CS297 Report

Quantum Threshold Gate Simulator

Xin Chen
(xinchen_2002@yahoo.com)

Advisor: Dr. Chris Pollett

December 2002

Table of Contents

List of Figures

List of Tables

1. Introduction

In recent years interest in quantum computation has been steadily increasing. One reason for this is due to Shor's [S97] discovery of a polynomial time quantum algorithm for factoring, which is one of the strongest arguments in favor of the superiority of quantum computing models over classical ones. Since this discovery, many efforts have been made to find new, efficient quantum algorithms for classical problems and to develop quantum complexity theory. The goal of this research will be to develop a simulator which will aid in understanding the robustness of certain quantum algorithms.

The first such algorithm to be considered is described in [S02] which gives a way to simulate threshold circuits with small depth quantum circuits. It is well-known that threshold circuits have been found to be surprisingly powerful. For example, Beame et al. (1986) gave an algorithm to compute iterated multiplication using threshold circuits in constant depth.

The simulation is an improvement over what can be done with classical AND, OR, NOT circuits. Unfortunately, the acceptance model for these quantum circuits is unrealistic. We say a 1 is output if the quantum circuit accepts with a non-zero probability; otherwise, we say a 0 is output. So the question is how much error is introduced if we choose a more realistic acceptance criteria? Good formal estimates of this are somewhat difficult to directly calculate from the algorithm itself, so it would be interesting to do some simulations. It would also be interesting to simulate how errors would propagate in networks built out of such quantum units.

Two other algorithms we wish to consider were proposed in [GHMP02] and [AMP02]. These algorithms give exact simulations of Mod gates and narrow width branching programs in terms of simpler kinds of quantum circuits. We would like to study via simulations how sensitive these algorithms are to errors in the quantum gates used to build up the quantum circuits.

This project rises to the level of a Master's students's project because of the substantial work and mathematical maturity needed to understand the algorithms involved, let alone implement them.

Three deliverables were finished in this semester. In Deliverable 1, I wrote a program that allows one to simulate the classical log-depth, AND, OR, NOT circuits for threshold gates on 10 bits. In Deliverable 2, I implement a simulator which can simlates the log*-depth threshold circuits for multiplication of two 5 bits numbers. By doing this two delivralbes, I gained first-hand knowledge of the expressive power of threshold ciruits. In Deliverable 3, I implemented a quantum circuit with unbounded fan-out gates for value gate.

2. Simulating classical log-depth, AND-OR circuits for threshold gates

A threshold gate is a logical gate doing the following computation. Given input of a 0-1 vector $A=(a_1, a_2,\ldots,a_n)$ and a threshold value k, the gate output a 0/1 value,

```
if  $(a_1+a_2+\ldots+a_n >= k)$  then
        Output = 1
else
        Output = 0
```

as shown in Figure 2-1:

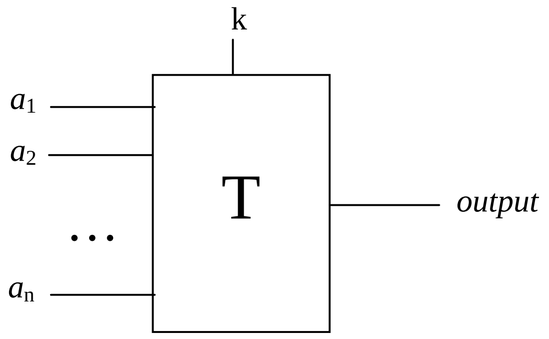


Figure 2-1 A threshold gate

An AND-OR circuit is a classical circuit with the restriction on fan -in removed. That is, the AND-gate can compute the conjunction of any number of values, and the OR-gate can compute the disjunction of any number of values. The value of the above threshold gate $T^{a1, a2,\ldots,an}_{k}$ can be evaluated in the following recursive way:

- $T^{a1, a2,\ldots,an}_{k} = T^{a1, a2,\ldots,an/2}_{0} \wedge T^{a(n/2+1), a(n/2+2),\ldots,an}_{k} \vee T^{a1, a2,\ldots,an/2}_{1} \wedge T^{a(n/2+1),a(n/2+2),\ldots,an}_{k-1} \vee \ldots \vee T^{k}_{a1, a2,\ldots,an/2}\ T^{0}_{a(n/2+1), a(n/2+2),\ldots,an}$
- $T^{n}_{0}\ldots = 1$
- $T^{a1, a2,\ldots,aj}_{i} = 0$ if $j>i$

Thus, we can construct a circuit with classical AND-OR gates to simulate a threshold gate. It is easy to see that the depth of this circuit is *log(n)*. We use the above algorithm to implement our program.

In this implementation, we use three classes. A CAND class is used for constructing and operating a classical AND gates. Similarly, a COR class is created for OR gates. Finally, a CThreshold class is used for describing threshold gate. We first construct a classical circuit consist of AND and OR gates for threshold gate. Then given input bits and a threshold value *m*, we evaluate the circuit and output the result.

3. Simulating log*-depth threshold circuits for multiplication

3.1 Program Description

This program simulates threshold circuits for multiplication of two 5-bit numbers. Given two numbers in 5-bit binary form, the program constructs a circuit consisting of only threshold gates. This circuit calculates the result of multiplying these two numbers. It has depth *O(log* n)*, and size of *O(n)*, where *n* is the number of bits for each input.

3.2 Background

In this section we will consider the following problems:

- *Problem*: ADD
  Input: two n bit numbers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$
  Output: $s = s_n \dots s_0,\ s =_{def} a + b$
- *Problem*: ITADD
  Input: n numbers in binary with n bits each
  Output: the sum of the input numbers in binary
- *Problem*: LOGITADD
  Input: log n numbers with n bits each
  Output: the sum of the input numbers in binary
- *Problem*: BCOUNT
  Input: n bits $a_{n-1}, \dots, a_0$
  Output: the sum of a_i, where i from 0 to n
- *Problem*: MULT
  Input: two n bit numbers
  Output: the product of the input numbers in binary
- *Problem*: T^n_m
  Input: n bits $a_{n-1}, \dots, a_0$, n bits $w_{n-1}, \dots, w_0$, and
  a threshold m(m is an integer and $0<m<n$)
  Output: 1 if $a_{n-1}.w_{n-1} + \dots + a_0.w_0 >= m$,
  0 otherwise

3.3 Algorithms

Given two n bit numbers $a = a_{n-1} \dots a_0$ and $b = b_{n-1} \dots b_0$, first, we can reduce MULT to ITADD by using the school method for multiplication.

Define

- $c_i = 0^{n-i-1}a_{n-1} \dots a_1a_00^i$, if $b_i = 1$, 0^{2n-1} otherwise.

Then the product of a, b is the sum of c_i, where $0 \le i < n$. Thus, the problem of computing multiplication reduces to solve ITADD.

Given the above c_i we define for every position k, $0 \le k < 2n-1$, the sum $s_k = \Sigma_{i=0}^{n-1} c_{i,k}$. Every s_k can be written in binary using no more than log(n) bits, so we have

$$\Sigma_{i=0}^{n-1} c_{i,k} = \Sigma_{j=0}^{\log(n)-1} s_{k,j} \,.\, 2^j$$

Therefore,

$$\Sigma_{i=0}^{n-1} c_i = \Sigma_{j=0}^{n-1} \Sigma_{j=0}^{n-1} s_{k,j} \,.\, 2^j$$
$$= \Sigma_{j=0}^{\log(n)-1} \Sigma_{j=0}^{n-1} s_{k,j} \,.\, 2^j \,.\, 2^k.$$

Notice that the problem of counting each s_k is a BCOUNT problem. Thus we can reduce the problem of adding *n* numbers to that of adding *log(n)* numbers $\Sigma_{k=0}^{n-1} s_{k,j} \,.\, 2^{j+k}$, where $0 \leq k < \log(n)$. Each of these numbers has no more than *n+log(n)* bits. We can iterate this procedure, that is, continue to add the above *log(n)* numbers using the same method, as the result, next time we get *log(log(n))* numbers with *n+log(n)+log(log(n))* bits, and so on. After *x* iterations, where *x* is the smallest number such that $log^{x+1} n \leq 2$, we have 2 numbers at the end, we can add them using a constant depth circuit described in [Vol00].

Next, we reduce BCOUNT to T^n_m. Let $l = log(n)$, $\Sigma_{i=0}^{n-1} s_i = s_{l-1} \ldots s_0$ in binary. Let $0 \leq j < log(n)$. Let R_j be the set of those numbers $r \in \{0, \ldots, n\}$ whose j-th bit is on (where the 0th bit is the rightmost bit and the (l-1)th bit is the leftmost bit). We have,

$$S_j = \cup_{r \in Rj} (\Sigma_{i=0}^{n-1} s_i = r).$$

Obviously,

$$(x = r) = (x \geq r) \wedge \neg(x \geq r+1)$$

thus,

$$S_j = \cup_{r \in Rj} (T^n_r (a_1, \ldots, a_n) \wedge \neg\, T^n_{r+1} (a_1, \ldots, a_n)).$$

Observe that the set R_j does not depend on the input value but only on the input length *n* and thus can be hardwired into the circuit. Now we have a circuit for BCOUNT, which uses $(T^n_r)_{n \in N}$ and $(T^n_{r+1})_{n \in N}$ gates.

To construct a multiplication circuit we also need to use the NOT, AND, OR gates. It is easy to see that:

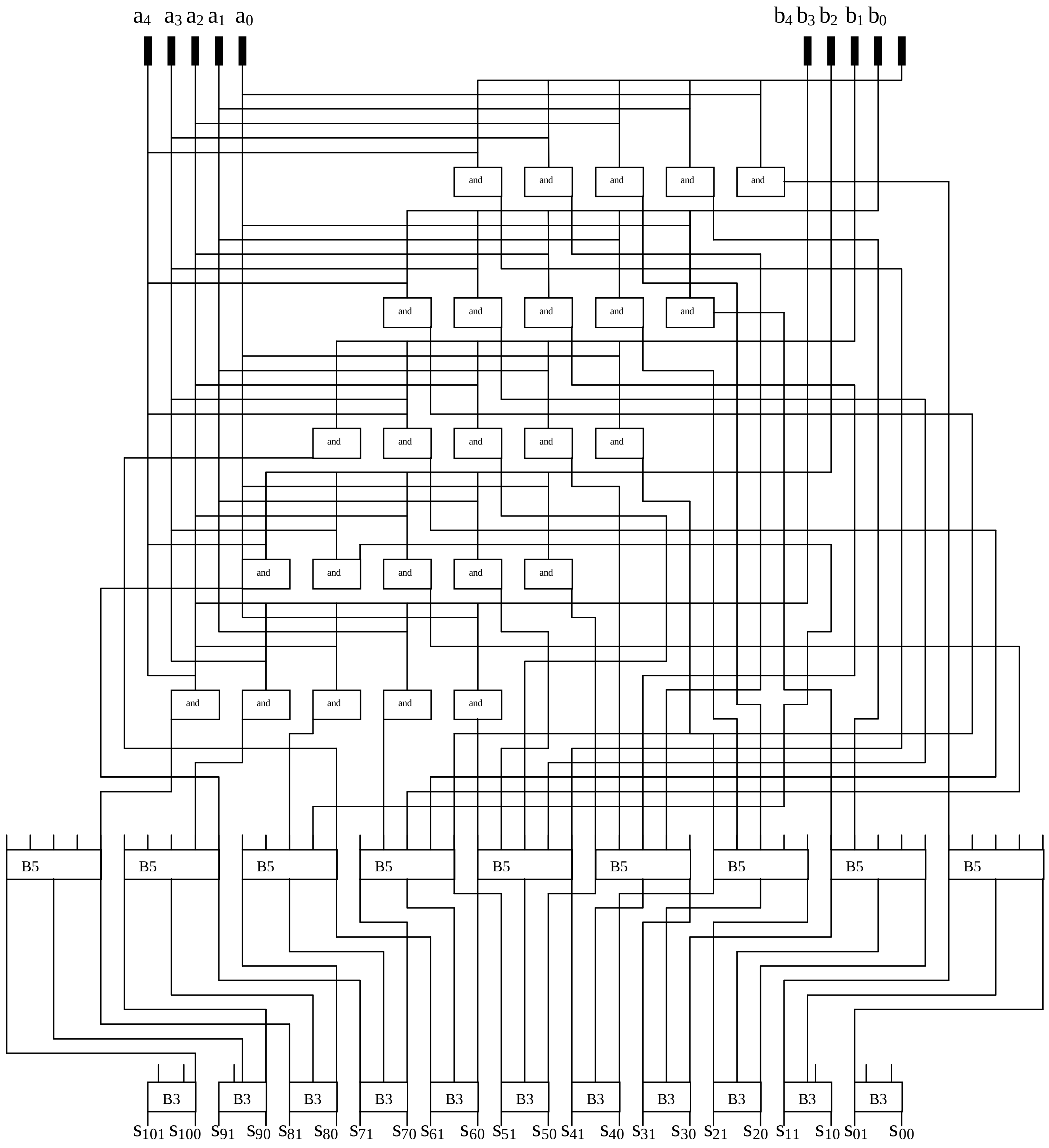
- AND = $T^n_n (w_i = 1$, where $0 \leq i < n)$
- OR = $T^n_1 (w_i = 1$, where $0 \leq i < n)$
- NOT = $T^1_0 (w = -1)$

Based on the above reductions, we can construct a circuit for MULT with only threshold gates.

3.4 Design and Implementation

3.4.1 Overall circuits design

Given two 5-bit binary numbers and to calculate the product, our program first generates a circuit. The input for this circuit are two 5-bit binary numbers, then evaluation is the process of calculation, and the evaluation result is the product of two numbers. The program generates the circuit only once, after each evaluation (calculation), it can reset the circuit for next evaluation. Figure 3-1 shows the multiplication circuits.



a_4 a_3 a_2 a_1 a_0
b_4 b_3 b_2 b_1 b_0
and
B5
B3
s_{101} s_{100} s_{91} s_{90} s_{81} s_{80} s_{71} s_{70} s_{61} s_{60} s_{51} s_{50} s_{41} s_{40} s_{31} s_{30} s_{21} s_{20} s_{11} s_{10} s_{01} s_{00}

Figure 3-1 Circuit for calculating the product of two five bits binary numbers

The depth of the circuit is seven. In first five levels we prepare $c_{i,k}$ (where $0 \leq i < n$, $0 \leq k < 2n-1$), then we send $c_{i,k}$ to the sixth level, that is, five numbers with nine bits each need to be added. Level six contains nine BCOUNT gates, where each gate has five input and three output bits. After applying level six, we have three numbers with nine bits each. We now send them to the seventh layer, which consists of 11 BCOUNT gates with three inputs and two outputs each. After this level, we get two 13-bit numbers. Adding two binary numbers can be done by a constant depth circuit that is described in [Vol00].

In Figure 3-1, the constructed circuit for MULT is made up of BCOUNT gates with five or three input bits. Each BCOUNT gate can be construct by a constant depth circuit including threshold, AND, OR, and NOT gates. Figure 3-2 illustrates the construction of a five inputs and three outputs BCOUNT gate using threshold, AND, OR gated. Figure 3-3 illustrates the construction of a three inputs and two outputs BCOUNT gate using threshold, AND, OR gated. We can further replace each AND, OR, NOT gate with threshold gates as described in section 3-3.

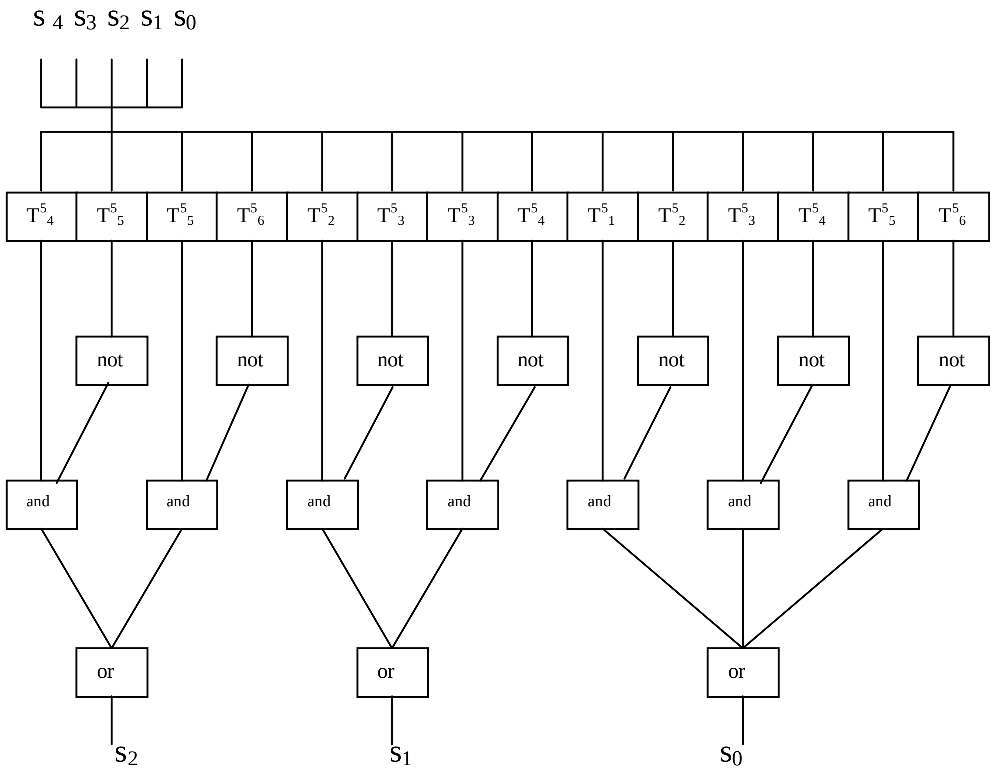


Figure 3-2 BCOUNT circuit with five inputs and three outputs

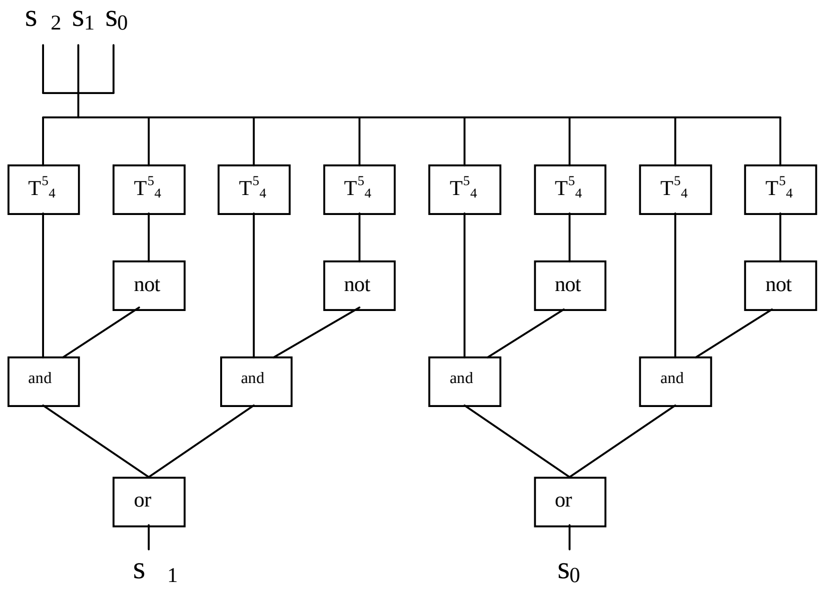


Figure 3-3 BCOUNT circuit with three inputs and two outputs

Given two numbers in binary with n bits each, using our circuit construction, we need *log*(n)* layers of BCOUNT gates. Each layer can be implemented by a constant depth threshold circuit. Thus the total depth of the threshold circuits for multiplication is *O(log*(n))*.

Note that in Figure 3-3, the value of S_0 and S_1 are solely depend on the initial input ($a_4a_3a_2a_1a_0$) and ($b_4b_3b_2b_1b_0$). We can create a circuit that directly calculates the final S_0 and S_1 result based on the initial ($a_4a_3a_2a_1a_0$) and ($b_4b_3b_2b_1b_0$) values. In the following, we show an idea of constructing constant threshold circuits for multiplication.

Suppose for any $S_{i,j}$ we have the following truth-value table, then we can use a circuit as shown in Figure 3-4 to calculate each value.

$a_4a_3a_2a_1a_0$ $b_4b_3b_2b_1b_0$	00000	…	11111
00000	0	…	1
…	…	….	…
11111	1	…	0

Table 3-1 Truth value for $S_{i,j}$

For two inputs ($a_4a_3a_2a_1a_0$) and ($b_4b_3b_2b_1b_0$), there are 2^{10}=1024 blocks, the output of each block is connected to a T^{1024}_1 gate. The output of the circuit is a truth-value in Table 3-1. Each block can further be constructed by a circuit of depth two, as shown in Figure 3-5 (using $a_4a_3a_2a_1a_0$ = 00000 and $b_4b_3b_2b_1b_0$ = 00000 as example). It is easy to replace each

NOT-gate by a depth-one threshold circuit. Thus, the total depth of threshold circuits for multiplication is three.

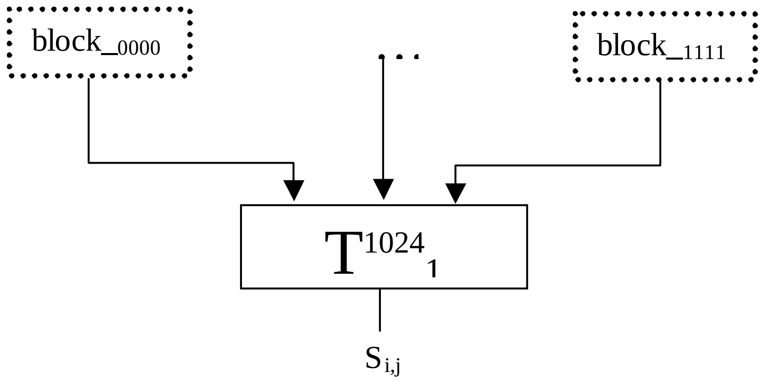


Figure 3-4 Threshold circuits to calculate Si,j

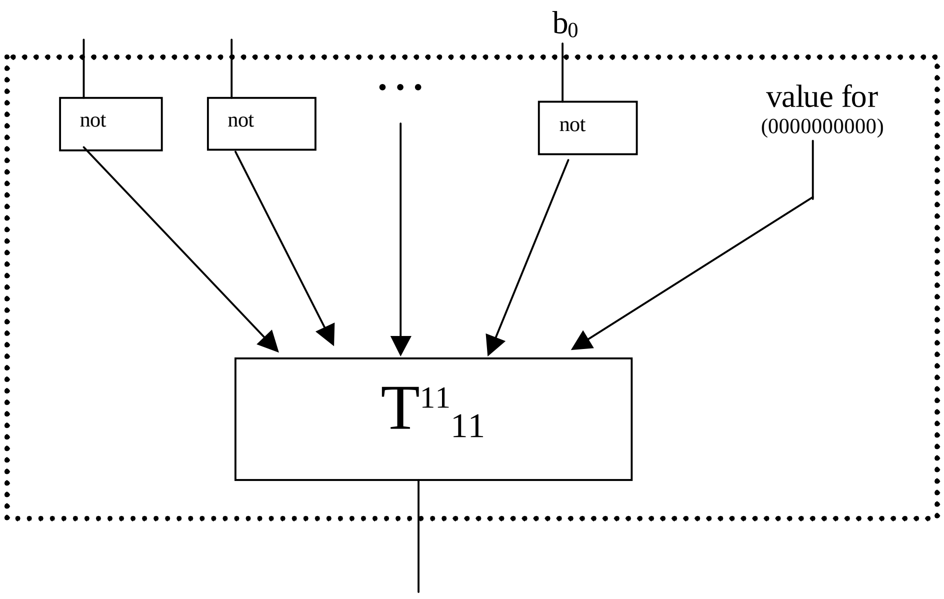


Figure 3-5 Circuit construction for each block

3.4.2 Program organization

Basically, three structures were used: *pin, threshold* and *blkbox* structure.

- In the *pin* structure, we record input and output information using linked-lists. Each element of a linked-list is a pointer that points to a *pin*.
- A *threshold* gate has input and output *pins*, and its threshold value k.

- A *blkbox* is a structure made of input *pins*, output *pins*, and internal *threshold* gates. We use different *blkbox* structures to implement AND, OR, NOT, or BCOUNT gates.

Having all these elements, we can finally construct our circuits for multiplication. Figure 3-6 shows our program organization.

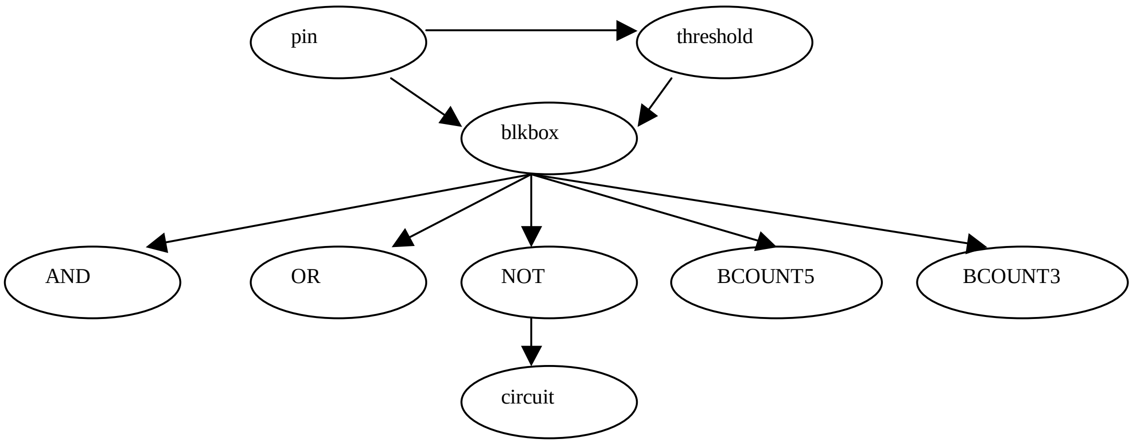


Figure 3-6 Program organization diagram

3.5 Test

A test program was written, which takes the circuit as its input and evaluates step-by-step the status and value of the threshold gates. Because this circuit is only calculating the product of two 5-bit binary numbers, it is possible to verify each possible input and output. My evaluation shows that my circuits correctly compute the result of multiplying two 5-bit binary numbers in constant time.

4. Simulating quantum circuits with unbounded fan-out for value gate

4.1 Program Description

This program performs a linear value gate calculation. Given n qubits $|x>$ and value m, the program constructs a quantum circuit with fan-out, and then evaluates the circuit. The idea for constructing such a circuit can be found in [S02].

4.2 Quantum Computation

Quantum mechanics is a mathematical framework for the development of physical theories. In this section, we will review some important concepts of quantum mechanics, for details, see the textbook of Quantum computation [NC00].

4.2.1 Models of quantum computation

In classical complexity theory, several equivalent models can represent the concept of universal computer. As for quantum computation, each of these classical concepts has a quantum counterpart, as shown in Table 4-1.

Model	Classical	Quantum
Mathematical	Partial recursive function	Unitary operators
Machine	Turing machine	Quantum Turing machine
Circuit	Logical circuit	Quantum circuit

Table 4-1 Classical and quantum computation models

4.2.2 State space

Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbert space), known as the state space of the system. The system is completely described by its state vector, which is a unit vector in the system's state space.

A qubit is a two-dimensional state space. Suppose $|0>$ and $|1>$ form an orthonormal basis for that state space. Then an arbitrary state vector in the state space can be written as:

$$|\psi> = a|0> + b|1>$$

where a and b are complex numbers, and $|a|^2 + |b|^2 = 1$.

4.2.3 Evolution

The evolution of a closed quantum system is described by a unitary transform. That is, the state $|\psi>$ of the system at time t_1 is related to the state $|\psi'>$ of the system at time t_2 by a unitary operator U which depends only on the times t_1 and t_2,

$$|\psi'> = U|\psi>$$

A transform is said to be unitary if the matrix, say U, representing this gate is unitary, that is, $U^\dagger U = I$. An example of unitary operator is the Hadamard gate, which we denote as H.

This has the action $H|0\rangle \equiv (|0\rangle + |1\rangle)/\sqrt{2}$, $H|1\rangle \equiv (|0\rangle - |1\rangle)/\sqrt{2}$, and corresponding matrix representation

$$H = 1/\sqrt{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

4.2.4 Quantum measurement

Quantum measurements are described by a collection $\{M_m\}$ of measurement operators. These are operators acting on the state space of the system being measured. The index *m* refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement, then the probability that result *m* occurs is given by

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$$

and the state of the system after the measurement is

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

The measurement operators satisfy the completeness equation

$$\sum_m M_m^\dagger M_m = I$$

4.2.5 Composite systems

The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through *n*, and system number *i* is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \dots \otimes |\psi_n\rangle$.

4.3 Algorithms

In this section, we will briefly review the main results of the basic algorithm, which is used to construct the circuit implemented in this program, for details, see [S02]. In forthcoming circuits, we will need the following tools: quantum fan-out operation, parallelization method, quantum Hadamard transformation, and increment operation.

4.3.1 Quantum fan-out operation

Notice that because of the no-cloning theorem, quantum circuits cannot contain a naïve quantum counterpart of the classical fan-out operation performing

$$|s\rangle|0\rangle^{\otimes n} \rightarrow |s\rangle^{\otimes n+1} \qquad (4.3.1)$$

for a general superposition state $|s\rangle$. However, a modified quantum fan-out operation can be defined. It performs exactly (4.3.1) for each computational basis state and the effect on superposition states is determined by linearity. The quantum fan-out operation can also be regarded as a sequence of controlled NOT operations sharing the source qubit.

Define a quantum fan-out operation on source qubit $|s>$ and *n* target qubits $|t_k>$ performs

$$|s>\bigotimes_{k=1}^{n}|t_k> \rightarrow |s>\bigotimes_{k=1}^{n}|t_k\oplus s>$$

4.3.2 Parallelization method

Having the above quantum fan-out operator, now we are able to perform a more general task: an arbitrary number of (possibly controlled) commuting operations operating on the same qubits can be performed at the same time in the model of quantum circuits with unbounded fan-out.

First, if some operators commute, then they are all diagonal in the same basis. Second, a diagonal unitary operator consists just of phase shifts, because every coefficient in the diagonal is a complex unit. Furthermore, applying multiple phase shifts on an individual qubit can be parallelized as following:

1. Apply the fan-out operation on a qubit to copy the state.
2. Apply each phase shift on a distinct "copy".
3. Apply the fan-out operation again, and clear the ancilla qubits.

This method can be readily generalized from one qubit to a set of qubits by copying all target qubits using the fan-out operation after basis change. Other tricks are similar to one qubit case.

4.3.3 Quantum Hadamard transform

Quantum Fourier Transform (QFT) is one of the main tools of quantum computing. It is a quantum equivalent of the classical Fourier Transform, but it operates on quantum amplitudes of superposition states instead of an array of numbers. For detail about QFT, see [NC00]. The main trick used in this algorithm is replacing QFT by Hadamard transform.

The Hadamard transform H_n on *n* qubits is the following operator (written in the computational basis):

$$H_n = 1/2^{n/2} \Sigma_{j=0}^{2n-1}|y> \Sigma_{j=0}^{2n-1}(-1)^{y.x}<x|,$$

where y.x is the bitwise scalar product. A useful property of the Hadamard transform is $H_n = H^{\otimes n}$.

4.3.4 Increment operation

An increment operator P on n qubits is an operator mapping each computational basis state $|x>$ to $|x+1 \bmod 2^n>$. P is diagonal in the Fourier basis and, in this basis, can be implemented exactly by a depth one quantum circuit.

Define operator $D = FPF^{\dagger}$, that is the increment in Fourier basis. Define a rotation operator about the z-axis by angle θ by $R_z(\theta) = |0><0| + e^{\theta i}|1><1|$. Then for every $k \in \{1, 2, .., n\}$, $D_k = R_z(\pi/ 2^{n-k}) \otimes D_{k-1}$. The 0-qubit operator D_0 is considered to be 1.

4.3.5 Value gate

Having all these tools, a circuit for value gate with unbounded fan -out can be constructed as shown in Figure 4-1.

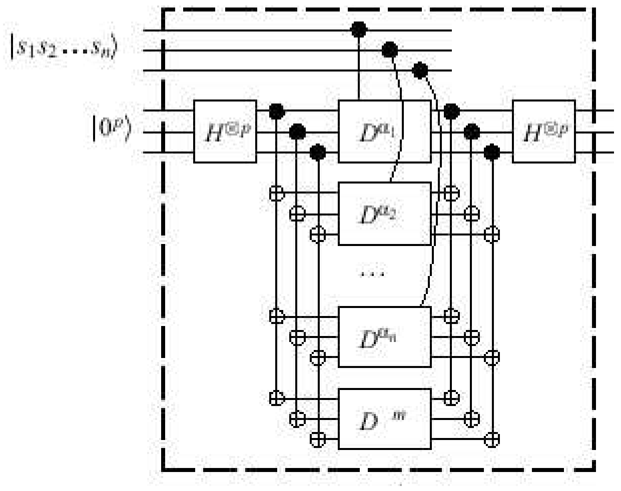


Figure 4 -1 A circuit for value gate

4.4 Design and Implementation

One of the most challenging parts in implementing this algorithm is to understand substantial mathematics involved in the algorithm and quantum computation. It took me long time to figure out the matrix for each quantum gate involved in this circuit.

4.4.1 Matrix design

From an implementation view, four types of layers are included in Figure 4-1: Permutation layer, Hadamard layer, Fan-out layer, and Increment layer. According to this consideration, from left to right, we first apply Hadamard layer to change the basis of the input vector. Then we apply permutation layer to change the order of input line. Then, apply fan-out layer, after this, we apply permutation layer again to change the order back, and apply another permutation layer to give an order that needed by applying increment layer. Then, by applying controlled increment layer on Hadamard basis, we can calculate if the number of inputs that is on is equal to a value *m*. We again applying permutation, fan-out, permutation, and Hadamard layer respectively to change the vector back to computational basis and eliminate ancilla qubits.

Each layer can be represented as a matrix. This matrix comes from the tensor-product of sub-matrixes. The entire circuit can also be represented as a matrix, which is the product of all layers. A matrix representation for the Hadamard layer (we call it H_layer) is straightforward. H_layer = $I_1 \otimes \ldots \otimes I_n \otimes H^{\otimes(n+1).p}$. Next we will explain matrix representation for other layers.

- *Matrix representation for permutation operator*

Give *n* inputs in order *x*, we may need to change order *x* to order *y*,

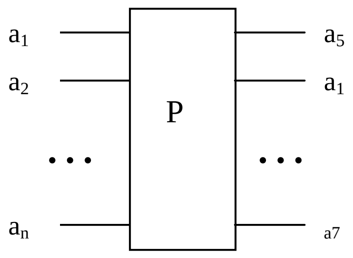


Figure 4-2 Permutation operator

The above Figure 4-2 illustrates a permutation. Input vector is $x = (a_1,a_2,.., a_n)$, after permutation operation, the output is a permutation of $(a_1,a_2,.., a_n)$, as $y = (a_5,a_1,..,a_7)$ shown above.

The permutation operation can be implemented with the following matrix:

$$\begin{bmatrix} p_{11} & p_{12} \cdots p_{1n} \\ p_{21} & p_{22} \cdots p_{2n} \\ \cdots & \cdots\cdots\cdots \\ p_{n1} & p_{n2} \cdots p_{nn} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \cdot\cdot \\ a_n \end{bmatrix} = \begin{bmatrix} a_5 \\ a_1 \\ \cdot\cdot \\ a_7 \end{bmatrix}$$

Figure 4-3 Matrix representation for permutation operator

For each element a_i in vector $x = (a_1,a_2,\ldots, a_n)$, it is permutated to the output element a_j in $y = (a_5,a_1,..,a_7)$ which y is a permutation of x. The matrix has the following property:

- $p(a_j, a_i) = 1$ for $1 <= i, j <= n$
- $p(i,j) = 0$, otherwise

- *Matrix representation for fan-out operator*

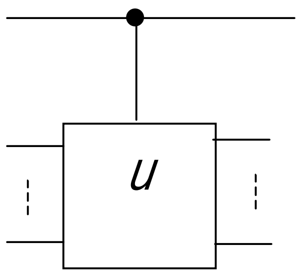
A matrix representation for fan-out gate of *n* target bits is illustrated in Figure 4-4. Define m = 2^(n+1). The size of this matrix is m x m. For the first m/2 rows, p(i, j) = 1 if i = j. For the rest rows with row number k, p(k, j) = 1 if j = m/2+m-1-k. Otherwise p(k, j) = 0.

$$\begin{bmatrix}
1 & 0 & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & 1 & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & 0 & 1 & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & 1 & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & 0 & 0 & \dots & \dots & \dots & 1 \\
0 & \dots & \dots & \dots & 0 & 0 & \dots & \dots & 1 & \dots \\
0 & \dots & \dots & \dots & 0 & 0 & \dots & 1 & \dots & 0 \\
0 & \dots & \dots & \dots & 0 & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & 0 & 1 & \dots & \dots & \dots & 0
\end{bmatrix}$$

Figure 4-4 Matrix representation of fan-out operator

- *Matrix representation for controlled-increment operator*

We consider a general case, that is, a controlled-U operator. In our implementation, we can limit our considerations to one controlled bit and *n* target bits. In this case, we have the matrix representation shown in Figure 4-5.



$$\rightarrow \begin{bmatrix}
1 & 0 & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & 1 & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & 0 & 1 & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & \dots & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & 1 & 0 & \dots & \dots & \dots & 0 \\
0 & \dots & \dots & \dots & 0 & & & & & \\
0 & \dots & \dots & \dots & 0 & & & & & \\
0 & \dots & \dots & \dots & 0 & & & \text{U} & & \\
0 & \dots & \dots & \dots & 0 & & & & & \\
0 & \dots & \dots & \dots & 0 & & & & &
\end{bmatrix}$$

Figure 4-5 Controlled-U operator and its matrix representation

In this matrix, we have 2^(n+1) rows and 2^(n+1) columns. The sub-matrix that is made from the first 2^(n+1)/2 rows by the first 2^(n+1)/2 columns is a identity matrix. To implement a controlled-increment operator on *n* target bits, we can replace U by a matrix for increment, as described in section 4.3.4.

4.4.2 Program organization

In implementation, we designed four classes: Complex number class, Matrix class, Gate class and Value gate class. In the Complex number class, we define complex number and its operations. In the Matrix class, we define matrix and matrix operations, such as addition, multiplication, transpose, tensor-product, and Hermitian-conjugate. In the Gate class, we define a base gate, and set matrix for identity gate, Hadamard gate, fan-out gate, flip(permutation) gate, and controlled increment gate. Finally, in the Value gate class, we set up a quantum circuit for value gate and evaluate the circuit to output result. Figure 4-6 shows a class organization.

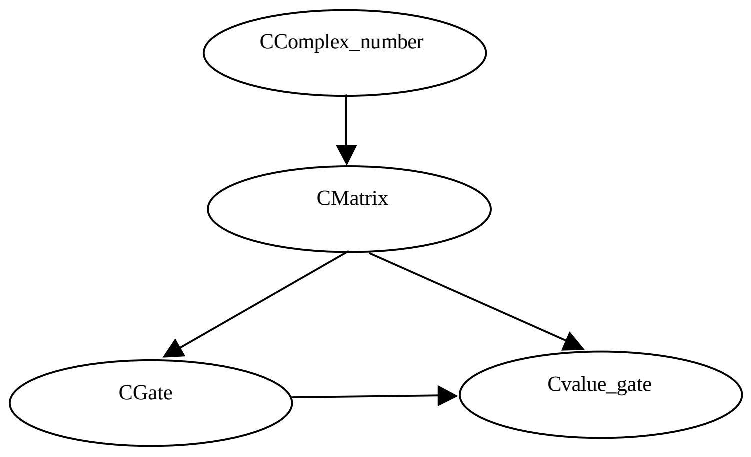


Figure 4-6 Class diagram

4.5 Test

We tested the one-qubit case. According to the algorithm, if the value gate is satisfied, that is, the number of inputs is on equal to the value m, then the register, say |Z> is in computational basis state |0>. Otherwise, register |Z> is in a gener al superposition state. By doing a measurement <Z|Z>, we get an output value, we then negate this value to output the result. Table 4-2 shows test result on one-qubit case. In this table, we see that if value gate is satisfied, program output 1, otherwise 0.

Input qubit	Value m	Output
|0>	0	1
|0>	1	0
|1>	0	0
|1>	1	1

Table 4-2 Test result on one qubit case

4.6 Limitation of the Implementation

Currently, our program is an initial implementation. For now, both the space complexity and the time complexity are high. Given *n* qubits, the space and the time complexity for this implementation is *O(2^nlog(n))*. We know, however, that this algorithm is in PSPACE, because BQP lies somewhere between P and PSPACE [NC00], see Figure 4-7.

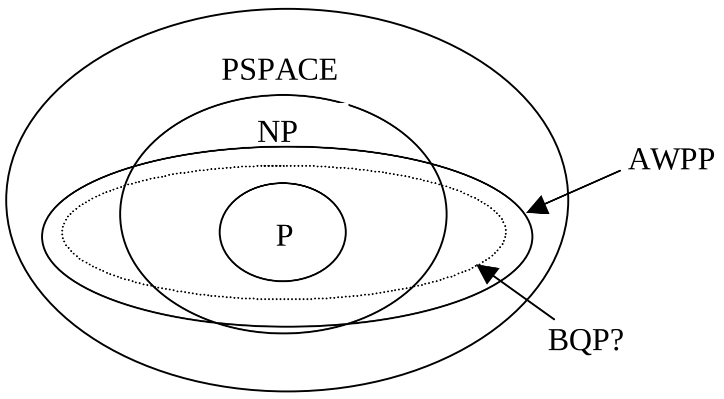


Figure 4-7 The relationship between classical and quantum complexity classes

Because of the exponential complexity of our current implementation, we are not able to evaluate large number of qubits. However, it does give us the insight of how the quantum algorithm works. Currently, we are actively exploring ways of reducing both the space and the time complexity and are making good progress. We will discuss this in conclusions and further works section.

5. Conclusions and further works

In this semester, I implemented three programs. By doing the first two programs, I became familiar with simulating classical circuits, and gained a feeling of the power of threshold circuits as a computational model.

In the third program, I implemented an initial version of a quantum circuit for unbounded fan-out value gate. The basic algorithm to construct this circuit is described in [S02]. My current implementation is a fully functioning simulator of quantum circuits for the value gate. One disadvantage for this simulator is its exponential space and time complexity, which makes simulating large circuits too costly. We are exploring the following ideas to reduce the space/time complexity to polynomial in the inputs of our circuits.

The basic operations we used to support our implementation include tensor-product, permutation operator, and increment operator. We are considering having a method element (i, j) that returns an element of the i-th row and the j-th column in a matrix. We found that it is possible to write such a method on tensor-product, permutation operator, and increment operator, and have the time complexity be polynomial. Thus in our constructed circuits, we can avoid setting up the exponential length by exponential width matrices. Yet, in practice, many considerations are needed to be thought out in more details.

In the next semester, I will implement the above ideas. Another work is to introduce error matrices and to do a large amount of tests to estimate how errors would propagate in networks built out of such quantum units.

References

[AMP02] F.Ablayev, C.Moore, and C.Pollett. Quantum and Stochastic Branching Programs of Bounded Width. 29th International Colloquium on Automata, Languages, and Programming (ICALP). 2002. p.343--354. ECCC TR02-013.

[FGHP99] S. Fenner, F. Green, S. Homer, and R. Pruim. Determining Acceptance Possibility for a Quantum Computation is Hard for the Polynomial Hierarchy. Proceedings of the Royal Society A (1999) 455, pp 3953--3966.

[GHMP02] F.Green, S.Homer, C.Moore, C.Pollett. Counting, Fanout, and the Complexity of Quantum ACC. Quantum Information and Computation. Vol. 2. No. 1. 2002. pp.35--65.

[I94] Circuit Complexity and Neural Networks. Ian Parberry. The Mit Press. 1994.

[NC00] Quantum Computation and Quantum Information. M.Nielson and I.Chuang. Cambridge. 2000.

[Pap94] Computational Complexity. C.Papapdimitriou. Addison Wesley. 1994.

[PS97] P.Shor. Polynomial -time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing 26(5): 1484-1509, 1997.

[S02] R. Spalek. Quantum Circuits with Unbounded Fan-out. STACS 2003 v2. 2003.

[Vol00] Introduction to Circuit Complexity. H.Vollmer. Springer-Verlag. 2000.

[Y93] A. C.-C. Yao. Quantum circuit complexity. Proc. 34th IEEE Symposium on Foundations of Computer Science, 352--361, 1993.

Appendix A: Source code for Deliverable 2

```
//mult.cpp

/////////////////////////////////////////////////////////////////
//This program performs multiplication between two
//five bits binary numbers. The program first generate
//a constant depth circuit that consist of only threshold
//gates, then evaluate the output value of generated circuits.
/////////////////////////////////////////////////////////////////
#include <iostream>
#include <string>
#include <cassert>

using namespace std;

const NUMBER_OF_BITS = 5;

const int MAX_FAN_IN = 22;
const int MAX_FAN_OUT = 22;

const int N_LEVEL1 = 9; //at level1 three 9 bits numbers are generated
const int N_LEVEL2 = 11;//at level2 two 11  bits numbers are generated
const int N_LEVEL3 = 13;//at level3 one 13 bits numbers are generated

//////////////////////////////////////////////
//Input/Output Pins structures and functions
//////////////////////////////////////////////

struct input_pin_struct
{
	bool is_value;
	union
	{
		int value;
		struct output_pin_struct *from;
	}union_field;
};

struct output_pin_node
{
	struct input_pin_struct *to_inpin;
	struct output_pin_node *next;
};

struct output_pin_struct
```

```
{
        bool valid;
        int value;
        struct output_pin_node *to_inpin;
};

void set_inpin_value(struct input_pin_struct *inpin, int value)
{
        inpin->is_value = true;
        inpin->union_field.value = value;
}

void connect_pin(struct output_pin_struct *outpin,
                                struct input_pin_struct *inpin)
{
        inpin->union_field.from = outpin;
        inpin->is_value = false;

        struct output_pin_node *p;
        p = new struct output_pin_node;
        p->to_inpin = inpin;
        if (outpin->to_inpin == NULL)
        {
                p->next = NULL;
                outpin->to_inpin = p;
        }
        else
        {
                p->next = outpin->to_inpin;
                outpin->to_inpin = p;
        }
}


////////////////////////////////////////
//Threshold structure and functions
////////////////////////////////////////

struct threshold_struct
{
        int num_input;
        int k;
        int weight[MAX_FAN_IN];
        struct input_pin_struct inpins[MAX_FAN_IN];
        struct output_pin_struct outpins;
};
```

```
// This structure is for evaluating the multiplication circuit
// When a threshold gate is generated, a eval_node is created
// and is linked to head list.
struct eval_node
{
        struct threshold_struct *item;
        bool valid;
        int value;
        struct eval_node *next;
} *head = NULL;

void construct_threshold(struct threshold_struct *t, int n, int k)
{
        int i;
        struct eval_node *peval;

        t->num_input = n;
        t->k = k;

        for(i=0; i<n; i++)
                t->weight[i] = 1;

        t->outpins.valid = false;
        t->outpins.to_inpin = NULL;

        peval = new struct eval_node;
        peval->item = t;
        peval->valid = false;
        peval->next = head;
        head = peval;
}

//This function is for NOT gate. Normally, each weight is 1.
//For NOT gate, weight is -1, and k is 0.
void change_threshold_weights(struct threshold_struct *t,
                                                        int weight[])
{
        int i;
        for (i=0; i<t->num_input; i++)
        {
                t->weight[i] = weight[i];
        }
}

bool evaluate_threshold(struct threshold_struct *t, int *result)
```

```
{
        int i,value=0;

        for(i=0; i<t->num_input; i++)
        {
                if(t->inpins[i].is_value == true)
                        value += t->inpins[i].union_field.value * t->weight[i];
                else
                {
                        if(t->inpins[i].union_field.from->valid == true)
                                value += t->inpins[i].union_field.from->value * t-
>weight[i];
                        else
                                return false;
                }
        }

        if (value >= t->k)
                *result = 1;
        else
                *result = 0;

        return true;
}


////////////////////////////////////////////////
//Black box structure and functions
////////////////////////////////////////////////

//A black box is a virtual box, which is made from inpins,
//outpins. BCOUNT,AND,OR gates can be seen as a constructed
//black box, each one has its own threshold circuits.
//Multiplication circuit is made from four levels black boxes,
//each level has diffrent number of black boxes.

struct blkbox_inpin_node
{
        struct input_pin_struct *pin;
        struct blkbox_inpin_node *next;
};

struct blkbox_struct
{
        int num_input;
        int num_output;
        struct blkbox_inpin_node *intable[MAX_FAN_IN];
        struct output_pin_struct *outtable[MAX_FAN_OUT];
}
 level0_boxes[NUMBER_OF_BITS][NUMBER_OF_BITS],
 level1_boxes[N_LEVEL1],
 level2_boxes[N_LEVEL2],
 level3_box;


void connect_blkbox(struct output_pin_struct *opin,
                                        struct blkb ox_inpin_node *bbpin)
{
        while (bbpin != NULL)
        {
                connect_pin (opin, bbpin->pin);
                bbpin = bbpin->next;
        }
}

void set_blkbox_inpin_value(struct blkbox_inpin_node *inpin,
                                                     int value)
{
        while (inpin != NULL)
        {
                set_inpin_value(inpin->pin,value);
                inpin = inpin->next;
        }
}

bool get_blkbox_outpin_value(struct output_pin_struct *outpin,
                                                     int *value)
{
        *value = outpin->value;
        return outpin->valid;
}


//////////////////////////////////////////////////////////////
//Construct AND, OR, BCNT gates
//Each construct function can be seen as a "fill" black box
//procedure.
//////////////////////////////////////////////////////////////

//This function implements construction of a unbounded AND
//gate. An AND gate can be construct through a threshold
//gate, whose k=n.
```

```
void set_and_gate(struct blkbox_struct *pand, int num_input)
{
        int i;
        struct blkbox_inpin_node *bbinp;
        struct threshold_struct *t;

        pand->num_input = num_input;
        pand->num_output = 1;

        t = new struct threshold_struct;
        construct_threshold(t,num_input,num_input);

        for(i=0; i<num_input; i++)
        {
                bbinp = new struct blkbox_inpin_node;
                bbinp->pin = &(t->inpins[i]);
                bbinp->next = NULL;
                pand->intable[i] = bbinp;
        }

        pand->outtable[0] = &(t->outpins);
}

//This function implements construction of a unbounded OR
//gate. A OR gate can be construct through a threshold
//gate, whose k=1.
void set_or_gate(struct blkbox_struct *por, int num_input)
{
        int i;
        struct blkbox_inpin_node *bbinp;
        struct threshold_struct *t;

        por->num_input = num_input;
        por->num_output = 1;

        t = new struct threshold_struct;
        construct_threshold(t,num_input,num_input);

        for(i=0; i<num_input; i++)
        {
                bbinp = new blkbox_inpin_node;
                bbinp->pin = &(t->inpins[i]);
                bbinp->next = NULL;
                por->intable[i] = bbinp;
        }
```

```
	por->outtable[0] = &(t->outpins);
}

//This function implements construction of a BCOUNT gate,
//which has five inputs and three outputs. A BCONT gate
//is made from four level threshold gates, each level hae
//14, 7, 7, 3 threshold gats respectively.
void set_bcnt5_gate(struct blkbox_struct *pbcnt)
{
	int minus_one[] = {-1};
	int i,j;
	struct threshold_struct *pt1[14], *pt2[7], *pt3[7], *pt4[3];
	int pt1k[] = {6,5,4,3,2,1,4,3,3,2,6,5,5,4};//k value of 14 first
				  //level threshold gates
	struct blkbox_inpin_node *bbinp;

	pbcnt->num_input = 5;
	pbcnt->num_output = 3;

//generate 14 first level threshold gates
	for(i=0; i<14; i++)
	{
		pt1[i] = new threshold_struct;
		construct_threshold(pt1[i],5,pt1k[i]);
	}
//generate 7 second level threshold gates
	for(i=0; i<7; i++)
	{
		pt2[i] = new threshold_struct;
		construct_threshold(pt2[i],1,0);
		change_threshold_weights(pt2[i],minus_one);
	}
//generate 7 third level threshold gates
	for(i=0; i<7; i++)
	{
		pt3[i] = new threshold_struct;
		construct_threshold(pt3[i],2,2);
	}
//generate 3 forth level threshold gates
//pt4[0] has three inputs instead of two like other gates
//so is handled seperatly.
	pt4[0] = new threshold_struct;
	construct_threshold(pt4[0],3,1);
	for(i=1; i<3; i++)
	{
		pt4[i] = new threshold_struct;
```

```
			construct_threshold(pt4[i],2,1);
		}

	//Connect network, which is to connect outpins and inpins
	//among threshold gates.
		for(i=0; i<7; i++)
			connect_pin( &(pt1[i*2]->outpins), &(pt2[i]->inpins[0]) );

		for(i=0; i<7; i++)
		{
			connect_pin( &(pt2[i]->outpins), &(pt3[i]->inpins[0]) );
			connect_pin( &(pt1[i*2+1]->outpins), &(pt3[i]->inpins[1]) );
		}

		connect_pin( &(pt3[0]->outpins), &(pt4[0]->inpins[0]) );
		connect_pin( &(pt3[1]->outpins), &(pt4[0]->inpins[1]) );
		connect_pin( &(pt3[2]->outpins), &(pt4[0]->inpins[2]) );

		for(i=1; i<3; i++)
		{
			connect_pin( &(pt3[i*2+1]->outpins), &(pt4[i]->inpins[0]) );
			connect_pin( &(pt3[i*2+2]->outpins), &(pt4[i]->inpins[1]) );
		}

		for(i=0; i<5; i++)
		{
			pbcnt->intable[i] = NULL;
			for(j=0; j<14; j++)
			{
				bbinp = new struct blkbox_inpin_node;
				bbinp->pin = &(pt1[j]->inpins[i]);
				bbinp->next = pbcnt->intable[i];
				pbcnt->intable[i] = bbinp;
			}
		}

		for(i=0; i<3; i++)
			pbcnt->outtable[i] = &(pt4[i]->outpins);

	}

	//Construnction of BCONT gate, which has three inputs two outputs
	//Similar to BCONT5
	void set_bcnt3_gate(struct blkbox_struct *pbcnt)
	{
		int minus_one[] = {-1};
```

```
        int i,j;
        struct threshold_struct *pt1[8], *pt2[4], *pt3[4], *pt4[2];
        int pt1k[] = {2,1,4,3,3,2,4,3};
        struct blkbox_inpin_node *bbinp;

        pbcnt->num_input = 3;
        pbcnt->num_output = 2;

//construct 4 level threshold gates
        for(i=0; i<8; i++)
        {
                pt1[i] = new threshold_struct;
                construct_threshold(pt1[i],3,pt1k[i]);
        }

        for(i=0; i<4; i++)
        {
                pt2[i] = new threshold_struct;
                construct_threshold(pt2[i],1,0);
                change_threshold_weights(pt2[i],minus_one);
        }

        for(i=0; i<4; i++)
        {
                pt3[i] = new threshold_struct;
                construct_threshold(pt3[i],2,2);
        }

        for(i=0; i<2; i++)
        {
                pt4[i] = new threshold_struct;
                construct_threshold(pt4[i],2,1);
        }

//Connect network
        for(i=0; i<4; i++)
                connect_pin( &(pt1[i*2]->outpins), &(pt2[i]->inpins[0]) );

        for(i=0; i<4; i++)
        {
                connect_pin( &(pt2[i]->outpins), &(pt3[i]->inpins[0]) );
                connect_pin( &(pt1[i*2+1]->outpins), &(pt3[i]->inpins[1]) );
        }

        for(i=0; i<2; i++)
        {
```

```
		connect_pin( &(pt3[i*2]->outpins), &(pt4[i]->inpins[0]) );
		connect_pin( &(pt3[i*2+1]->outpins), &(pt4[i]->inpins[1]) );
	}

	for(i=0; i<3; i++)
	{
		pbcnt->intable[i] = NULL;
		for(j=0; j<8; j++)
		{
			bbinp = new struct blkbox_inpin_node;
			bbinp->pin = &(pt1[j]->inpins[i]);
			bbinp->next = pbcnt->intable[i];
			pbcnt->intable[i] = bbinp;
		}
	}

	for(i=0; i<2; i++)
		pbcnt->outtable[i] = &(pt4[i]->outpins);

}

///////////////////////////////////////////////////////
//Implementaion of adding two numbers.
///////////////////////////////////////////////////////

void set_fast_adder(struct blkbox_struct *padder)
{
	int i, j, k;
	int minus_one = -1;
	struct threshold_struct *g[N_LEVEL2+1], *p[N_LEVEL2+1],
		*c[N_LEVEL3], *t;
	struct threshold_struct *xors[N_LEVEL2+1][3], *nots[N_LEVEL2+1],
		*ands[N_LEVEL2+1], *ors[N_LEVEL2+1];
	struct blkbox_inpin_node  *bbp;

	padder->num_input = 22;
	padder->num_output = N_LEVEL3;

	for (i=0; i<N_LEVEL2+1; i++)
	{
		g[i] = new threshold_struct;
		construct_threshold(g[i], 2, 2);

		p[i] = new threshold_struct;
		construct_threshold(p[i], 2, 1);
```

```
        xors[i][0] = new threshold_struct;
        construct_threshold(xors[i][0], 3, 3);

        xors[i][1] = new threshold_struct;
        construct_threshold(xors[i][1], 3, 2);

        xors[i][2] = new threshold_struct;
        construct_threshold(xors[i][2], 3, 1);

        nots[i] = new threshold_struct;
        construct_threshold(nots[i], 1, 0);
        change_threshold_weights(nots[i], &minus_one);
        connect_pin(&(xors[i][1]->outpins), &(nots[i]->inpins[0]));

        ands[i] = new threshold_struct;
        construct_threshold(ands[i], 2, 2);
        connect_pin(&(nots[i]->outpins), &(ands[i]->inpins[0]));
        connect_pin(&(xors[i][2]->outpins), &(ands[i]->inpins[1]));

        ors[i] = new threshold_struct;
        construct_threshold(ors[i], 2, 1);
        connect_pin(&(xors[i][0]->outpins), &(ors[i]->inpins[0]));
        connect_pin(&(ands[i]->outpins), &(ors[i]->inpins[1]));
}

c[1] = g[0];
for (j=0; j<3; j++)
        connect_pin(&(c[1]->outpins), &(xors[1][j]->inpins[2]));
for (i=2; i<N_LEVEL2+1; i++)
{
        c[i] = new threshold_struct;
        construct_threshold(c[i], i, 1);
        connect_pin(&(g[i-1]->outpins), &(c[i]->inpins[i-1]));
        for (j=i-1; j>0; j--)
        {
                t = new threshold_struct;
                construct_threshold(t, j+1, j+1);

                connect_pin(&(g[i-1-j]->outpins), &(t->inpins[j]));
                for (k=j; k>0; k--)
                        connect_pin(&(p[i-k]->outpins), &(t->inpins[k-1]));
                connect_pin(&(t->outpins), &(c[i]->inpins[j-1]));
        }
        for (j=0; j<3; j++)
                connect_pin(&(c[i]->outpins), &(xors[i][j]->inpins[2]));
}
```

```
for (i=0; i<N_LEVEL2; i++)
{
	padder->intable[i] = NULL;
	bbp = new blkbox_inpin_node;
	bbp->pin = &(p[i]->inpins[0]);
	bbp->next = padder->intable[i];
	padder->intable[i] = bbp;
	bbp = new blkbox_inpin_node;
	bbp->pin = &(g[i]->inpins[0]);
	bbp->next = padder->intable[i];
	padder->intable[i] = bbp;
	for (j=0; j<3; j++)
	{
		bbp = new blkbox_inpin_node;
		bbp->pin = &(xors[i][j]->inpins[0]);
		bbp->next = padder->intable[i];
		padder->intable[i] = bbp;
	}
}
set_inpin_value(&(g[N_LEVEL2]->inpins[0]), 0);
set_inpin_value(&(p[N_LEVEL2]->inpins[0]), 0);
for (j=0; j<3; j++)
	set_inpin_value(&(xors[N_LEVEL2][j]->inpins[0]), 0);


for (i=N_LEVEL2; i<2 * N_LEVEL2; i++)
{
	padder->intable[i] = NULL;
	bbp = new blkbox_inpin_node;
	bbp->pin = &(p[i % N_LEVEL2 + 1]->inpins[1]);
	bbp->next = padder->intable[i];
	padder->intable[i] = bbp;
	bbp = new blkbox_inpin_node;
	bbp->pin = &(g[i % N_LEVEL2 + 1]->inpins[1]);
	bbp->next = padder->intable[i];
	padder->intable[i] = bbp;
	for (j=0; j<3; j++)
	{
		bbp = new blkbox_inpin_node;
		bbp->pin = &(xors[i % N_LEVEL2 + 1][j]->inpins[1]);
		bbp->next = padder->intable[i];
		padder->intable[i] = bbp;
	}
}
set_inpin_value(&(g[0]->inpins[1]), 0);
```

```
          set_inpin_value(&(p[0]->inpins[1]), 0);
          for (j=0; j<3; j++)
          {
                    set_inpin_value(&(xors[0][j]->inpins[1]), 0);
                    set_inpin_value(&(xors[0][j]->inpins[2]), 0);
          }

          for (i=0; i<N_LEVEL3; i++)
                    padder->outtable[i] = &(ors[i]->outpins);

}

///////////////////////////////////////////////
//Construct and evaluate multiplication circuit
///////////////////////////////////////////////

int mult1[NUMBER_OF_BITS], mult2[NUMBER_OF_BITS];

//This function is for reset multiplication circuit
void set_values_level0()
{
          int i, j;

          for (i=0; i<NUMBER_OF_BITS; i++)
                    for (j=0; j<NUMBER_OF_BITS; j++)
                    {
                              set_blkbox_inpin_value(level0_boxes[i][j].intable[0],
                                                                  mult1[i]);
                              set_blkbox_inpin_value(level0_boxes[i][j].intable[1],
                                                                  mult2[j]);
                    }
}

//initialize level0 circuit
void init_level0()
{
          int i,j;

          for(i=0; i<NUMBER_OF_BITS; i++)
                    for(j=0; j<NUMBER_OF_BITS; j++)
          {
                    set_and_gate( &(level0_boxes[i][j]), 2);
                    set_blkbox_inpin_value(level0_boxes[i][j].intable[0], mult1[i]);
                    set_blkbox_inpin_value(level0_boxes[i][j].intable[1], mult2[j]);
          }
}
```

```
//initialize level1 circuit
void init_level1()
{
        int i,j;

        for(i=0; i<N_LEVEL1; i++)
        {
                set_bcnt5_gate(&(level1_boxes[i]));
                for (j=0; j<NUMBER_OF_BITS; j++)
                        set_blkbox_inpin_value(level1_boxes[i].intable[j], 0);
        }

        for(i=0; i<5; i++)
                for(j=0; j<5; j++)
                {
                        connect_blkbox(level0_boxes[i][j].outtable[0],
                                        level1_boxes[i+j].intable[j]);
                }
}

//initialize level2 circuit
void init_level2()
{
        int i,j;

        for(i=0; i<N_LEVEL2; i++)
        {
                set_bcnt3_gate(&(level2_boxes[i]));
                for(j=0; j<3; j++)
                        set_blkbox_inpin_value(level2_boxes[i].intable[j], 0);
        }

        for(i=0; i<N_LEVEL1; i++)
                for(j=0; j<3; j++)
                {
                        connect_blkbox(level1_boxes[i].outtable[j],
                                        level2_boxes[i+j].intable[j]);
                }
}

//initialize level3 circuit
void init_level3()
{
        int i;
```

```
	set_fast_adder(&level3_box);

	for (i=0; i<N_LEVEL2; i++)
	{
		connect_blkbox(level2_boxes[i].outtable[0],
			level3_box.intable[i]);
		connect_blkbox(level2_boxes[i].outtable[1],
			level3_box.intable[i + N_LEVEL2]);
	}
}


//evaluate multiplication circuit
void evaluate_mult_circuit()
{
	bool t_is_evalued, changed;
	int result;
	int pass = 0,size = 0;
	struct eval_node *p;

	do
	{
		pass++;
		changed = false;
		p = head;
		while (p != NULL)
		{
			t_is_evalued = evaluate_threshold(p->item, &result);
			if (t_is_evalued == true)
			{
				p->valid = true;
				p->value = result;
				if (p->item->outpins.valid == false)
					changed = true;
			}
			p = p->next;
		}

		p = head;
		while (p != NULL)
		{
			if ((p->valid == true) && (p->item->outpins.valid == false))
			{
				p->item->outpins.valid = true;
				p->item->outpins.value = p->value;
			}
```

```
                        p = p->next;
                }
        } while (changed);

        cout <<"The depth of multiplication circuit is:"
                 <<pass-1 <<endl;

        p = head;
        while (p != NULL)
        {
                size++;
                p = p->next;
        }
        cout <<"The size of multiplication circuit is:"
                 <<size <<endl;
}

//Reset multiplication circuit for next calculation to
//avoid realocating the whole circuit.
void reset_mult_circuit()
{
        struct eval_node *p;

        p = head;
        while (p != NULL)
        {
                p->valid = false;
                p->item->outpins.valid = false;
                p = p->next;
        }
}

///////////////////////////////////////////////////
// For main()
//////////////////////////////////////////////////

void init_input()
{
        string s1, s2;
        int i;

        cout << "Enter input s1:";
        cin >> s1;
        cout << "Enter input s2:";
        cin >> s2;
```

```
	for (i=0; i<NUMBER_OF_BITS; i++)
	{
		mult1[i] = s1[NUMBER_OF_BITS-i-1] - '  0'  ;
		mult2[i] = s2[NUMBER_OF_BITS-i-1] - '  0'  ;
	}
}

void print_result()
{
	int i, value;
	bool b;

	cout << "Final result is:";
	for (i=N_LEVEL2; i>=0; i--)
	{
		b = get_blkbox_outpin_value(level3_box.outtable[i], &value);
		assert(b);
		cout << value;
	}

	cout << endl;
}

int menu()
{
	char ch = '  Y'  ;

	cout << endl;
	cout << "Continue?(Y/N)";
	cin >> ch;
	cout << endl;

	while ((ch=='  Y'  ) || (ch=='  y'  ))
	{
		init_input();
		set_values_level0();
		reset_mult_circuit();
		evaluate_mult_circuit();
		print_result();
		cout << endl;
		cout << "Continue?(Y/N)";
		cin >> ch;
		cout << endl;
	}

	return 0;
```

```
}


main()
{
        init_input();

        init_level0();
        init_level1();
        init_level2();
        init_level3();

        evaluate_mult_circuit();
        print_result();

        menu();

        return 0;
}
```

Appendix B: Source code for Deliverable 3

```
//value_gate.cpp

#include <iostream>
#include <cassert>
#include <iomanip>
#include <cmath>
#include <math.h>

using namespace std;

const double EPSILON = 1e-12;
const double PI = 3.14159265358979323846264;
const MaxQubit = 10;


class CComplex_number
{
private:
	double real_part, imaginary_part;
	void post_calc()
	{
		if (fabs(real_part) < EPSILON)
			real_part = 0;
		if (fabs(imaginary_part) < EPSILON)
			imaginary_part = 0;
	}

public:
	CComplex_number()
	{
		real_part = 0;
		imaginary_part = 0;
	}

	CComplex_number(double r, double v)
	{
		real_part = r;
		imaginary_part = v;
		post_calc();
	}

	void set_value(double r, double v)
	{
		real_part = r;
		imaginary_part = v;
		post_calc();
	}

	CComplex_number operator = (CComplex_number& c2)
	{
		real_part = c2.real_part;
		imaginary_part = c2.imaginary_part;
		return CComplex_number(real_part, imaginary_part);
	}

	CComplex_number operator + (CComplex_number c2) const
	{
		double r = real_part + c2.real_part;
		double v = imaginary_part + c2.imaginary_part;
		return CComplex_number(r, v);
	}

	CComplex_number operator * (CComplex_number c2) const
	{
		double r = real_part*c2.real_part -
			imaginary_part*c2.imaginary_part;
		double v = real_part*c2.imaginary_part +
			imaginary_part*c2.real_part;
		return CComplex_number(r, v);
	}

	bool operator == (CComplex_number c2) const
	{
/*
		cout << real_part - c2.real_part << "   ";
		cout << imaginary_part - c2.imaginary_part << "\n";
		return(real_part==c2.real_part &&
			imaginary_part==c2.imaginary_part);

*/
		return(fabs(real_part-c2.real_part)/
			fabs(real_part)+fabs(c2.real_part)<EPSILON &&
			fabs(imaginary_part-c2.imaginary_part)/
			fabs(imaginary_part)+fabs(c2.imaginary_part)<EPSILON);
	}

	void comlex_conjugate()
	{
		imaginary_part = 0.0 - imaginary_part;
	}
	void display()
	{
		cout << setw(4) << real_part;
		if (imaginary_part != 0.0)
		{
			if (imaginary_part == 1.0)
				cout << "+i ";
			else if (imaginary_part == -1.0)
				cout << "-i ";
			else if (imaginary_part < 0.0)
				cout << "-" << (imaginary_part*(-1)) << "i";
			else
				cout << "+" << imaginary_part << "i";
		}
		else
			cout << "   ";
	}

};


class CMatrix
{
private:
	int m_NumOfRow,m_NumOfCol;
	CComplex_number **m_pMatrix;

public:

	CMatrix()
	{}

/*	~CMatrix()
	{
		delete[] m_pMatrix;
	}
*/
	CMatrix(int row, int col) //Constructor
	{
		int i;

		m_NumOfRow = row;
		m_NumOfCol = col;

		m_pMatrix = new (CComplex_number *[m_NumOfRow]);
```

```
        for (i=0; i<m_NumOfRow; i++)
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];
}

void setMatrix (int row, int col, CComplex_number **ptMatrix)
{
        int i;

        m_NumOfRow = row;
        m_NumOfCol = col;

        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];

        for (i=0; i<m_NumOfRow; i++)
                memcpy(m_pMatrix[i], ptMatrix[i],
                        sizeof(CComplex_number)*m_NumOfCol);
}

CMatrix(const CMatrix& matrix) //Copy constructor
{
        int i;

        m_NumOfRow = matrix.m_NumOfRow;
        m_NumOfCol = matrix.m_NumOfCol;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
        {
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];
                memcpy(m_pMatrix[i], matrix.m_pMatrix[i],
                        sizeof(CComplex_number)*m_NumOfCol);
        }
}

CMatrix operator=(const CMatrix& matrix)//Overload assignment
{
        int i;

        m_NumOfRow = matrix.m_NumOfRow;
        m_NumOfCol = matrix.m_NumOfCol;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
        {
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];
                memcpy(m_pMatrix[i], matrix.m_pMatrix[i],
                        sizeof(CComplex_number)*m_NumOfCol);
            }
            return *this;
        }

        void display()
        {
            int i,j;
            for (i=0; i<m_NumOfRow; i++)
            {
                cout << endl;
                for (j=0; j<m_NumOfCol; j++)
                    m_pMatrix[i][j].display();
            }
            cout << endl;
        }

        void CMatrix::mult_matrix(CMatrix m1, CMatrix m2);
        void CMatrix::tensor_products(CMatrix m1, CMatrix m2);
        void CMatrix::transpose(CMatrix m);
        void CMatrix::hermitian_conjugate(CMatrix m);
        void CMatrix::mult_number(CComplex_number cplx, CMatrix m);
        void CMatrix::add_matrix(CMatrix m1, CMatrix m2);

        friend class CGate;
        friend class CValue_gate;
};


void CMatrix::mult_matrix(CMatrix m1, CMatrix m2)
{
        int i, j, k, n;

        assert(m1.m_NumOfCol == m2.m_NumOfRow);
        n = m1.m_NumOfCol;
        m_NumOfRow = m1.m_NumOfRow;
        m_NumOfCol = m2.m_NumOfCol;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
            m_pMatrix[i] = new CComplex_number[m_NumOfCol];

        for (i=0; i<m_NumOfRow; i++)
            for (j=0; j<m_NumOfCol; j++)
            {
                m_pMatrix[i][j].set_value(0, 0);
                for (k=0; k<n; k++)
```

```
                                        m_pMatrix[i][j] =
                                                m_pMatrix[i][j] +
                                                m1.m_pMatrix[i][k] *
                                                m2.m_pMatrix[k][j];
                        }
}


void CMatrix::tensor_products(CMatrix m1, CMatrix m2)
{
        int i, j, m, n;

        m_NumOfRow = m1.m_NumOfRow * m2.m_NumOfRow;
        m_NumOfCol = m1.m_NumOfCol * m2.m_NumOfCol;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];

        for (i=0; i<m1.m_NumOfRow; i++)
                for (j=0; j<m1.m_NumOfCol; j++)
                        for (m=0; m<m2.m_NumOfRow; m++)
                                for (n=0; n<m2.m_NumOfCol; n++)
                                {
                                        m_pMatrix[i*m2.m_NumOfRow+m]
                                                [j*m2.m_NumOfCol+n] =
                                                        m1.m_pMatrix[i][j] *
                                                        m2.m_pMatrix[m][n];
                                }
}

void CMatrix::transpose(CMatrix m)
{
        int i,j;

        m_NumOfRow = m.m_NumOfCol;
        m_NumOfCol = m.m_NumOfRow;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];

        for (i=0; i<m_NumOfRow; i++)
                for (j=0; j<m_NumOfCol; j++)
                        m_pMatrix[i][j] = m.m_pMatrix[j][i];
}

void CMatrix::hermitian_conjugate(CMatrix m)
```

```
{
        int i, j;

        transpose(m);
        for(i=0; i<m_NumOfRow; i++)
                for(j=0; j<m_NumOfRow; j++)
                        m_pMatrix[i][j].comlex_conjugate();
}

void CMatrix::mult_number(CComplex_number cplx, CMatrix m)
{
        int i,j;

        m_NumOfRow = m.m_NumOfRow;
        m_NumOfCol = m.m_NumOfCol;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];

        for(i=0; i<m_NumOfRow; i++)
                for(j=0; j<m_NumOfCol; j++)
                        m_pMatrix[i][j] = m.m_pMatrix[i][j] * cplx;
}

void CMatrix::add_matrix(CMatrix m1, CMatrix m2)
{
        int i,j;

        assert(m1.m_NumOfRow == m2.m_NumOfRow);
        assert(m1.m_NumOfCol == m2.m_NumOfCol);

        m_NumOfRow = m1.m_NumOfRow;
        m_NumOfCol = m1.m_NumOfCol;
        m_pMatrix = new (CComplex_number *[m_NumOfRow]);
        for (i=0; i<m_NumOfRow; i++)
                m_pMatrix[i] = new CComplex_number[m_NumOfCol];

        for(i=0; i<m_NumOfRow; i++)
                for(j=0; j<m_NumOfRow; j++)
                        m_pMatrix[i][j] = m1.m_pMatrix[i][j] + m2.m_pMatrix[i][j];
}

//-----Add on 11/22/02-----------------------------
class CGate
{
protected:
        int m_num_of_qubits;
        CComplex_number **m_ppt;
public:
        CMatrix m_operator_matrix;

        CGate()
        {}

/*      ~CGate()
        {
                delete[] m_ppt;
        }
*/
        CGate(int qubits)
        {
                int i,j;
                int exp_qubits = pow(2,qubits);

                m_num_of_qubits = qubits;

                m_ppt = new CComplex_number *[exp_qubits];
                for(i=0; i<exp_qubits; i++)
                        m_ppt[i] = new CComplex_number[exp_qubits];

                for(i=0; i<exp_qubits; i++)
                        for(j=0; j<exp_qubits; j++)
                                m_ppt[i][j].set_value(0.0, 0.0);

                m_operator_matrix.setMatrix(exp_qubits,exp_qubits,m_ppt);
        }

        void CGate::set_hadamard_gate();
        void CGate::set_identity_gate();
        void CGate::set_fanout_gate(int num_of_target);
        void CGate::set_flip_gate(int qubits, int order[MaxQubit]);
        void CGate::set_increment_gate(int p_qubits, double weights);
        void CGate::set_controlled_increment_gate(int p_qubits,

        double weights);
};

void CGate::set_hadamard_gate()
{
        int i;

        m_num_of_qubits = 1;
```

```
	m_ppt = new CComplex_number *[2];
	for(i=0; i<2; i++)
		m_ppt[i] = new CComplex_number[2];

	m_ppt[0][0].set_value(sqrt(0.5), 0.0);
	m_ppt[0][1].set_value(sqrt(0.5), 0.0);
	m_ppt[1][0].set_value(sqrt(0.5), 0.0);
	m_ppt[1][1].set_value(0-sqrt(0.5), 0.0);

	m_operator_matrix.setMatrix(2,2,m_ppt);
}

void CGate::set_identity_gate()
{
	int i;

	m_num_of_qubits = 1;

	m_ppt = new CComplex_number *[2];
	for(i=0; i<2; i++)
		m_ppt[i] = new CComplex_number[2];

	m_ppt[0][0].set_value(1.0, 0.0);
	m_ppt[0][1].set_value(0.0, 0.0);
	m_ppt[1][0].set_value(0.0, 0.0);
	m_ppt[1][1].set_value(1.0, 0.0);

	m_operator_matrix.setMatrix(2,2,m_ppt);
}

void CGate::set_fanout_gate(int num_of_target)
{
	int i,j;

	m_num_of_qubits = num_of_target + 1;

	int exp_qubits = pow(2,m_num_of_qubits);
	int m = exp_qubits/2;

	m_ppt = new CComplex_number *[exp_qubits];
	for(i=0; i<exp_qubits; i++)
		m_ppt[i] = new CComplex_number[exp_qubits];

	for(i=0; i<exp_qubits; i++)
		for(j=0; j<exp_qubits; j++)
```

```
                        m_ppt[i][j].set_value(0.0, 0.0);

        for (i=0; i<m; i++)
                m_ppt[i][i].set_value(1.0,0.0);
        for (i=m; i<exp_qubits; i++)
        {
                j = m+exp_qubits-1-i;
                m_ppt[i][j].set_value(1.0,0.0);
        }

        m_operator_matrix.setMatrix(exp_qubits,exp_qubits,m_ppt);
}

void CGate::set_flip_gate(int qubits, int order[MaxQubit])
{
        int i,j,k;

        m_num_of_qubits = qubits;

        int exp_qubits = pow(2,m_num_of_qubits);
        int extract[MaxQubit] = {1,2,4,8,16,32,64,128,256,512};

        m_ppt = new CComplex_number *[exp_qubits];
        for(i=0; i<exp_qubits; i++)
                m_ppt[i] = new CComplex_number[exp_qubits];

        for(i=0; i<exp_qubits; i++)
                for(j=0; j<exp_qubits; j++)
                        m_ppt[i][j].set_value(0.0, 0.0);

        for (i=0; i<exp_qubits; i++)
        {
                k=0;
                for (j=0; j<m_num_of_qubits; j++)
                                if(i & extract[order[j]])
                                        k = k | (1<<j);

                m_ppt[i][k].set_value(1.0,0.0);
        }

        m_operator_matrix.setMatrix(exp_qubits,exp_qubits,m_ppt);
}

void CGate::set_increment_gate(int p_qubits, double weights)
{
        int i;
        CMatrix *pt_m;
        CMatrix tmp0(2,2), tmp1(2,2);
        CMatrix multtmp, addtmp;
        CComplex_number cplx;

        m_num_of_qubits = p_qubits;

        pt_m = new CMatrix[p_qubits+1];
        pt_m[0].m_NumOfRow =1;
        pt_m[0].m_NumOfCol =1;
        pt_m[0].m_pMatrix = new CComplex_number *[1];
        pt_m[0].m_pMatrix[0] = new CComplex_number[1];
        pt_m[0].m_pMatrix[0][0].set_value(1.0, 0.0);

        tmp0.m_pMatrix[0][0].set_value(1.0, 0.0);
        tmp0.m_pMatrix[0][1].set_value(0.0, 0.0);
        tmp0.m_pMatrix[1][0].set_value(0.0, 0.0);
        tmp0.m_pMatrix[1][1].set_value(0.0, 0.0);

        tmp1.m_pMatrix[0][0].set_value(0.0, 0.0);
        tmp1.m_pMatrix[0][1].set_value(0.0, 0.0);
        tmp1.m_pMatrix[1][0].set_value(0.0, 0.0);
        tmp1.m_pMatrix[1][1].set_value(1.0, 0.0);

        for (i=1; i<p_qubits+1; i++)
        {
                cplx.set_value(cos(PI/ldexp(1.0,p_qubits-i)*weights),
                                        sin(PI/ldexp(1.0,p_qubits-i)*weights));
                multtmp.mult_number(cplx,tmp1);
                addtmp.add_matrix(tmp0,multtmp);
                pt_m[i].tensor_products(addtmp,pt_m[i-1]);
        }

        m_operator_matrix = pt_m[p_qubits];
}

void CGate::set_controlled_increment_gate(int p_qubits, double weights)
{
        int i,j;
        CGate incr;

        m_num_of_qubits = p_qubits;

        int exp_qubits = pow(2,m_num_of_qubits);
        m_ppt = new CComplex_number *[2*exp_qubits];
        for(i=0; i<2*exp_qubits; i++)
```

```
		m_ppt[i] = new CComplex_number[2*exp_qubits];

	for(i=0; i<2*exp_qubits; i++)
		for(j=0; j<2*exp_qubits; j++)
			m_ppt[i][j].set_value(0.0, 0.0);

	for (i=0; i<exp_qubits; i++)
		m_ppt[i][i].set_value(1.0,0.0);

	incr.set_increment_gate(p_qubits, weights);
//incr.m_operator_matrix.display();
	for (i=exp_qubits; i<2*exp_qubits; i++)
		for(j=exp_qubits; j<2*exp_qubits; j++)
			m_ppt[i][j] = incr.m_operator_matrix.
							m_pMatrix[i-exp_qubits][j-
exp_qubits];

	m_operator_matrix.setMatrix(2*exp_qubits,2*exp_qubits,m_ppt);
}


class CValue_gate : public CGate
{
private:
	int m_p;
	int m_num_of_ancilla;
	double m_threshold;
public:
	CValue_gate()
	{}

	CValue_gate(int qubits, double m)
	{
		m_num_of_qubits = qubits;
		m_p = 1 + ceil(log(1+qubits)/log(2));
		m_num_of_ancilla = (1+qubits)*m_p;
		m_threshold = 0 - m;
	}

	CMatrix CValue_gate::compose(char *str);
	void CValue_gate::set_hadamard_layer(bool first_time);
	void CValue_gate::set_fanout_layer();
	void CValue_gate::set_increment_layer();
	void CValue_gate::set_value_gate();
	void CValue_gate::evaluate_value_gate(char *str);
};
```

```
CMatrix CValue_gate::compose(char *str)
{
        int i,n;
        CMatrix result1[MaxQubit], result2, result3, result;
        CMatrix v0(2,1),v1(2,1);

        v0.m_pMatrix[0][0].set_value(1.0, 0.0);
        v0.m_pMatrix[1][0].set_value(0.0, 0.0);
        v1.m_pMatrix[0][0].set_value(0.0, 0.0);
        v1.m_pMatrix[1][0].set_value(1.0, 0.0);

        n = strlen(str);
        for (i=0; i<n; i++)
        {
                if (str[i] == '  0'  )
                        result1[i] = v0;
                else if (str[i] == '  1'  )
                        result1[i] = v1;
                else
                        assert(false);
        }

        result2 = result1[0];
        for (i=1; i<n; i++)
                result2.tensor_products(result2, result1[i]);

        result3 = v0;
        for (i=1; i<m_num_of_ancilla; i++)
                result3.tensor_products(result3, v0);

        result.tensor_products(result2, result3);
        return result;
}

void CValue_gate::set_hadamard_layer(bool first_time)
{
        int i;
        CGate I,H;
        CMatrix hadamard_layer,tmp;

        I.set_identity_gate();
        H.set_hadamard_gate();

        hadamard_layer = I.m_operator_matrix;
        for (i=1; i<m_num_of_qubits; i++)
```

```
		hadamard_layer.tensor_products(hadamard_layer,

	I.m_operator_matrix);

	tmp = H.m_operator_matrix;
	for (i=1; i<m_p; i++)
		tmp.tensor_products(tmp, H.m_operator_matrix);

	hadamard_layer.tensor_products(hadamard_layer, tmp);

	for (i=0; i<m_num_of_qubits*m_p; i++)
		hadamard_layer.tensor_products(hadamard_layer,

	I.m_operator_matrix);

	if (first_time)
		m_operator_matrix = hadamard_layer;
	else
		m_operator_matrix.mult_matrix(m_operator_matrix,

	hadamard_layer);
}

void CValue_gate::set_fanout_layer()
{
	int i,j,k;
	CGate I, fanout;
	CMatrix fanout_layer, tmp;

	I.set_identity_gate();
	fanout.set_fanout_gate(m_num_of_qubits);

//  num_target = m_num_of_qubits+1-1;
//	num_fanout = m_p;
	tmp = fanout.m_operator_matrix;
	for (i=1; i<m_p; i++)
		tmp.tensor_products(tmp,fanout.m_operator_matrix);

	fanout_layer = I.m_operator_matrix;
	for (i=1; i<m_num_of_qubits; i++)
		fanout_layer.tensor_products(fanout_layer,I.m_operator_matrix);

	fanout_layer.tensor_products(fanout_layer,tmp);

	int total_qubits = m_num_of_qubits + m_num_of_ancilla;
	int *order;
	order = new int[total_qubits];
	for(k=0; k<total_qubits; k++)
		order[k] = k;
	k = 0;
	for(i=0; i<1+m_num_of_qubits; i++)
		for(j=0; j<m_p; j++)
		{
			order[k] = i + j*m_p;
			k++;
		}
//	for(k=0; k<total_qubits; k++)
//		cout << order[k];

	CGate flip;
	flip.set_flip_gate(total_qubits, order);

	fanout_layer.mult_matrix(flip.m_operator_matrix,fanout_layer);

	CMatrix reverse_flip;
	reverse_flip.hermitian_conjugate(flip.m_operator_matrix);

	fanout_layer.mult_matrix(fanout_layer,reverse_flip);

	m_operator_matrix.mult_matrix(m_operator_matrix, fanout_layer);
}

void CValue_gate::set_increment_layer()
{
	int i,j,k;
	CGate increment1, increment_m;
	CMatrix increment_layer, tmp;

	increment1.set_controlled_increment_gate(m_p,1.0);

	increment_layer = increment1.m_operator_matrix;
	for (i=1; i<m_num_of_qubits; i++)
		increment_layer.tensor_products(tmp,

	increment1.m_operator_matrix);

	increment_m.set_increment_gate(m_p,m_threshold);
	increment_layer.tensor_products(increment_layer,

	increment_m.m_operator_matrix);
	int total_qubits = m_num_of_qubits + m_num_of_ancilla;
	int *order;

	order = new int[total_qubits];
	for(k=0; k<total_qubits; k++)
		order[k] = k;
/*	k = 0;
	for(i=0; i<m_num_of_qubits; i++)
	{
		order[k] = i;
		k++;
		for(j=0; j<m_p; j++)
		{
			order[k] = m_num_of_qubits + i*m_p +j;
			k++;
		}
	}
*/
	k = m_p;
	for (i=0; i<m_num_of_qubits; i++)
	{
		for (j=0; j<m_p; j++)
		{
			order[k] = (i+1)*m_p + j;
			k++;
		}
		order[k] = (i+1)*m_p + (m_num_of_qubits-i)*m_p +i;
		k++;
	}
//	for(k=0; k<total_qubits; k++)
//		cout << order[k];

	CGate flip;
	flip.set_flip_gate(total_qubits, order);

	increment_layer.mult_matrix(flip.m_operator_matrix,increment_layer);

	CMatrix reverse_flip;
	reverse_flip.hermitian_conjugate(flip.m_operator_matrix);

	increment_layer.mult_matrix(increment_layer,reverse_flip);

	m_operator_matrix.mult_matrix(m_operator_matrix, increment_layer);
}

void CValue_gate::set_value_gate()
```

```
{
	set_hadamard_layer(true);
	set_fanout_layer();
	set_increment_layer();
	set_fanout_layer();
	set_hadamard_layer(false);
}

void CValue_gate::evaluate_value_gate(char *str)
{
	CMatrix e1,e2,e3;

	set_value_gate();

	e1 = this->compose(str);
//e1.display();
//m_circuit.display();
	e2.mult_matrix(this->m_operator_matrix,e1);
	e2.display();
	e3.transpose(e1);
	e3.display();
	e2.mult_matrix(e3,e2);
	e2.display();
//	e3 = this->decompose(e2);
//	e3.display();
}


int main()
{
//**Test on 11/22/02
//	CGate f1;
//	f1.set_fanout_gate(2);
//	f1.m_operator_matrix.display();

//	int order[3] = {0, 2, 1};
//	CGate g1;
//	g1.set_flip_gate(3,order);
//	g1.m_operator_matrix.display();

//	CGate h1;
//	h1.set_hadamard_gate();
//	h1.m_operator_matrix.display();

//	CGate I1;
//	h1.set_identity_gate();
//	h1.m_operator_matrix.display();

//	CGate d1;
//	d1.set_increment_gate(3);
//	d1.m_operator_matrix.display();

//	CGate c_d1;
//	c_d1.set_controlled_increment_gate(5,4);
//	c_d1.m_operator_matrix.display();

	char str[MaxQubit];
	double m;
	cout << "Input a binary string no longer than " <<MaxQubit
		<< endl;
	cin >> str;
	cout << "Input a threshold value"
		<< endl;
	cin >> m;
	int qubits = strlen(str);
	CValue_gate v1(qubits,m);
	v1.evaluate_value_gate(str);

	return 0;
}
```