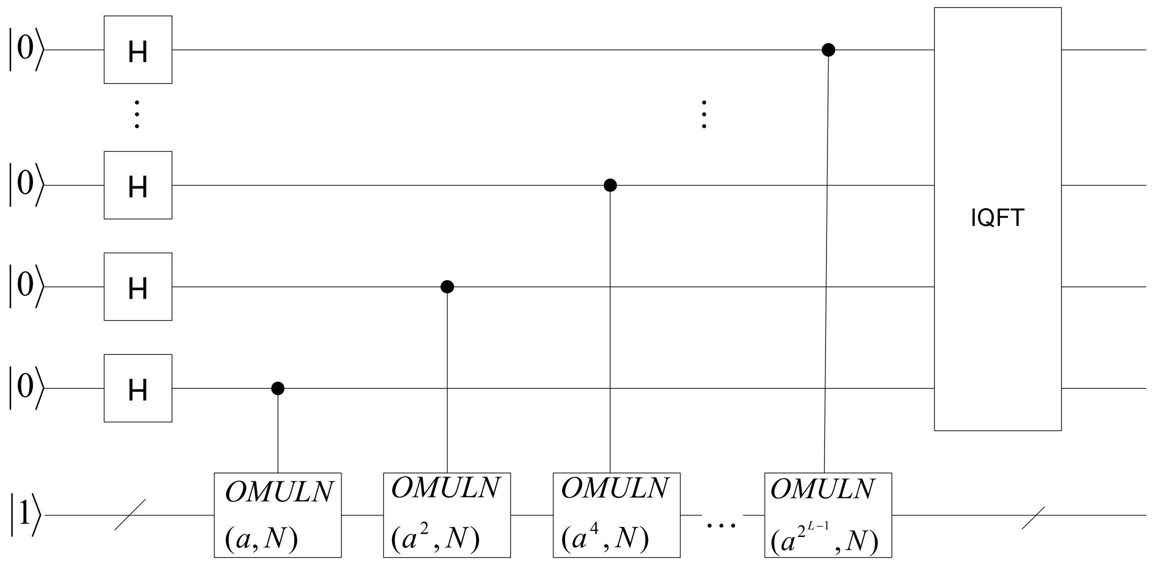

Figure 2.4: Circuit implementing Shor's algorithm

Notice that the register which stores the result of modular exponentiation is initialized in $|1\rangle$, not $|0\rangle$. Moreover, the order in which we apply the *OMULN* gates doesn't matter, since the same product is produced. We have to explain how the *OMULN* operator is implemented. If *y* is an *M*-bit integer, using its binary notation we have

$$k \cdot y(\bmod N) = \left(\sum_{i=0}^{M} y_i 2^i\right) \cdot k(\bmod N) = \sum_{i=0}^{M} y_i \left[2^i k(\bmod N)\right] \tag{2.25}$$

The product $k \cdot y(\bmod N)$ can be computed by the pseudo-code

$$\textit{For } i = 0 \textit{ to } M\text{-}1, \textit{ if } y_i = 1 \textit{ then ADD } 2^i k \bmod N \tag{2.26}$$

Once again, the *if... then* statement is implemented using controlled gates. The modular addition is computed by the following operator

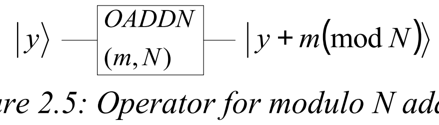

Figure 2.5: Operator for modulo N addition

Combining the ideas of equations (2.25) and (2.26) we see that the *OMULN* operator can be constructed by chaining *M controlled-OADDN* gates. The *OADDN* gate can be implemented using a *multiplexed* modulo adder which implements the following pseudo-code

$$\textit{If } (N - m > y) \textit{ ADD } m \textit{ else ADD } m\text{-}N \tag{2.27}$$

The multiplexed adder uses a comparator and its result qubit controls which number is to be added. In fact, the above description only captures the basic ideas of how we can build the quantum circuit for modular exponentiation. David Beckman et al. [13] describe in full detail how to implement such a circuit, using the appropriate number of *ancilla* qubits and erasing garbage (see *section 1.11*). It turns out that if the number N to be factored is L-bits, the modular exponentiation circuit uses $2L+1$ *ancilla* qubits, which together with the L qubits of the register that stores the result, make a total of $3L+1$. Supposing that we use $2L$ qubits in the first register of Shor's algorithm we need $5L+1$ qubits to factor an L-bit integer. In the next section we describe a way to reduce this number to $3L+2$, that is, we use only one qubit in the first register to compute the inverse *QFT*.

2.6 Semi-classical Fourier Transform

Griffiths and Niu [14] pointed out that the measurements of the qubits of the first register in Shor's algorithm can be performed before the controlled rotations. Moreover, we can replace the quantum controlled rotations with semi-classically controlled rotations. This means that the control qubit is measured and if the outcome is 1, the rotations are done quantumly. We give a simple example to make things clearer. Suppose that we have only three qubits in the first register of Shor's algorithm. Then the corresponding circuits is

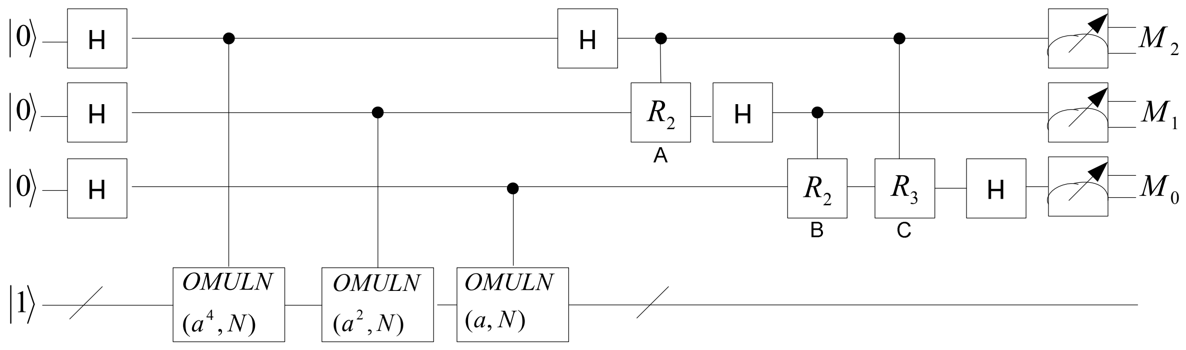

Figure 2.6: Running Shor's algorithm with just three qubits

What Griffiths and Niu say is that the top qubit can be measured before applying gates A and C. Only if the outcome of the measurement is 1 we apply gates A and C. Following this procedure we can use just one control qubit. Suppose a scenario where the circuit of *Figure 2.6* gives $M_2 = 0$, $M_1 = 1$, $M_0 = 0$. Shor's algorithms is equivalently implemented by the following circuit

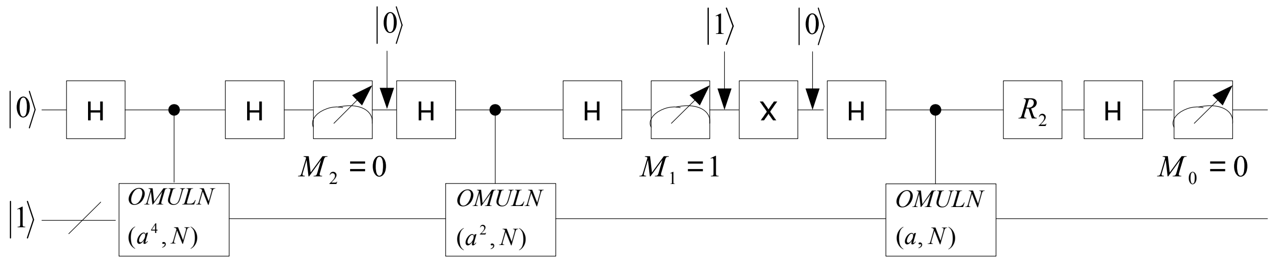

Figure 2.7: Running Shor's algorithm using the semi-classical QFT

We see that we apply only gate B of *Figure 2.6*, since only the second measurement gives an output 1. After a measurement of 1, the control qubit collapses in the state $|1\rangle$, so we have to transform it back to state $|0\rangle$ before continuing. This is the purpose of the quantum *Not* gate in *Figure 2.7*. The binary number $M_2M_1M_0 = 010 = 2$ is the outcome *p* (see equation (2.19)) when measuring the first register of Shor's algorithm. Using just one control qubit not only saves space, but it may also reduce the number of rotation gates, since they are only applied after a measurement of 1. The overhead of *Not* gates for reinitializing the state to $|0\rangle$ is negligible.

Chapter 3

Simulating Quantum Computers

Current technology allows us to build quantum computers with just a few qubits. In addition, these quantum computers are not very stable, so they cannot be used to run quantum algorithms which require many qubits. Simulation on a classical computer can instead be used to test quantum algorithms, but this also turns out to be a difficult task. In this chapter we describe the main reasons why this task is difficult and then we mention the related work on simulating quantum computers.

3.1 The Main Difficulties

We have seen in *section 1.4* that the state of an *n*-qubit system is described by a vector which has 2^n elements. The naïve approach of storing each element of the state vector needs $\mathrm{O}(2^n)$ space and time to be manipulated. Furthermore, an *n*-qubit operator is described by a 2^n by 2^n matrix. Storing each element, the whole matrix requires space $\mathrm{O}(2^{2n})$. Therefore, a simulator that stores every amplitude of the state vector and every element of each operator would require exponential (in the number of qubits) time and memory, which in turn means that simulation would be possible only for a small number of qubits. Another problem is *entanglement* which makes it impossible to consider each qubit as an independent system. Thus, we cannot simulate the actions on each qubit and then combine the results.

We have to find ways to compress the state vectors and the operators' matrices. A simple solution is storing only nonzero amplitudes or/and avoiding storing the same amplitudes more than once, similar to the ways described in literature to store sparse matrices. However, some datastructures for storing compressed matrices have to be uncompressed before using them, which is still a major problem. An efficient simulator should use a datastructure that keeps the matrices and vectors compressed, and performs operations on them quickly and without using excessive space.

3.2 Related Work

The need for quantum computer simulators has been translated into a number of available simulators. J. Wallace [16, 17] has described a number of quantum simulators, but only a few implementations are still functional. The following table shows some of the simulators that can be found in [17]

Name	Description
QuBit	QuBit is a library to support Quantum Superpositions in C++
Quantum-Entanglement-0.31	QM-like entanglement of variables in Perl
OpenQUACS	OpenQUACS is an Open-source general-purpose Quantum Computer Simulator written in the Maple programming language. It comes as a precompiled Maple library or Maple source and has a full tutorial included
QuCalc	QuCalc is a library of Mathematica functions whose purpose is to simulate quantum circuits and solve quantum computation problems
OpenQubit 0.2.0	C++ quantum computer simulator which aims to demonstrate Shor's algorithm, and its efficiency on a quantum computer. Current development version is NewSpin 0.3.3a
QCL	QCL (Quantum Computation Language) is a high level, architecture independent programming language for quantum computers
Q-gol 3	A high-level programming language to allow researchers to describe quantum algorithms

Table 3.1: Some of the simulators that can be found in [17]

A very important work comes from B. Omer [18] who has implemented a procedural language for quantum programming. The idea behind it is that when a quantum computer is going to be constructed, we should have a programming language to program it. The programming language is called *QCL* and it treats operators as functions. *QCL* includes many concepts of traditional programming languages like variables, control structures and functions but it is also equipped with quantum properties. Since no quantum computer is available yet to be programmed, *QCL*'s interpreter is connected with a C++ library that simulates the action of a quantum computer, called *QCLIB* [19]. This library stores only nonzero amplitudes of the state vector using a hash table and a linear array. A basevector is mapped onto a hashtable using a hash function to get a pointer (if any exist) to the array entry where the amplitude for this basevector is stored. In this way, *QCLIB* achieves O(1) complexity for finding and inserting a basevector's amplitude. For matrices which represent unitary operators, *QCLIB* stores only the nonzero elements of each row in an array of linear lists. *QCL* is enriched with simulation code for several algorithms like Shor's and Grover's. Although the main concept behind *QCL* is quantum programming, it can be easily used as a simulator. *QCL* can simulate efficiently a small number of qubits and can run Shor's algorithm to factor up to about 10-11 bit numbers depending on the available memory. However, it has several weaknesses, e.g. applying the *Hadamard* gate to each qubit of a 23-qubit register consumes excessive memory.

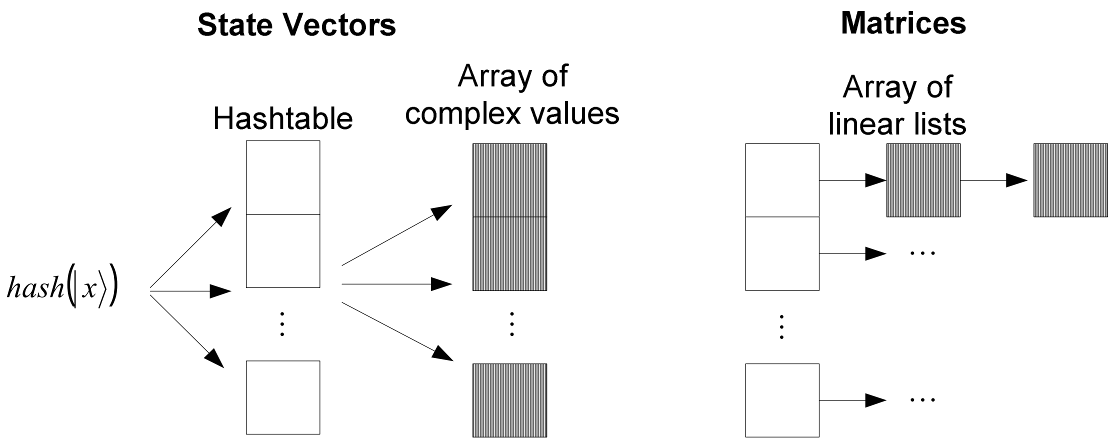

Figure 3.1: Datastructures used by QCL to represent vectors and matrices

S. Bettelli [20] proposes another approach for quantum programming. Actually, he implements *libquantum*, a C++ library for quantum programming which treats operators as objects. This library produces a stream of bytecode which can be used to drive a quantum device or can be passed to a quantum simulator. Like *QCL*, a simulator acts like a quantum computer to run the program. Once again, applying the *Hadamard* gate to a 25-qubit register over-consumes the free memory. Some other works are available for quantum programming ([21], [22]), but they are still in a theoretical level, so there is no way to use them for simulation.

The efforts described above target quantum programming, not quantum simulation. Several programs exist that can be used for quantum simulation. For example, we can use *Matlab* or *Octave*, which are designed to maneuver matrices and vectors and are capable of performing matrix multiplications and tensor products. Unfortunately, it turns out that they are not so efficient in simulating quantum circuits, since they usually run out of memory for circuits with more than 12-14 qubits. National Institute of Standards and Technology [23] provides source code and examples for *QCSim* quantum computer simulator. *QCSim* does not compress matrices and vectors, and its creators mention that it can only allocate up to 13 qubits. Surely, this number of qubits is not enough for factoring big integers using Shor's algorithm.

A simulator that uses a more sophisticated way to represent state vectors and operators is *QuIDDPro* [24, 25]. This simulator uses a variant of *Algebraic Decision Diagrams* (*ADDs*) called *Quantum Information Decision Diagrams*, which achieve quick simulation of quantum circuits combined with low memory consumption. The final version of *QuIDDPro* is not available yet, but results show that *QuIDDPro* can simulate Grover's algorithm faster and with much less memory than *Matlab* and *Octave*. The low memory usage allows *QuIDDPro* to be able to simulate circuits with many qubits. Our work uses *Algebraic Decision Diagrams* too, so we are going to describe them in detail in the next chapter.

Chapter 4

Binary Decision Diagrams

In this chapter we are going to describe the main properties of *Binary Decision Diagrams* (*BDDs*) and ways to manipulate them. Then we present *Algebraic Decision Diagrams* (*ADDs*), a variant of *BDDs*, which are used to represent matrices. Finally, we describe how to perform matrix multiplications and tensor products using *ADDs* and the corresponding time complexity.

4.1 Reduced Ordered Binary Decision Diagrams

Binary Decision Diagrams were introduced by Lee [26] in 1959 and later by Akers [27]. Bryant [28] eliminated redundancy from *BDDs* by formulating some limitations and algorithms to manipulate them, ending up in a more powerful datastructure called *Reduced*, *Ordered*, *BDD* (*ROBDD*). A *ROBDD* is a datastructure which represents a boolean function $f(x_0, x_1, \ldots, x_n)$ using a directed acyclic graph. The graph has vertices of two types, terminals and non-terminals. Each non-terminal, also called internal, vertex of the graph is associated with a variable x_i, and has outdegree 2. One outgoing edge (usually represented by a solid line) denotes the assignment of 1 (or true) to variable x_i and the other (dashed line) denotes the assignment of 0 (or false). Terminal vertices, also called external, are not associated with any variable; they have outdegree 0 and store a boolean value of either 0 or 1. A *BDD* is reduced and ordered when the following *conditions* hold:

1. There is no internal vertex *v* with its two outgoing edges pointing to the same vertex *u*.

2. There are no vertices *v* and *u* such that the subgraphs rooted at *v* and *u* are isomorphic.

3. *Condition* 2 implies that there are no terminal vertices with the same value.

4. If x_i is the associated variable of a vertex and if x_j and x_m are the variables of the vertices pointed by its edges then $i < j$ and $i < m$. This means that while we are traversing a path of the graph we should always meet variables with higher indexes.

Let see the *BDD* representing the Boolean function $f(x_0, x_1) = x_0 + x_1$

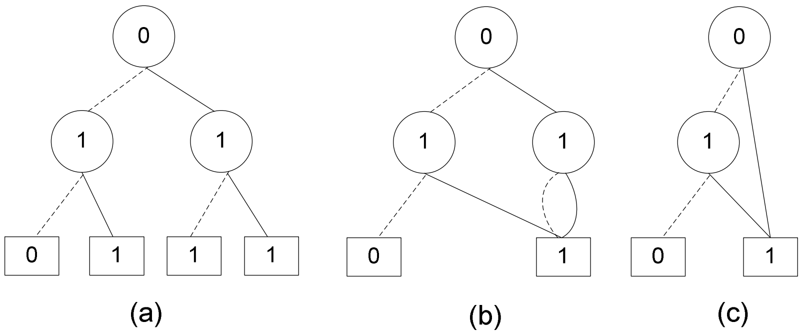

Figure 4.1: BDDs for function $f(x_0, x_1) = x_0 + x_1$

Figure 4.1a shows the unreduced *BDD* of $f(x_0, x_1) = x_0 + x_1$. Using *condition* 3 we reduce the three identical terminal vertices into one, to get *Figure 4.1b*. Here we see that there exists one vertex which violates *condition* 1 so the graph can be reduced even more. *Figure 4.1c* shows the *ROBDD* for $f(x_0, x_1) = x_0 + x_1$. Translating the *ROBDD* starting from top and moving down, we can say:

If variable x_0 is 1 *then* the function evaluates to 1, *else* if variable x_1 is 1 *then* the function evaluates to1, *else* it evaluates to 0.

Each non-terminal vertex of a *ROBDD* expresses an *if…then…else* statement. If the value of the corresponding variable is 1 traversal takes the *then* edge, otherwise it takes the *else* edge. Traversing a *ROBDD* using a variable assignment ends in a terminal vertex. This vertex contains the value to which the function evaluates for this specific variable assignment. The *then* edge is also known as *high* and the *else* edge is also known as *low*.

Bryant described some very important algorithms for manipulating *ROBDDs*, which make them so practical. The most important algorithm is called *REDUCE* and as it is denoted by its name, it takes as input a *BDD* and return its equivalent *ROBDD*. We have to say that each function $f(x_0, x_1, \ldots, x_n)$ has a unique *ROBDD*. For example, the transformation of the *BDD* in *Figure 4.1a* to that in the *Figure 4.1c* is the action of *REDUCE*. Bryant showed that if |*G*| is the number of vertices in the *BDD* to be reduced, then *REDUCE* has time complexity $O(|G|\log|G|)$. The only restriction on the input *BDD* is that it should obey to *condition* 4.

Another operation on *ROBDDs* that is extremely useful is *APPLY*. *APPLY* takes as input two *ROBDDs* representing functions $f_1(x_0,x_1,\ldots,x_n)$ and $f_2(x_0,x_1,\ldots,x_n)$ together with an operator <*op*> and produces a *ROBDD* which represents $f_1 < op > f_2$. Before describing how *APPLY* works we have to give the notation to be used. For a vertex *v* which is associated with variable x_i, *index*(*v*) = *i*. The *high* and *low* edges of a vertex *v* point to vertices *low*(*v*) and *high*(*v*) respectively. A terminal vertex *v* has value *val*(*v*). To apply the operator <*op*> to the functions represented by graphs with roots *v* and *u*, we must consider several cases. First, if both *v* and *u* are terminal vertices then the result graph consist of a terminal vertex having value *val*(*v*) <*op*> *val*(*u*). Now suppose that at least one of the two vertices is non-terminal. *APPLY* is implemented by a recursive traversal of the two *ROBDD* operands. For every pair of vertices visited during the traversal, it produces a non-terminal vertex using the rules of the following figure

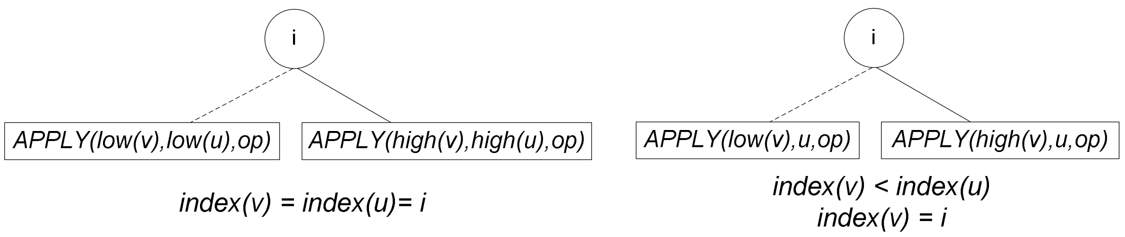

Figure 4.2: The recursive rules used by APPLY when at least one vertex is non-terminal

The algorithm does not have to evaluate a given pair of subgraphs more than once. We can use a structure, i.e. a hash table, where we keep entries of the form (*v, u, k*) which means that the result of *APPLY* on the subgraphs rooted by vertices *v* and *u* is a graph rooted by vertex *k*. In addition, if the algorithm is applied to two vertices where one is a terminal vertex with a *controlling value*, then it can return immediately a terminal vertex with this value. The controlling value for boolean *AND* is 0, while the controlling value for boolean *OR* is 1. The produced graph is not reduced, so *APPLY* has to call *REDUCE* at the end. If the number of vertices of the input *ROBDDs* are $|G_1|$ and $|G_2|$, then the time complexity of *APPLY* is $O(|G_1||G_2|)$. An example follows to illustrate the steps described above

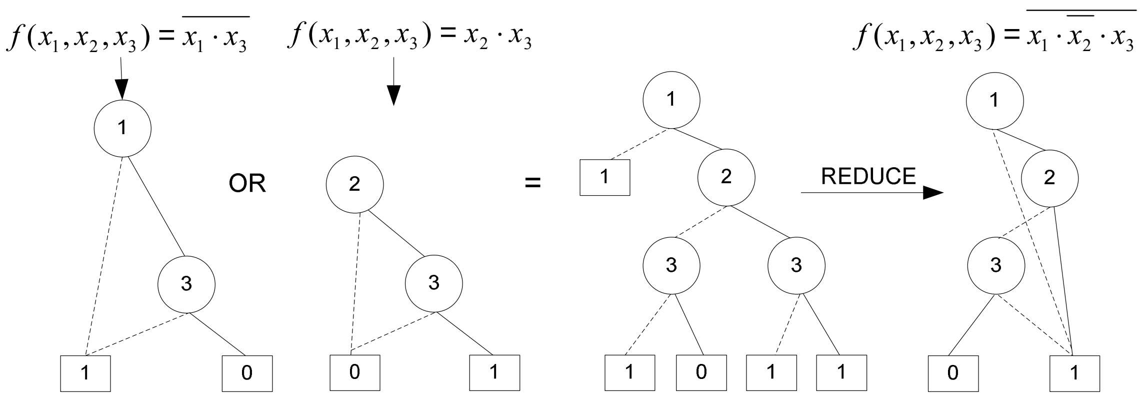
$f(x_1,x_2,x_3)=\overline{x_1\cdot x_3}$
$f(x_1,x_2,x_3)=x_2\cdot x_3$
$f(x_1,x_2,x_3)=\overline{x_1\cdot\overline{x_2}\cdot x_3}$
1
3
1
0
OR
2
3
0
1
=
1
1
2
3
3
1
0
1
1
REDUCE
1
2
3
0
1

Figure 4.3: $\left(\overline{x_1\cdot x_3}\right)$ *OR* $\left(x_2\cdot x_3\right) = \overline{x_1\cdot\overline{x_2}\cdot x_3}$

Another important operation on *ROBDDs* is *RESTRICT*. This operation restricts the vertices associated with a specific variable to their *low* or *high* children. In other words, *RESTRICT* assigns the value 0 or 1 to a variable x_i in the function represented by the *ROBDD*. The operation can be easily implemented by traversing the *ROBDD* and changing the pointer to vertices v with variables x_i to point to *low*(v) or *high*(v). The complexity of *RESTRICT* is $O(|G|)$. Of course, we have to call *REDUCE* to get the final restricted *ROBDD*. If B is a *ROBDD* then we denote the restriction $x_i = 1$ of B as B_{x_i} and its restriction $x_i = 0$ as $B_{\overline{x_i}}$. *Figure 4.4a* shows the original *ROBDD B*. *Figure 4.4b* shows $B_{\overline{x_1}}$ and *Figures 4.4c,d* show B_{x_1} before and after *REDUCE*.

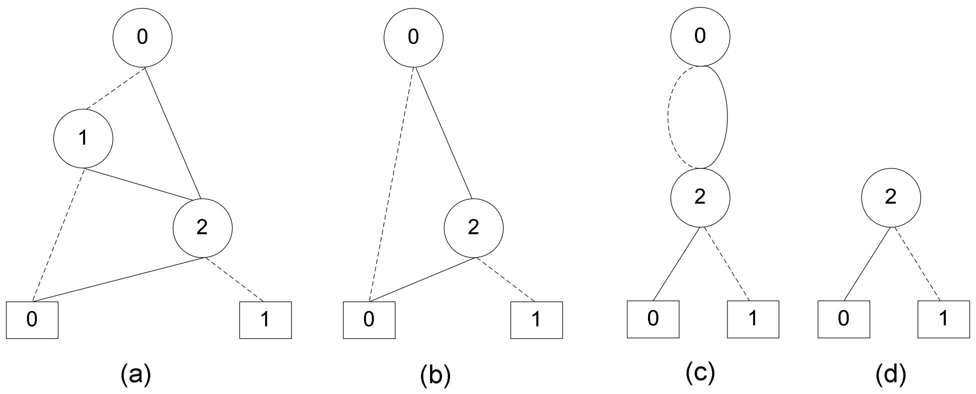

Figure 4.4: (*a*) *Original ROBDD,* (*b*) *Restricted by* $x_1 = 0$, (c) *Restricted by* $x_1 = 1$, (d) *Reducing the BDD of* (c)

4.2 Algebraic Decision Diagrams

Algebraic Decision Diagrams [29] extend *ROBDDs* by allowing terminal vertices to store any numerical value. Thus, an *ADD* can represent a function $f(x_0, x_1, \ldots, x_n)$ which evaluates to any numerical value. The variables x_i still take values from the boolean domain {0,1}. The following figure shows the *ADD* for the function $f(x_0, x_1) = 5x_0x_1 + 3x_1$

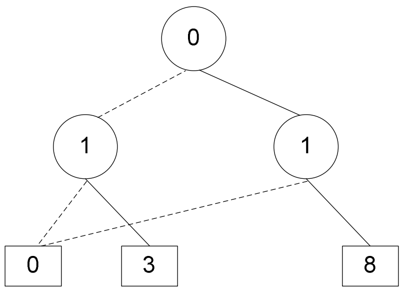

Figure 4.5: ADD for $f(x_0, x_1) = 5x_0x_1 + 3x_1$

We have to mention that the variable ordering is of key importance for the size of the *BDD*. For example, the function $f(x_0, x_1) = 5x_1x_0 + 3x_0$ is represented by the *ADD*

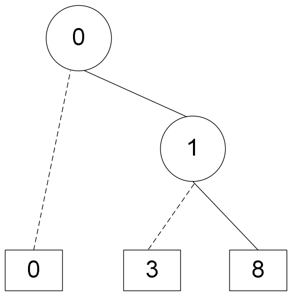

Figure 4.6: ADD for $f(x_0, x_1) = 5x_1x_0 + 3x_0$

which has less vertices than that of the *Figure 4.5*. This means that if we have a function $f(x, y) = 5xy + 3x$ we achieve better compression if we assign x_0 to x and x_1 to y, than assigning x_0 to y and x_1 to x. Finding the best variable ordering is a *coNP-complete* problem. However, we can find a quit good variable ordering for most problems based on our experience and just a few experimental results.

We are going to use *ADDs* to represent matrices and vectors. We can write the row and column indexes of a matrix using their binary representation and then use each bit of this representation as a variable of the function which represents the matrix. For example if R_0 is the row index and C_0 is the column index, R_0, C_0 = {0, 1}, then the matrix

$$\begin{array}{ccc} C_0 & 0 \quad 1 & \\ & \begin{bmatrix} a & b \\ c & d \end{bmatrix} & \begin{array}{c} 0 \\ 1 \end{array} \\ & & R_0 \end{array} \tag{4.1}$$

can be represented by the function $f(R_0, C_0) = a\overline{R_0C_0} + b\overline{R_0}C_0 + cR_0\overline{C_0} + dR_0C_0$. Supposing that variable R_0 precedes C_0, the *ADD* for matrix (4.1) is

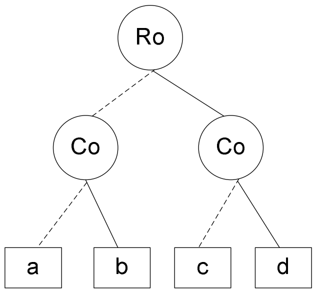

Figure 4.7: ADD representing matrix (4.1)

Let's move to a more complex example. Suppose we have the matrix produced by the tensor product of two *Hadamard* matrices,

$$H \otimes H = \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} \otimes \begin{bmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{bmatrix} = 1/2 \begin{array}{c|cccc|c} C_0C_1 & 00 & 01 & 10 & 11 & \\ & 1 & 1 & 1 & 1 & 00 \\ & 1 & -1 & 1 & -1 & 01 \\ & 1 & 1 & -1 & -1 & 10 \\ & 1 & -1 & -1 & 1 & 11 \\ & & & & & R_0R_1 \end{array} \tag{4.2}$$

The function that represents this matrix has four variables; therefore we have to decide for the variable ordering before giving the corresponding *ADD*. We choose the interleaved variable ordering, that is R_0, C_0, R_1, C_1, ...,R_n, C_n. Thus, the *ADD* representing the matrix (4.2) is

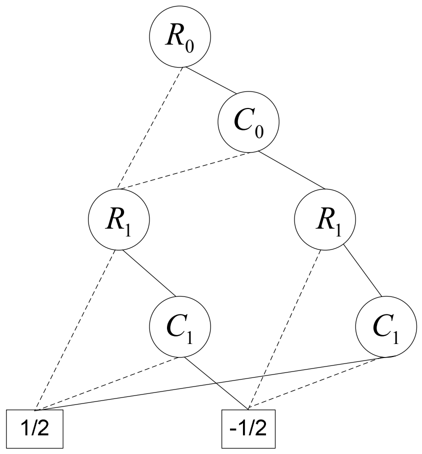

Figure 4.8: ADD representing the matrix (*4.2*)

The interleaved variable ordering is demonstrated to be efficient for representing certain rectangular matrices [30] and offers great compression for matrices that have regular block sub-matrices, which is a usual feature of the tensor product [25].

We have skipped mentioning that *ADDs* represent only matrices or vectors, which have a dimension that is a power of 2. If that's not the case, the matrix has to be padded with zeros. Fortunately, all state vectors and operators in quantum mechanics do have a dimension that is a power of 2. There is another variant of *BDD* that can represent matrices, called *Multi-Terminal Binary Decision Diagram* (*MTBDD*) [30]. A *MTBDD* is almost identical to an *ADD*, so it can be used instead. We have used *ADDs* for our implementation, but *MTBDDs* can be used to provide the same efficiency.

4.3 Matrix Operations with *ADDs*

We have seen how an *ADD* can represent a matrix, but as said in *section 3.1* we need efficient ways to manipulate a compressed matrix datastructure. It turns out that an *ADD* can perform the basic matrix operations without having to be uncompressed. First, we can use *APPLY* to perform element-wise operations, like matrix addition and subtraction, by setting the *op* argument to + or -.

A recursive algorithm for matrix multiplication is also available [29]. Suppose that the *ADD K*(*R,C*) represents a matrix *K* which has row variables R_i and column variables C_i. We want to multiply *ADD A*(*x,z*) with *ADD B*(*z,y*). Notice that the definition of matrix multiplication demands that the column variables of *A* agree with the row variables of *B*. Let *u* be the variable of least index in either *A* or *B*. Then the recursive action of the algorithm is the following

1. if *u* is z_i then
   $$A(x,z) * B(z,y) = APPLY((A_u(x,z) * B_u(z,y)),(A_{\bar{u}}(x,z) * B_{\bar{u}}(z,y)),+)$$

2. if u is either x_i or y_i then

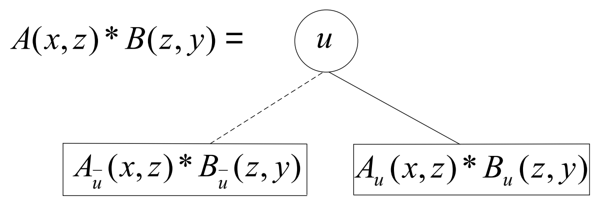

Figure 4.9: The second recursive option for matrix multiplication using ADDs

We see that in every step, the algorithm restricts exactly one variable; therefore the algorithm always proceeds to smaller *ADDs*. If *A* and *B* become terminal vertices *v* and *u*, then *A*B* returns a terminal vertex with value *val*(*v*) * *val*(*u*). Moreover, if *A* or *B* is a terminal vertex that has value 0, then we can immediately return a terminal vertex with value 0. The algorithm does not require uncompressing the matrices, but it has to keep trace of the missing *z* variables. That is, while we are traversing the *ADDs*, we count the missing *z* variables, and then we multiply the result by $2^{\#\ missing\ z}$. If *A*(*x,z*) and *B*(*z,y*) have $|G_1|$ and $|G_2|$ vertices respectively, then the time complexity of multiplication is $\mathrm{O}\left((|G_1||G_2|)^2\right)$. Practically, we can reduce the runtime of the algorithm by not multiplying the same pair of vertices more than once, an idea which is also used in *APPLY*. The resulting *ADD* is not reduced, so we have to call *REDUCE*.

Next, we need an algorithm to implement tensor products. Tensor product $A \otimes B$ can be easily implemented using *ADDs*. Recalling equation (1.7), we see that each element of *A* is multiplied with *B*. But the elements of matrix *A* are stored in the terminal vertices. If we change any pointer to terminal vertices of *ADD A* to point to the root of a new instance of *ADD B*, then we are almost done. If *ADD A* has *n* row and *n* column variables, we must add *n* to each variable of *B*. Moreover, we have to multiply the terminals of each instance of *B* with the value of the corresponding terminal vertex of *A*. An example should make things clear

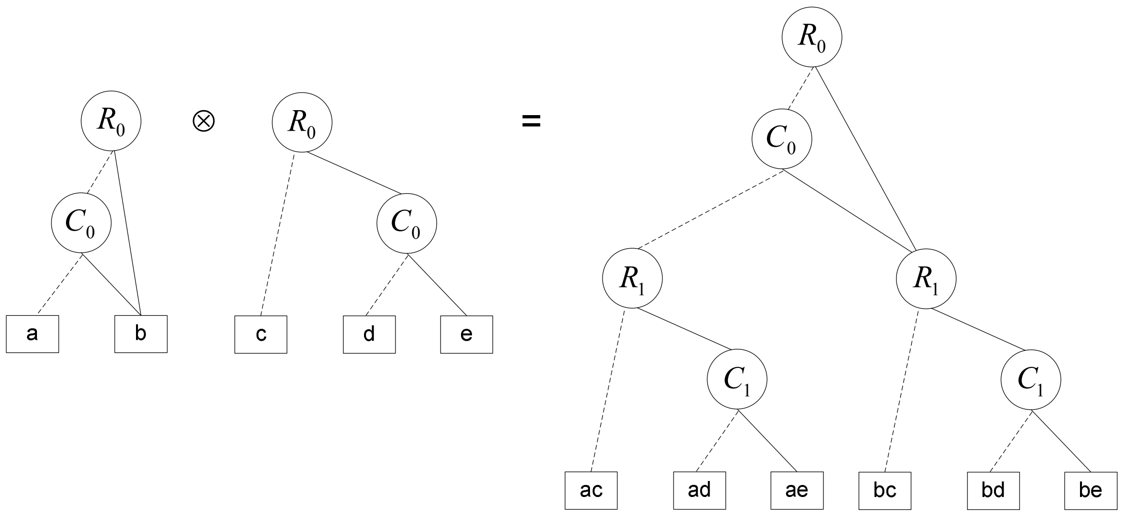

Figure 4.10: Tensor product of matrices represented by ADDs

Once again, *REDUCE* is necessary, because the resulting *ADD* may have redundant vertices. The algorithm requires traversing *A* to find all the terminal vertices, and then traverse *B* to find its terminal vertices too, and perform the required variable shifting simultaneously. If *A* has *k* terminal vertices and a total of $|G_1|$ vertices, and *B* has $|G_2|$ vertices, then the tensor product $A \otimes B$ has time complexity $O(|G_1| + k|G_2|)$. Generally, *k* is $O(|G_1|)$, so $O(|G_1| + k|G_2|) = O(|G_1||G_2|)$. Practically, *k* is usually 2 or 3 so the complexity can be $O(|G_1| + |G_2|)$, which is linear in the size of the two operands. An improvement can be made in the case where the terminal vertex of *A* has value 0. Then we have to do nothing for this terminal.

4.4 Efficiency of the *ADD* representation

The complexities of the various operations on *ADDs* seem quit good provided that the sizes of the operands are not big. That is, if the sizes of the *ADD* operands are polynomial in the number of qubits, then the aforementioned operations can be performed efficiently. Although it is impossible to know the size of an *ADD* representing an arbitrary matrix, we can check the sizes of the most important ones. First, we need to know how many vertices are used by an *ADD* to represent a basis state $|x\rangle$ of *n* qubits. It turns out that using the interleaved variable ordering such an *ADD* has exactly *n*+2 vertices. The following figure shows the *ADD* representing the state vector $|0\rangle^{\otimes 10} = |0\rangle|0\rangle|0\rangle|0\rangle|0\rangle|0\rangle|0\rangle|0\rangle|0\rangle|0\rangle$

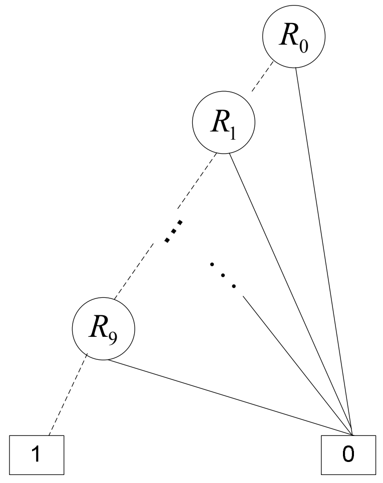

Figure 4.11: ADD representing the state vector $|0\rangle^{\otimes 10}$

Moreover, a state in an equal superposition of arbitrary many qubits is represented by an *ADD* which has a single terminal vertex.
Figure 4.12 shows the *ADD* for the identity matrix *I*

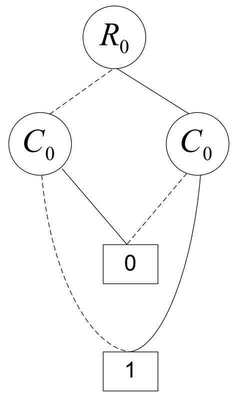

Figure 4.12: ADD representing the 2x2 identity matrix

Larger identity matrices are produced by repeated tensoring. To create the tensor product $I \otimes I$ we have to replace the terminal vertex which has value 1 in *Figure 4.12* with a whole *ADD I*

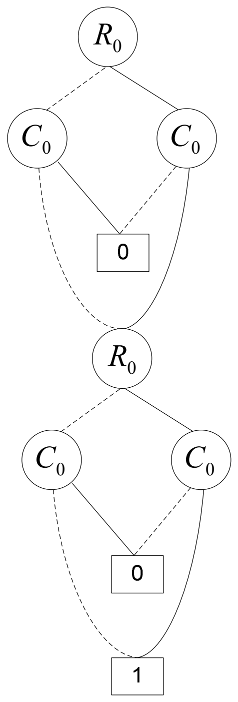

Figure 4.13: Unreduced ADD representing the 4x4 identity matrix produced by $I \otimes I$

The *ADD* of *Figure 4.13* is not reduced, since there are two terminal vertices with value 0. After this reduction, the *ADD* for $I \otimes I$ should have 3 more vertices than the *ADD* for *I*. Following this sequence, we find that the *ADD* for $I^{\otimes n}$ has $3n$+2 vertices. Similarly, the *ADD* for the gate $H^{\otimes n}$ has 4*n* vertices. We have just seen that some of the most basic matrices and vectors can be represented with *ADDs* that have size linear in the number of qubits. Of course, these matrices are ideal to be represented by *ADDs*, since they have many common sub-matrices and only a few different elements, that is, only a few terminal vertices for their *ADDs*. Still, they affirm the belief that *ADD* is a good datastructure for simulating quantum computers. Results from simulating Grover's algorithm using *ADDs* [25] have also boosted to this direction. In the next chapter we describe the results of simulating Shor's algorithm using *ADDs*.

Chapter 5

Simulating Shor's Algorithm Using *ADDs*

In this chapter we describe the details of our implementation for simulating Shor's algorithm. Then we present some experimental results and discuss conclusions and ideas for future works.

5.1 Implementing *ADDs*

There are a few *BDD* or *ADD* packages which provide an interface to manipulated *BDDs*. The most advanced is maybe *CUDD* [31] which is used by *QuIDDPro* [24, 25]. These packages can reduce the implementation overhead, since they have implemented the algorithms described in chapter 4. However, they are not specialized in quantum computer simulation, so we have to change some of their features. For example, *CUDD* does not support complex values in the terminal vertices. *QuIDDPro* faces this problem by storing the complex values in a table and keeping the table indexes of the appropriate complex values in the terminal vertices. We decided to implement everything from scratch, because we wanted to specialize our code for quantum computer simulation.

Furthermore, this approach gave as the opportunity to deepen into the properties of *ADDs* and their relation to quantum computers.

Our simulator is written in C++ and includes two major classes. One represents a vertex of the *ADD* and the other represents an *ADD*. Following Bryant's [28] suggestion for what a vertex structure should store, our vertex class has the following members

	Type	Comments
index	unsigned short	variable index
low	unsigned long	low child
high	unsigned long	high child
id	unsigned long	unique identifier
mark	boolean	mark to avoid visiting the vertex more than once in a traversal
val	complex	value of terminal vertices

Table 5.1: Member variables of Vertex class

First, we have to say that each *ADD* keeps its vertices in a table of vertices. Thus, a vertex can store the table indexes of its low and high children rather than keeping pointers to them. The *id* variable is unique for each vertex and it is mainly used by *REDUCE*. *Mark* is used when traversing a graph, to avoid visiting the same vertices more than once by marking them as true. *Val* stores the complex value of a terminal vertex and a trivial value for non-terminal vertices. Last but not least, the *index* variable indicates the variable's index. Indexing starts from 1 and following the interleaved variable ordering, we have the correspondence

dex	1	2	3	4	$2n$ 1
orresponding w or column variable	R_0	C_0	R_1	C_1	R_n

Table 5.2: variable – index correspondence

An n qubit operator is described by a 2^n x 2^n matrix which needs n row and n column variables to be indexed using an *ADD*, a total of $2n$ variables. This means that an unsigned short integer is enough to store the *index*. A way to distinguish terminal vertices is to store k+1 as their *index*, where $k = 2n$ is the total number of variables. It is possible to have two different classes for vertices, one for terminals and one for non-terminals. This approach may save a little space since non-terminal vertices should not have a value and terminals should not have children, but it would add implementation overhead. In addition, the results indicate that the memory saving would be minor.

The second class implements an *ADD* and includes the following member variables to manipulate it

	Type	Comments
root	unsigned long	table index of the root vertex
number_of_vertices	unsigned long	number of vertices
number_of_variables	unsigned short	total number of variables
number_of_row_vars	unsigned short	number of row variables
number_of_terminals	unsigned long	number of terminal vertices
table_size	unsigned long	size of the table that stores the vertices
table	vertex *	table where the vertices are stored

Table 5.3: Member variables of ADD class

The pointer *table* points to the first element of the table where vertices are stored. This table has available space for *table_size* vertices, which is limited to unsigned long (2^{32}-1) vertices. This size is big enough, since the aforementioned algorithms for an *ADD* with more than 2^{32}-1 elements would be extremely inefficient. The memory for the table is allocated using *malloc* and then increased or decreased using *realloc*. An *ADD* has *number_of_vertices* vertices which cannot be greater than *table_size*. Variable *root* stores the table index, where the root vertex is stored. Finally, the variable *number_of_variables* stores the total number of variables and *number_of_row_vars* stores the number of row variables. These two variables help us to determine the type of matrix which is represented by this *ADD*, i.e. row vector, column vector or matrix. Suppose that

N = *number_of_variables*/2 and
X = *number_of_row_vars*

then we have the following cases

	number_of_row_vars = 0	*number_of_row_vars* ≠ 0
odd *number_of_variables*	-	column vector (2^X x 1)
even *number_of_variables*	row vector (1 x 2^N)	matrix (2^N x 2^N)

Table 5.4: Finding the type of the matrix which is represented by the ADD class

Our code creates some important matrices, like the *Hadamard* gate, *Not* gate, identity matrix, state vectors $|0\rangle$, $|1\rangle$ and the like. For example, our implementation stores the state vector $|0\rangle$ as

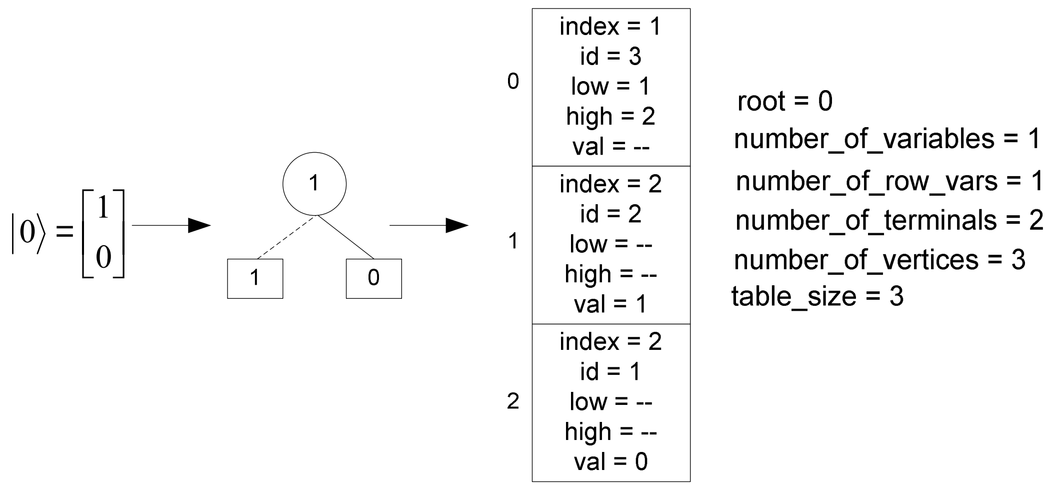

Figure 5.1: Using the ADD class to represent the state vector $|0\rangle$

5.2 Simulating Arbitrary Quantum Circuits

We have implemented all the algorithms described in chapter 4 as member functions of the *ADD* class, together with some extra functions that build special vectors or matrices. Although our code targets Shor's algorithm, it can be used for simulating arbitrary quantum circuits. To give a brief description of some member functions that can be used for quantum circuit simulation, suppose that X, I are the names of the classes representing the *Not* gate, and the identity matrix. For each function we provide a short example.

-*apply*(*ADD * add1, ADD * add2, const char op*)

```
ADD result;
result.apply(&I, &X, '+');
// result becomes the ADD which represents the matrix I + X
```

-*tensor*(*ADD *add1, ADD *add2*)

```
ADD result;
result.tensor(&I, &X);
// result becomes the ADD which represents the matrix I ⊗ X
```

-*mult*(*ADD *add1, ADD *add2*)

```
ADD result;
result.mult(&I, &X);
```

```
        // result becomes the ADD which represents the matrix $I \cdot X$
```

-*copy*(*ADD *add1*)

```
        ADD cp;
        cp.copy(&X);
        // cp becomes the same ADD as the ADD X
```

-*state*(*unsigned long i, unsigned short number_of_qubits*)

```
        ADD sv;
        sv.state(3, 7);
        // sv becomes the ADD which represents the state vector $|0\rangle|0\rangle|0\rangle|0\rangle|0\rangle|1\rangle|1\rangle = |3\rangle$
```

-*identity*(*unsigned short number_of_qubits*)

```
        ADD ident;
        ident.identity(10);
        // ident becomes the ADD which represents the matrix $I^{\otimes 10}$
```

-*hadamard*(*unsigned short number_of_qubits*)

```
        ADD had;
        had.hadamard(5);
        // had becomes the ADD which represents the matrix $H^{\otimes 5}$
```

-*vector_adjoint*()

```
        ADD vec;
        vec.state(0,1);
        vec.vector_adjoint();
        // vec becomes the ADD which represents the matrix $\langle 0|$
```

-*print_matrix*()

```
        X.print_matrix();
        // it will print  0 1
        //                1 0
```

-*measure_one_qubit*(*unsigned short qubit*)

```
        ADD sv;
        sv.state(0, 5);
        int k;
        k = sv.measure_one_qubit(3);
        // k has the outcome (0 or 1) of measuring the fourth qubit of the state vector
        // represented by the ADD sv. The most significant qubit is the qubit 0.
```

-*measure_qubits*(*unsigned short from, unsigned short to*)

```
ADD sv;
sv.state(0, 10);
double k;
k = sv.measure_qubits(3, 6);
// k has the outcome of measuring qubits 3, 4, 5, 6 of the state vector
// represented by the ADD sv. Notice that if we measure more than 32 qubits
// then the result might be too big to be stored in an integer.
```

-*CU*(*unsigned short c, unsigned short t, ADD *U, unsigned short number_of_qubits*)*;*

```
ADD ctrl_not;
ctrl_not.CU(2, 4, &X, 10);
// The ADD ctrl_not represents the controlled-NOT gate for a ten qubit circuit
// where the control qubits is qubit #2 and the target qubits is qubit #4.
```

-*CCU*(*unsigned short c1, unsigned short c2, unsigned short t, ADD *U, unsigned short number_of_qubits*)

```
ADD ctrl2_not;
ctrl2_not.CCU(1, 2, 4, &X, 10);
// The ADD ctrl2_not represents the controlled-NOT gate for a ten qubit circuit
// where the control qubits are qubits #1 and #2, and the target qubits is qubit #4.
```

-*Swap*(*unsigned short qubit1, unsigned short qubit2, unsigned short number_of_qubits*)

```
ADD swap_gate;
swap .Swap(3, 5, 10);
// The ADD swap_gate represents the Swap gate for a ten qubit circuit which
// swaps qubits #3 and #5
```

We can mention a few details about how these functions are implemented. The algorithms for the first three functions were described in chapter 4. Functions *state*, *identity* and *hadamard* produce their result by repeatedly tensoring $|i\rangle$, *I* or *H*. We can do a trick to make these computations faster. Since $I^{\otimes n} = I^{\otimes n/2} \otimes I^{\otimes n/2}$ and $H^{\otimes n} = H^{\otimes n/2} \otimes H^{\otimes n/2}$, we can compute half of the tensor products and then tensor the result by itself. For large *n* we may partition the computation to four or even more steps.

Having an *ADD* representing a state vector $|\psi\rangle$, we can easily convert it to an *ADD* representing its dual (adjoint) $\langle\psi|$. This can be done by changing the row variables to column variables and complex conjugating the values of the terminal vertices.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \langle 0| = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

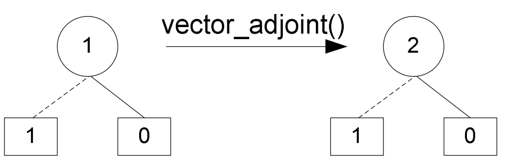

Figure 5.2: Using vector_adjoint on an ADD representing $|0\rangle$

Measuring a qubit seems to be more interesting. Suppose that we have a quantum register in the state

$$|\psi\rangle = \underbrace{|x\rangle}_{\substack{p \\ qubits}} \overbrace{|i\rangle}^{\substack{1 \\ qubit}} \underbrace{|z\rangle}_{\substack{q \\ qubits}} \tag{5.1}$$

The probability of getting 0 when we observe register $|i\rangle$ is given by the equation (1.49) and it is

$$p(0) = \langle\psi|M^{\dagger}_{p0q} M_{p0q}|\psi\rangle \tag{5.2}$$

where

$$M_{p0q} = I^{\otimes p} \otimes |0\rangle\langle 0| \otimes I^{\otimes q} = I^{\otimes p} \otimes M_0 \otimes I^{\otimes q} \tag{5.3}$$

Our code can easily create the identity matrices of equation (5.3) using function *identity* and then tensor them with the built-in *ADD* that represents the measurement operator M_0. Since both M_0 and $I^{\otimes k}$ are diagonal matrices, their tensor product is also a diagonal matrix. Moreover, the only nonzero element of matrix (5.3) is 1, which is not a complex number. That means that the adjoint of matrix (5.3) is equal to itself, and their product is still matrix (5.3)

$$M^{\dagger}_{p0q} = M_{p0q} \tag{5.4}$$

$$M^{\dagger}_{p0q} \cdot M_{p0q} = M_{p0q} \tag{5.5}$$

We can rewrite (5.2) using (5.5)

$$p(0) = \langle\psi|M_{p0q}|\psi\rangle \tag{5.6}$$

Equation (5.6) is easily computed using the functions described above, since we can use *identity* and *tensor* to create the measurement operator, then we multiply it with the state vector and finally we multiply the result with the adjoint of the state vector which is taken by *vector_adjoint*. Having calculated the probability to measure 0, we call a random number generator to take a value in the range [0, 1]. If this value is less than or equal to *p*(0) we consider that the measurement outcome is 0, else it is 1. Next, we have to create the post-measurement state which is given by (1.50). If the measurement's outcome was 0 then the post-measurement state is

$$\frac{M_{p0q}|\psi\rangle}{\sqrt{p(0)}} \tag{5.7}$$

The numerator has already been computed as part of equation (5.6), so the only thing needed for (5.7) is to divide the terminal vertices of the *ADD* representing the numerator with the square root of the probability $p(0)$. If the measurement outcome was 0 then the post-measurement state is

$$\frac{M_{p1q}|\psi\rangle}{\sqrt{p(1)}} = \frac{M_{p1q}|\psi\rangle}{\sqrt{1-p(0)}} \tag{5.8}$$

which means that we have to create the measurement operator M_{p1q}, multiply the state vector with it, and then divide with the square root of the probability. To measure multiple qubits we can measure each qubit separately and then sum the appropriate powers of 2, depending on the measurement outcomes.

Finally, we have to describe how we implement the controlled gates. If we have an *ADD* that represents the gate *U*, then the overall gate of *Figure 5.3* can be built by the following code

```
ADD gate;
gate.CU(k, k + 1 + n, &U, k + 1 + n + p + q)
```

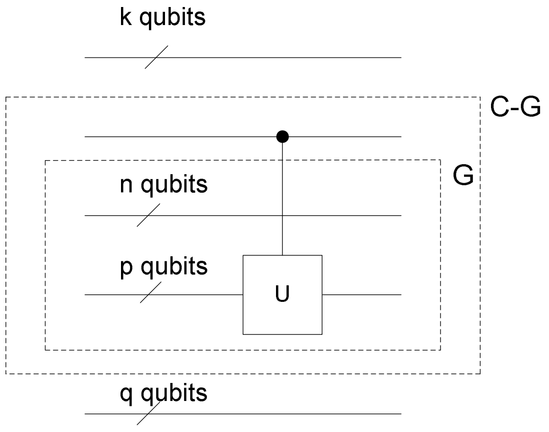

Figure 5.3: Creating the matrix for a controlled gate

The first step is to build gate *G* which is fairly easy

$$G = I^{\otimes n} \otimes U \tag{5.9}$$

Following, we have to create the *ADD* for the *controlled-G* gate *CG*. This gate is describe by the matrix

$$CG = \begin{bmatrix} I^{\otimes(n+p)} & 0 \\ 0 & G \end{bmatrix} \tag{5.10}$$

Having the *ADDs* for $I^{\otimes(n+p)}$ and *G* we can construct the *ADD* for matrix (5.10)

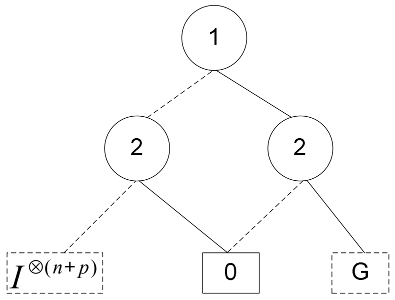

Figure 5.4: Unreduced ADD representing matrix (5.10)

The *ADDs* for $I^{\otimes(n+p)}$ and *G* in *Figure 5.4* have to be variable-shifted, that is, we have to add 2 in each variable index. The last step for creating the overall gate is trivial

$$overall = I^{\otimes k} \otimes CG \otimes I^{\otimes q} \tag{5.11}$$

The same ideas can be used to create *ADDs* that represent controlled gates with multiple control qubits. Furthermore, we can create the *Swap* gate using three *controlled-NOT* gates as shown in *Figure 1.4*.

5.3 Simulating Shor's Algorithm

We have already described some features of our implementation that can be used for simulating arbitrary quantum circuits. However, our main goal is to discover how efficiently these features can be used to simulate Shor's algorithm. Shor's algorithm includes two main parts. The first is the quantum circuit for modular exponentiation and the other is the quantum Fourier transform. In *section 2.5* we gave a brief description of how we can implement the quantum circuit for modular multiplication. Beckman et al [13] present in full detail how to build such a circuit. Just think that we have a classical circuit that calculates the modular multiplication. This circuit is built with *AND*, *OR*, *NOT*, *NAND* and *XOR* gate. In *section 1.7* we showed that we can construct an equivalent quantum circuit by replacing these gates with only *controlled-NOT* gates. This means that our quantum circuit for modular exponentiation is implemented using only controlled not gates. It turns out that these controlled not gates may have up to four control qubits. We described the process of creating a controlled gate in the previous section. The modular multiplication circuit uses thousands of such gates, so their construction time would add a serious time overhead. Fortunately, the number of different *controlled-NOT* gates that are used is much smaller. Thus, when we create such a gate we store it, just in case it is used in the future. The disadvantage is that we need memory to store these gates, but hopefully the amount of memory used is immaterial when compared to the speedup that it offers.

Concerning the *QFT*, we inherited the semi-classical approach that was presented in *section 2.6*. The only point that is worth mentioning is that we replaced the chains of *rotation* matrices with just one matrix. Recall that a rotation matrix has the form

$$\begin{bmatrix} 1 & 0 \\ 0 & e^{a} \end{bmatrix} \tag{5.12}$$

Then a series of rotation matrix multiplications can be replaced by a single rotation matrix

$$\begin{bmatrix} 1 & 0 \\ 0 & e^{a} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & e^{b} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & e^{c} \end{bmatrix}\begin{bmatrix} 1 & 0 \\ 0 & e^{d} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{a+b+c+d} \end{bmatrix} \tag{5.13}$$

That is, instead of multiplying the control qubit of the *QFT* with several rotation matrices in a row, we replace them with just one rotation matrix which performs the same rotation.

The last thing that needs to be explored is the order of multiplications. To give a sense of this problem, consider that we want to apply the *Hadamard* and the *NOT* gate to a single qubit which is in the state $|\psi\rangle$. The resulting state is

$$XH|\psi\rangle \tag{5.14}$$

A simulator can perform the multiplications of (5.14) in two different ways

$$XH|\psi\rangle = X\big(H|\psi\rangle\big) = \big(XH\big)|\psi\rangle \tag{5.15}$$

The simulation of Shor's algorithm includes thousands of matrix multiplications, which entails a huge number of possible multiplication orderings. It turns out that the order of multiplications is very important for the efficiency of the simulator. For example, we may multiply many *controlled-Not* matrices to create the modulo multiplier *OMULN* (*section 2.5*) which is used in modular exponentiation, and then multiply it with the state vector. This approach proved extremely inefficient. Thus, we focused on multiplying the state vector with smaller modules, like the modular adder *OADDN*, which was much faster. We continued the process of creating smaller modules and then multiplying them with the state vector until we reached the point where each module was a single *controlled-NOT* gate. Multiplying the state vector with every single gate turned to be faster and less memory consuming than creating any intermediate matrix and then multiplying it with the state vector.

5.4 Experimental Results

We have simulated Shor's algorithm to factor various integers. Here we present how fast these simulations ran and how much memory they consumed. In order to judge the efficiency of our simulator we also present the same results for *QCL* Recall that Shor's algorithm has to choose a random number *a* co-prime to the number *N* that we want to factor. For each pair of such numbers *a* and *N* the algorithm computes the *order r* of *a* modulo *N*, which is the period of the function $f(x) = a^{x} \bmod N$. The following results were taken on a PC with an AMD Athlon XP 2400+ CPU and 512 MB RAM, running Linux. The C++ compiler was g++ 3.3.1

N	# bits of N	a	r	T_{QCL} (sec)	T_{ADDs} (sec)	MEM_{QCL} (MByte)	MEM_{ADDs} (MByte)
$3 \cdot 5 = 15$	4	13	4	0.007	0.5	0.95	1.6
$3 \cdot 7 = 21$	5	2	6	0.025	2	1	2.1
$3 \cdot 11 = 33$	6	17	10	0.17	6	1.2	2.9
$7 \cdot 11 = 77$	7	9	15	0.6	27	1.9	4.1
$11 \cdot 13 = 143$	8	61	30	2.3	52	4.9	5.7
$13 \cdot 23 = 299$	9	226	44	12	72	16.9	7.8
$23 \cdot 31 = 713$	10	687	66	75	307	64.9	10.8
$31 \cdot 37 = 1147$	11	662	90	-	648	-	14.2
$37 \cdot 61 = 2257$	12	858	90	-	985	-	18.2
$71 \cdot 73 = 5183$	13	1298	126	-	2033	-	22.7
$83 \cdot 127 = 10541$	14	995	126	-	3104	-	28.6
$131 \cdot 139 = 18209$	15	972	130	-	4085	-	35.1
$191 \cdot 241 = 46031$	16	3132	190	-	8551	-	42.7

Table 5.5: Indicative time and memory required to factor numbers of various bit-lengths using QCL and ADDs

Notice that *QCL* was unable to factor integers of 11 bits or more, because it ran out of memory. Therefore, we can compare it with our simulator only for relatively small inputs. As expected, *QCL* is always faster than *ADDs* because it uses a faster datastructure. However, as the input becomes larger, the gap between the two simulators becomes smaller. That is, while *QCL* is about 70 times faster than *ADDs* when factoring a 5-bit number, it is just 4 times faster when factoring a 10-bit number. The following plots help us to visualize the results. Unfortunately, the results for *QCL* are limited to factoring up to 10-bit integers.

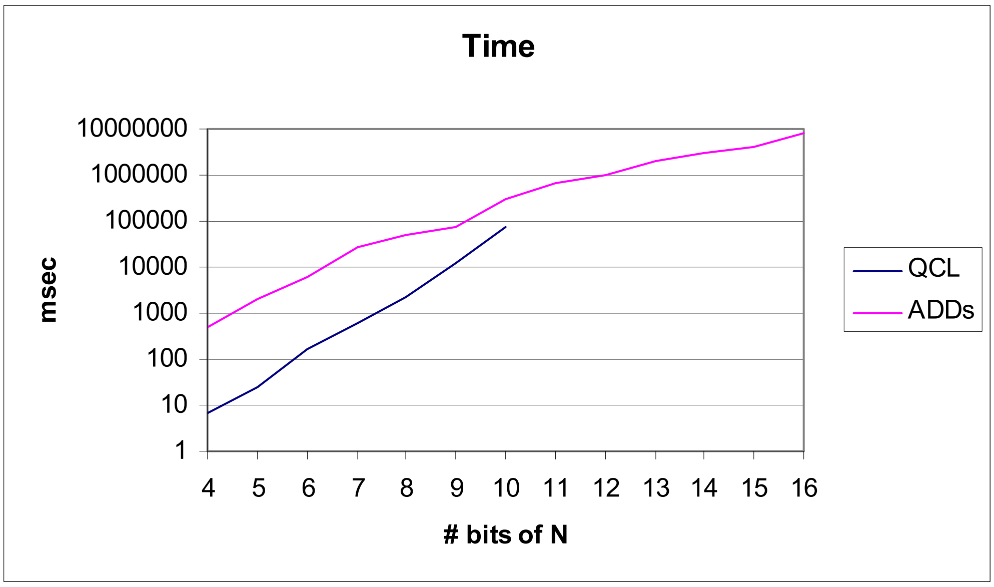

Figure 5.5: The time in msec that was required to run Shor's algorithm

Judging from *Figure 5.5*, the simulation of factoring integers which are longer than 16-bits will take several hours. However, such a simulation seems to be possible with *ADDs*, since they keep memory usage in a very low level.

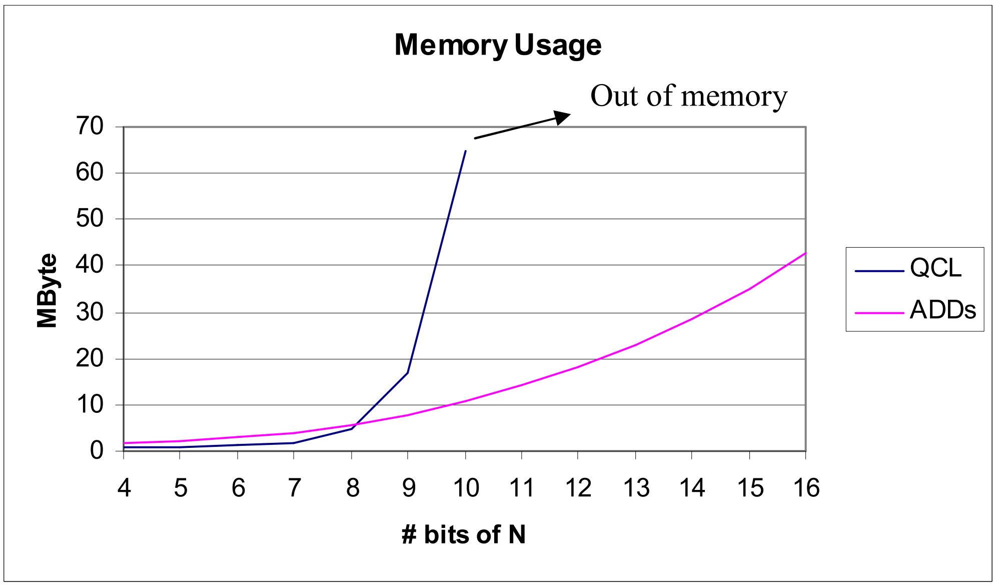

Figure 5.6: The amount of memory that was required to run Shor's algorithm

While *QCL* demands excessive amounts of memory when factoring an integer of 11 bits or more, *ADDs* require an amount of memory that is almost linear in the bit-length of the number to be factored.
Our simulator applies each gate directly to the state vector without keeping any intermediate results. The next table shows the number of vertices of the *ADDs* which stored the state vector at the end of each simulation, together with the total number of matrix multiplications (operations) that were performed

N	# bits of N	# qubits used by Shor's algor.	*a*	*r*	Max vertices for the *ADD* of the state vector	# operations
$3\cdot 5=15$	4	14	13	4	31	3880
$3\cdot 7=21$	5	17	2	6	90	8415
$3\cdot 11=33$	6	20	17	10	169	14530
$7\cdot 11=77$	7	23	9	15	288	24711
$11\cdot 13=143$	8	26	61	30	636	37046
$13\cdot 23=299$	9	29	226	44	1037	54963
$23\cdot 31=713$	10	32	687	66	1727	75524
$31\cdot 37=1147$	11	35	662	90	2524	100600
$37\cdot 61=2257$	12	38	858	90	2805	131200
$71\cdot 73=5183$	13	41	1298	126	4241	165920
$83\cdot 127=10541$	14	44	995	126	4572	216465
$131\cdot 139=18209$	15	47	972	130	5048	261274
$191\cdot 241=46031$	16	50	3132	190	7862	320955

Table 5.6: Number of vertices for the state vector at the of each simulation and the total number of operations

It is important to repeat that using the ideas described in chapter 2, our simulator uses 3*L*+2 qubits to factor an *L*-bit integer. That is, we have a state vector of 3*L*+2 qubits, which is a vector with 2^{3L+2} complex values. Thus, the number of vertices for the state vector remains relatively low even for large inputs.

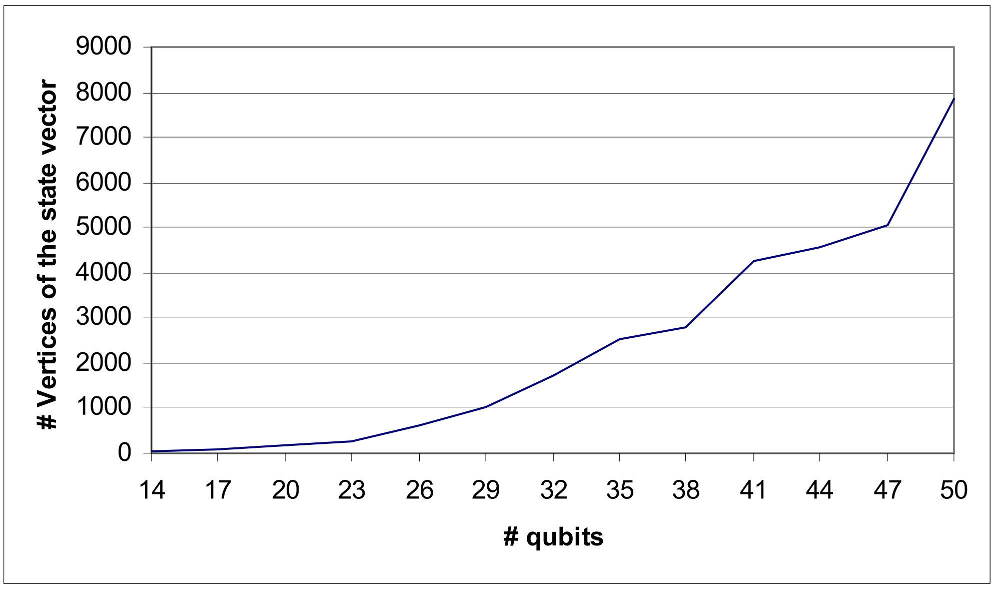

Figure 5.7: Number of vertices of the ADDs storing the state vectors at the end of simulation

A vertex class needs 36 bytes to be stored; therefore it is easy to see how much memory is used by the state vector and the *controlled-NOT* cache. For instance, in the case of factoring 5183 the state vector is stored in approximately $4241 \cdot 36 \approx 153$ Kbytes. During the computations, the size of the state vector may be a little bigger, but not more than a few hundred extra vertices. That means that the rest 22 MB that are required by the simulator are mainly used to store *controlled-NOT* gates. Thus, we can greatly reduce the memory usage of our simulator by not caching the *controlled-NOT* gates of the modular exponentiation circuit, but this would be significantly slower.

Table 5.6 also shows the total number of operations that were performed by our simulator. The major part of these operations is produced by the modular exponentiation circuit, which is the bottleneck of the simulation. Unfortunately, if we want to simulate everything that a quantum computer would do to simulate Shor's algorithm, we have to apply all these gates.

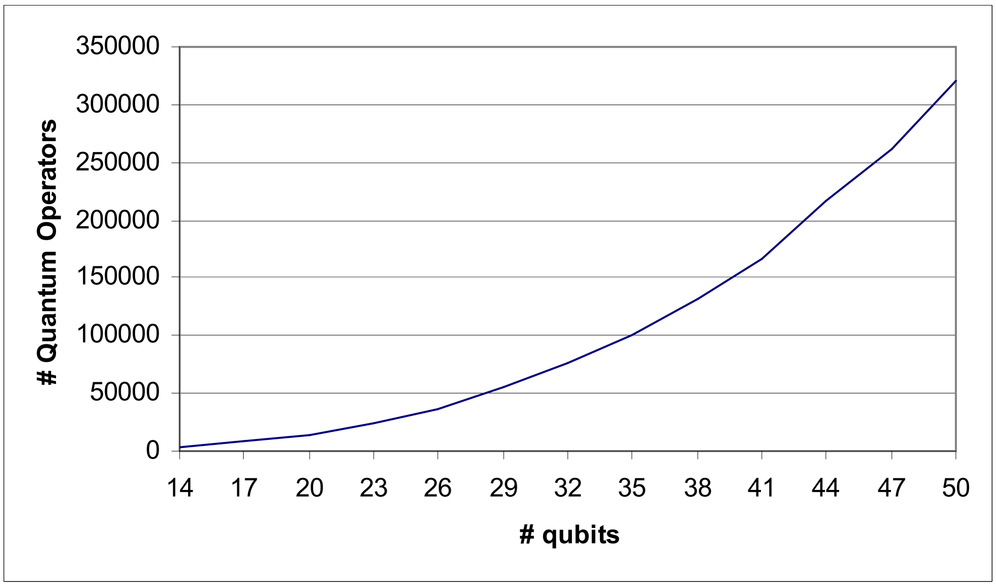

Figure 5.8: Number of quantum operators applied to the state vector

It is maybe more interesting to see how fast these operations are performed.

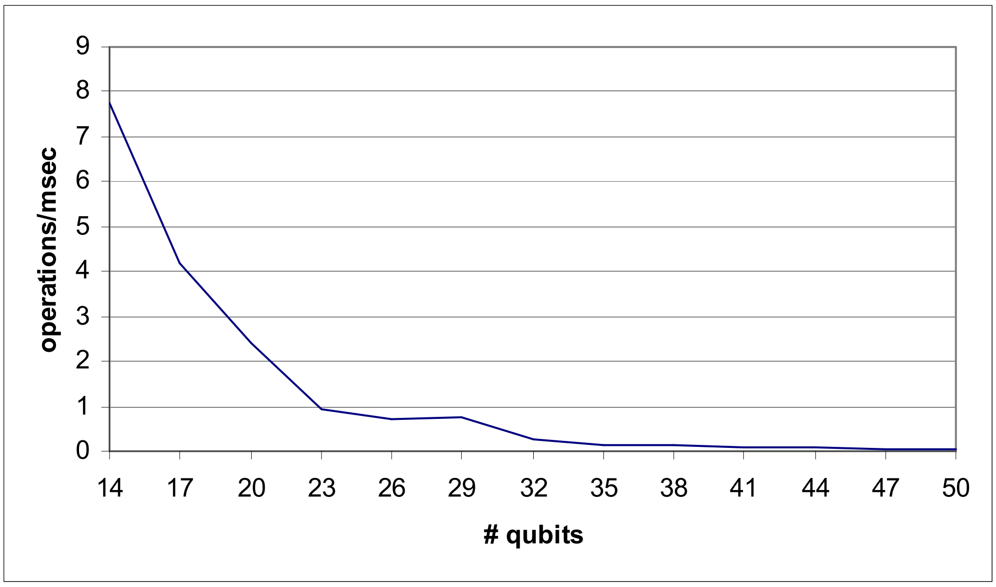

Figure 5.9: Speed of operations

We see that the speed of matrix multiplications is reduced rapidly as the input becomes bigger. This is more or less expected, since a large number to be factored requires a bigger *ADD* for the state vector and bigger matrices to represent gates. Furthermore, the efficiency of matrix multiplication using *ADDs* is greatly affected by the size of the operand *ADDs*. This also explains the following two figures, which show that starting with a small state vector's *ADD*, the first 30% of matrix multiplications are quit fast, but gradually this *ADD* becomes big and the performance of matrix multiplication is reduced.

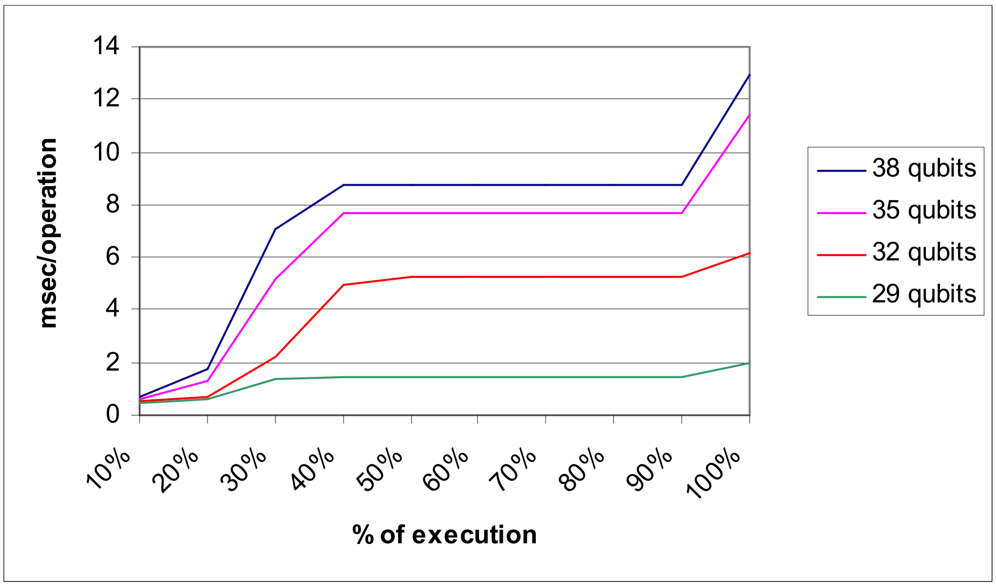

Figure 5.10: Cost of operations vs % of execution during the simulation

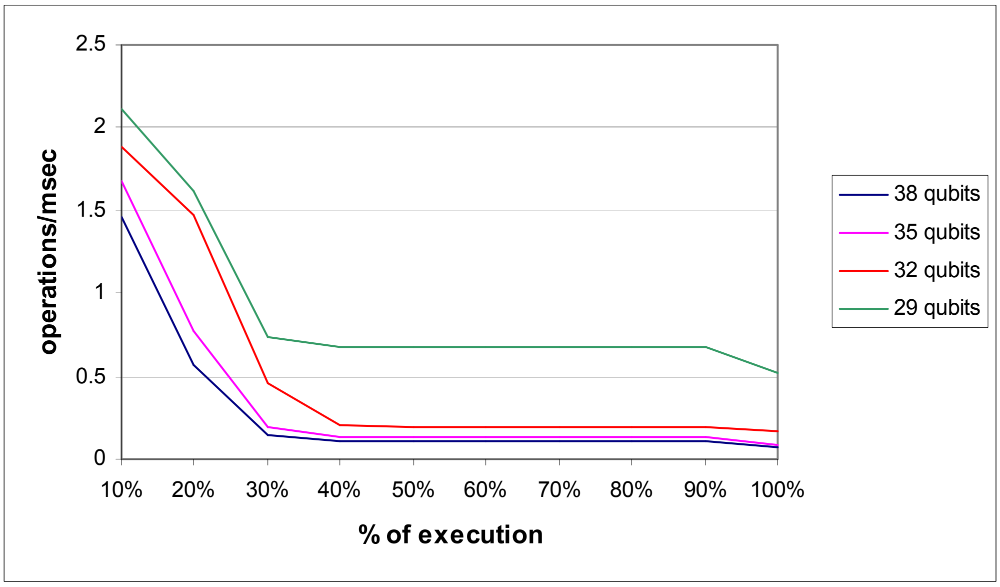

Figure 5.11: Speed of operations during the simulation

The last thing that is worth mentioning is that the runtime of simulation is dramatically affected by the *order r* of *a* modulo *N*. The next table shows the results of simulating the factoring of 1147 (11 bits) for various random *a*.

a	*r*	T (sec)	MEM (MByte)	Max vertices for the *ADD* of the state vector
714	6	56	12.3	193
1048	9	140	13.1	289
192	18	150	13.5	548

861	30	210	13.5	898
623	60	250	13.7	1713
662	90	648	14.2	2524
351	180	700	14.4	4864

Table 5.7: Using the simulator to factor 1147 with various random numbers a

We see that smaller *orders* lead to a smaller *ADD* for the state vector which entails smaller execution times. The algorithm calculates the function $f(x) = a^x \bmod N$ for all *x*, and stores the results in a superposition in the state vector. Thus, a smaller *order* means less possible outcomes for *f*(*x*), which is translated in less nonzero elements in the state vector. The number *a* is chosen randomly and its *order* is what we are looking for, so we can't know a priori if it is a "good" *a*. However, a bad selection means slower simulation, but not excessive memory usage, so we can be sure that the simulation will end normally.

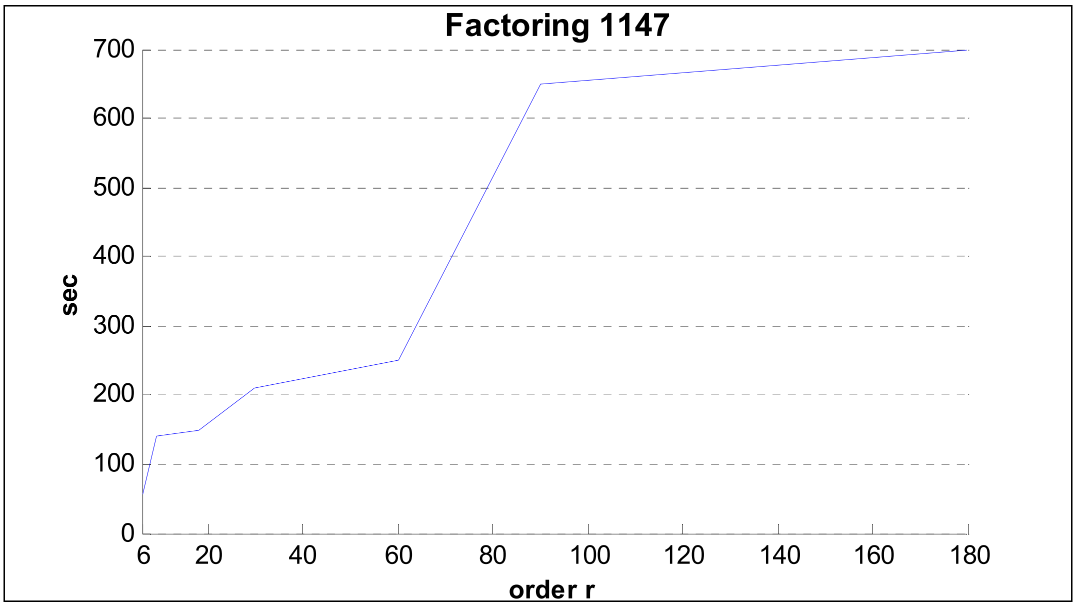

Figure 5.12: Time to factor 1147 vs order of a modulo N

Chapter 6

Conclusions and Future Work

6.1 Overview of the work presented

We described how *ADDs* can be used to simulate quantum computers and focused on simulating Shor's algorithm. We present the results of factoring up to 16-bit integers using 50 qubits which is impossible for other simulators. Of course, our simulator can factor even bigger integers, since *ADDs* achieve a great compression for both state vectors and operators, which helps us to simulate circuits with many qubits. Execution time is still a problem, but we have to accept that it is not the easiest thing in the world to simulate a system with huge state vectors. Furthermore, someone who wants to simulate an important quantum circuit may have no problem to wait for several hours, even days, but he would surely have a problem if he cannot simulate the circuit at all.

6.2 Immediate work ahead

As previously mentioned, we implemented the simulator from scratch without using any *BDD* package or library. Thus, a little more code tuning may be needed to improve several parts of the code. Although we do not expect miracles, it is possible to achieve a small time improvement. We can also change some things concerning the *ADDs* themselves, i.e. we can choose and test a different variable ordering. Our main purpose is to make the *REDUCE* and matrix multiplication algorithms faster. While simulating Shor's algorithm, our simulator spends most of its time running these two algorithms. For the matrix multiplication algorithm we can test various hash table techniques and sizes.

An interesting idea is to find a way to create the *ADD* for the state vector $\frac{1}{\sqrt{M}}\sum_{x=0}^{M-1}|x\rangle|a^x \bmod N\rangle$ without using the modular exponentiation circuit, which is responsible for the huge number of operations. Then, we can apply the semi-classical QFT to this state vector to get the final result. In this way, we simulate a quantum algorithm rather than a quantum circuit. This is desirable in many situations, when we want to test a new quantum algorithm, or a new idea, but we just don't care yet for the circuit which implements it.

Finally, if our simulator is to be used for simulating arbitrary quantum circuits, it would be wise to supply it with a friendlier interface. One approach is to implement a command line interface. Another approach is to implement a GUI. A command line interface is usually more flexible than a GUI, but the later helps us to visualize a quantum circuit. In the next section we discuss one more option for providing an interface for the simulator.

6.3 Quantum programming languages

It is obvious that quantum hardware would be useless without a way to program it. Although we have no such hardware available yet, we do have the experience of programming a classical computer. Based on this experience several people try to implement quantum programming languages, which incorporate classical ideas together with quantum properties. Several compilers, interpreters and libraries for quantum programming exist, but no quantum computer is available to test them. Since quantum programming is a new field, we need ways to use quantum languages in order to understand and develop their unique features. The solution is to use a simulator of a quantum computer, which runs the quantum program. Therefore, instead of creating a new interface for our simulator, we can integrate the *ADD* representation into a quantum programming language. This way, any algorithms implemented for this quantum language will run without any change using our simulator, and users won't have to learn one more tool. Furthermore, it will be possible to use the quantum language to run algorithms with many qubits.

6.4 Improving *ADDs*

It has been showed that *ADD* is an efficient datastructure for simulating quantum computers. However, to make simulation more efficient we have to introduce a variant of *ADD* that is designed explicitly for quantum simulation. One idea is to increase the number of children for internal vertices. Moreover, we may change terminal vertices to store not only complex values, but something more “interesting”. For example, they may store a pointer to another *ADD* together with a complex number, which implements the tensor product. It is also worth the effort to explore ways for parallelizing the simulation, to make it faster when using many processors. To achieve this, we can change the algorithms described in chapter 4, but we can also change the nature of *ADDs* to favor parallel processing. It is not sure if any of these approaches can work efficiently, but results prompt us to specialize *ADDs* even more for quantum simulation.

References

[1] Feynman R. (1982)
Simulating physics with computers
International Journal of Theoretical Physics 21, 6&7, 467-488

[2] Shor P.W. (1994)
Algorithms for quantum computation: discrete log and factoring
Proceedings of the 35th Annual IEEE Symposium on Foundations of Com. Science

[3] Grover L. K. (1996)
A fast quantum mechanical algorithm for database search
In Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (Philadelphia, Pennsylvania, 22-24 May 1996), pp. 212-219

[4] Dirac P. (1958)
The Principles of Quantum Mechanics (*4th ed.*)
Oxford University Press

[5] Wootters W.K. and Zurek W.H (1982)
A single quantum cannot be cloned
Nature 299, 802

[6] Bennet C.H. (1973)
IBM J. Res. Develop. 17,525

[7] Bennet C.H. (1989)
SIAM J. Comput. 18, 766

[8] Josef Gruska (1999)
Quantum Computing
McGraw-Hill

[9] Nielsen M.A. and Chuang I.L. (2000)
Quantum Computation and Quantum Information
Cambridge University Press

[10] Rieffel E. and Polak W. (2000)
An Introduction to Quantum Computing for Non-Physicists
ACM Computing Surveys, Vol. 32, No. 3, September 2000, pp. 300-335

[11] Hirvensalo M. (2001)
Quantum Computing
Natural Computing Series – Springer

[12] Knuth D. (1981)
The Art of Computer Programming, Vol. 2: Seminumerical Algorithms
Addison-Wesley

[13] Beckman D. et al (1996)
Efficient networks for quantum factoring
Phys. Rev. A 54, pp. 1034-1063
arXiv:quant-ph/9602016

[14] Griffiths R.B. and Niu C.-S (1995)
Semi-classical Fourier Transform for Quantum Computation
Phys. Rev. Lett. 76 (1996) 3228-3231
arXiv:quant-ph/9511007

[15] Mosca M. and Ekert A. (1998)
The Hidden Subgroup Problem and Eigenvalue Estimation on a Quantum Computer
Proceedings of the 1st NASA International Conference on Quantum Computing and Quantum Communication, Palm Springs, USA, Lecture Notes in Computer Science 1509 (1999), 174-188.
arXiv:quant-ph/9903071

[16] Wallace J. (2001)
Quantum Computer Simulators
Partial Proc. of the 4th Int. Conference CASYS 2000, D. M. Dubois (Ed.), International
Journal of Computing Anticipatory Systems, volume 10, 2001, pp. 230-245

[17] Wallace J. (2002)
Quantum Computer Simulators
http://www.dcs.ex.ac.uk/~jwallace/simtable.htm

[18] Omer B. (2003)
Structured quantum programming
http://tph.tuwien.ac.at/~oemer

[19] Omer B. (1996)
Simulation of Quantum Computers
http://tph.tuwien.ac.at/~oemer/papers.html

[20] Bettelli S. (2003)
Toward an architecture for quantum programming
arXiv:quant-ph/0103009

[21] Zuliani P. (2001)
Quantum Programming PhD thesis, University of Oxford,
http://web.comlab.ox.ac.uk/oucl/work/paolo.zuliani/pzthesis.ps.gz

[22] Blaha S. (2002)
Quantum Computers and Quantum Computer Languages: Quantum Assembly Language and Quantum C Language
arXiv:quant-ph/0201082

[23] National Institute of Standards and Technology (2003)
QCSim
http://hissa.nist.gov/~black/Quantum/qcsim.html

[24] Viamontes G.F., Rajagopalan M, Markov I.L. and Hayes J.P. (2002)
Gate-Level Simulation of Quantum Circuits
arXiv:quant-ph/0208003

[25] Viamontes G.F., Markov I.L. and Hayes J.P. (2003)
Improving Gate-Level Simulation of Quantum Circuits
arXiv:quant-ph/0309060

[26] Lee C.Y (1959)
Representation of Switching Circuits by Binary Decision Programs
Bell System Technical Journal, Vol 38, pp. 985-999

[27] Akers S.B. (1978)
Binary Decision Diagrams
IEEE Transactions on Computers, Vol. C-27, No. 6, pp. 509-516

[28] Bryant R. (1986)
Graph-Based Algorithms for Boolean Function Manipulation
IEEE Transactions on Computers, Vol. C-35-8, pp 677-691

[29] Bahar R.I. et al (1997)
Algebraic Decision Diagrams and their Applications
Journal of Formal Methods in System Design, Vol. 10

[30] Clarke E., Fujita M., McGeer P.C., McMillan K., Yang J.
Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation
IWLS '93, pp.6a1-15

[31] Somenzi F. (1998)
CUDD: CU Decision Diagram Package, ver. 2.3.0
University of Colorado at Boulder

List of Tables

List of Figures