CAB402 Programming Paradigms
Quantum Computing

Thanat Chokwijitkul n9234900

Contents

List of Figures

1. Introduction

The theory of computation has been extensively developed during the last few decades. Computers have provided reliable solutions for a myriad of community's seemingly unsolvable problems. Notwithstanding, various complicated problems have been continuously introduced to society as it never stops growing and becomes more complex. Even though technology nowadays has been steady advancing to approach the demands of society, whereas such steady progress has its limitation since many of those problems are intricate to model or appear to require time-intensive solutions. This phenomenon implies to the necessity of a new computing revolution since classical computation no longer has an ability to reach the increased demand.

According to Moore's law, the computational power of computers would dramatically increase approximately every two years (Thompson & Parthasarathy, 2006). The theory can be proven real since the size of transistors has rapidly become smaller to a few nanometres (Nielsen & Chuang, 2000). This results in a higher number of transistors mounted on an integrated circuit. The laws of classical physics do not function with objects with such small size. Thus, quantum computing becomes a next solution to deal with these complex problems.

Quantum computing is the revolution. Although it is a relatively new field of research, this technology has the potential to introduce the world of computing to a new stage where certain computationally intense problems can be solved with a shorter amount of processing time. One can consider quantum computing as the art of utilising all the possibilities that the laws of quantum physics contribute to solving computational problems while classical computers merely use a minuscule subset of these possibilities. However, quantum computers are not a replacement for conventional computers as quantum mechanics only improve the computing efficiency for certain types of computation.

This research report delivers a brief glimpse into the world of quantum computers and the laws of quantum mechanics applied to this entirely new type of computer. It will also introduce the fundamental concepts of the field along with exploring quantum computing in practice using Language-Integrated Quantum Operations (LIQ$Ui|\rangle$). This report concludes with experience evaluation in the quantum computing research in relation to the content of the unit (Programming Paradigms).

2. Early History

In the early 1980s, physicist Richard Freyman recognised that it was not possible for phenomena associated with entangled particles (quantum phenomena) to be efficiently simulated on classical computers (Deutsch, 1982). Freyman was the first who suggested that quantum-mechanical systems might have higher computational power than conventional computers.

In the same decade, Paul Benioff proved that quantum-mechanical systems were at least as powerful as classical computers because Turing machines could be modelled with such system (Yanofsky & Mannucci, 2008).

In 1985, David Deutsch, of Oxford, published the paper "Quantum theory, the Church-Turing principle and the universal quantum computer" (Deutsch, 1985). Deutsch claimed that the universal Turing machine has superior abilities compared with Turing machine, including random number generation, parallelism and physical system simulation with finite-dimensional state spaces.

Another of Deutsch's paper "Quantum computational networks" (Deutsch, 1989) was published in 1989. The article proved that quantum circuits are as powerful as the universal Turing machine. Deutsch also introduced the first truly quantum algorithm in 1990, namely Deutsch's algorithm which later was generalised to the Deutsch-Jozsa algorithm.

In 1993, Andrew Chi-Chih Yao extended Deutsch's paper "Quantum computational networks" (Yao, 1993) in the paper "Quantum circuit complexity" by addressing the complexity of quantum computation according to Deutsch's work. Yao's findings indicated quantum computing researchers to concentrate on quantum circuits instead of quantum Turing machine.

Peter Shor proposed another quantum algorithm in 1994. This algorithm uses the concept of qubit entanglement and superposition for integer factorisation (Bhunia, 2010). In principle, executing the algorithm on a quantum computer would far surpass the efficiency of all classical computers.

The University of California, Harvard University, Massachusetts Institute of Technology and IBM researchers conducted an experiment using nuclear magnetic resonance (NMR) to manipulate quantum data in liquids. The team also developed a 2-bit quantum computer with radio frequency as its input. Afterwards, a new quantum algorithm that executes on quantum computers was introduced by Lov Grover of Bell Laboratories in 1996 by the name Grover's quantum algorithm (Bhunia, 2010).

In 1998, researchers at the University Innsbruck in Austria put the idea of quantum teleportation (IBM, 2014), proposed in 1993, into practice. The theorem demonstrates the concept of entanglement and teleportation. This research is an implication for data transfer and network in the quantum system.

3. Basic Concepts

3.1 Qubits

A bit is the most fundamental building block of the classical model of computer, which has a single logical value, either false or true or simply 0 or 1. In a quantum computer, the quantum bit or qubit also has two computational basis states; 0 and 1, represented by $|0\rangle$ and $|1\rangle$ respectively. However, it can be in a superposition of quantum mechanical two-state systems, meaning the qubit is both in state 0 and 1 simultaneously. A superposition of the qubit can be represented by $\alpha|0\rangle + \beta|1\rangle$ for some α and β such that $|\alpha|^2 + |\beta|^2 = 1$.

In computer system, information is represented in binary form since it is stored in the registers. For example, the non-negative numbers can be represented in binary form as

$$0, 1, 10, 11, 100, 101, 110, 111...$$

A number of bits can determine how many configurations that binary string can represent since $2^n = y$ where n is a number of bits and y is a number of different configurations. For example, a three-bit binary string can represent $2^3 = 8$ numbers including 0 to 7. On the other hand, in quantum computers, the non-negative numbers can be represented in binary form as

$$|0\rangle, |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle \otimes |0\rangle ...$$

In this case, an integer can be written in the form of $|x_{n-1}\rangle \otimes |x_{n-2}\rangle \otimes |x_{n-3}\rangle \otimes ... \otimes |x_1\rangle \otimes |x_0\rangle$ where $|x\rangle$ is a single qubit and $x \in \{0, 1\}$. Therefore, a quantum register of size three must be able to represent positive integers from 0 to 7 as the following:

$$|0\rangle \otimes |0\rangle \otimes |0\rangle \equiv |000\rangle \equiv |0\rangle \ ... \ |1\rangle \otimes |1\rangle \otimes |1\rangle \equiv |111\rangle \equiv |7\rangle$$

However, each qubit can be in both states simultaneously. A superposition of a single qubit can be denoted by $1/\sqrt{2}(|0\rangle + |1\rangle)$. Therefore, a superposition of a quantum register of size three will be

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

This can be represented in binary and decimal forms without the constant respectively as

$$|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle$$

$$\equiv |0\rangle + |1\rangle + |2\rangle + |3\rangle + |4\rangle + |5\rangle + |6\rangle + |7\rangle$$

$$\equiv \sum_{x=0}^{7} |x\rangle$$

3.2 Quantum Gates

In order to explain the concept of quantum gates, the basis states must be represented differently from the previous section. Since matrix transformations will be used to demonstrate the concept of each gate, each qubit state will be represented by any two orthogonal unit column vectors as follows:

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

In this case, the state $|0\rangle$ and $|1\rangle$ are the representations of logical zero and logical one respectively. The qubit's actual state $|\Psi\rangle$ or its superposition can be represented by

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

3.2.1 Quantum Gates

Each quantum gate can be represented by a square matrix. It also needs to be unitary since being unitary will preserve the unit length of the state vector $|\Psi\rangle$ after matrix multiplication and the new state must meet the normalisation criteria $|\alpha|^2 + |\beta|^2 = 1$. Thus, given the 2×2 identity matrix I, the output state will remain in the same state.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$I|\Psi\rangle = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = |\Psi\rangle$$

The matrix multiplication yields a new qubit state which is identical to the input state. This kind of quantum gate can be represented by a single wire as the following:

3.2.2 The NOT Gate

The NOT gate, represented by the negation matrix, flips its input into the opposite value. This rule indicates that if the initial state of the input is 0, and result will be 1 and vice versa. Thus, given the negation matrix X, each state will be inverted into its opposite value after the multiplication process.

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

$$X|\Psi\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}$$

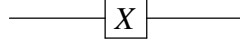The following diagram represents the NOT gate in the quantum circuit:

Figure 3.1: The NOT gate

3.2.3 The Hadamard Gate

This quantum gate is very important since the actual state of each qubit can be in a superposition state. Given the Hadamard matrix H, if the input is 0, its output will be the normalised sum of both basis states (0 and 1). In contrast, if the input is 1, the output will be the normalised difference of both basis states.

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

$$H\Psi = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} \beta \\ \alpha \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} \alpha + \beta \\ \alpha - \beta \end{bmatrix}$$

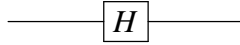The following diagram represents the Hadamard gate in the quantum circuit:

Figure 3.2: The Hadamard gate

3.2.4 Entanglement

Each qubit has its own quantum state. However, two or more qubits can act on one another which leads to the formation of an entangled system. When qubit states are entangled, it needs to be treated as the entire system or overall state, instead of individual quantum state. For example, given a two-bit system, it can be any integer between the range 0 and 3 inclusive as the following:

$$|00\rangle + |01\rangle + |10\rangle + |11\rangle$$

Thus, its normalised superposition can be expressed as

$$|\Psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

This rule also needs to meet the condition $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$. Thus, given an arbitrary set of orthogonal column vectors, each vector represents a possible quantum state, it should yield a new column vector with multi-qubit states.

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \; thus, |\Psi\rangle = \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix}$$

3.2.5 The Controlled-NOT Gate

Unlike single-qubit gates such as the Hadamard gate or the NOT gate, the controlled-NOT or CNOT gate operates on two qubits by flipping the value of the second bit if the first bit is 1, but the value of the second bit remains unchanged if the first bit is 0. The following matrices illustrate how the CNOT gate operate on two qubits.

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

$$CNOT\left|\Psi\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{bmatrix} = \begin{bmatrix} \alpha \\ \beta \\ \delta \\ \gamma \end{bmatrix}$$

In order to make the demonstration more concrete, the state can be represented with a 4×2 matrix representing state values of both qubits.

$$CNOT\left|\Psi\right\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 0 \end{bmatrix}$$

The following diagram illustrates the CNOT gate in the quantum circuit:
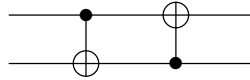
Figure 3.3: The CNOT gate

3.3 Unitary Transformation

As demonstrated in the quantum gates section, unitary transformations can be justified as matrix operations on vectors. This kind of quantum state manipulation can be represented as the following:

$$\left|\Psi\right\rangle \mapsto M \cdot \left|\Psi\right\rangle$$

In this case $\left|\Psi\right\rangle$ is a quantum state and M is a matrix. M is unitary if $M' \cdot M = I$ such that I is the identity matrix. Unitary transformations are reversible and opposed to being information destructive.

3.4 Measurements

Contrary to the concept of unitary transformations, measurements are not reversible and therefore information destructive. However, this kind of operation is effective in retrieving classical information back from quantum information along with interfering or destroying the quantum state.

For example, given the quantum state $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the state $|0\rangle$ with probability α^2 and the state $|1\rangle$ with probability β^2 will be used to measure the actual state $|\Psi\rangle$. The qubit will be determined to be in one of both states until a new transformation occurs.

4. Applications

Since quantum computing is relatively a new area of study, many of its real-world applications are not yet to be seen. Nevertheless, this young field has been being explored by researchers who acknowledge its possibility in advancing the area of high-performance computing. This section delineates potential applications of quantum computing in cryptography along with its classical quantum algorithms, including Shor and Grover's algorithms.

4.1 Cryptography

Public key cryptography has a long history in information security since it is an effectively practical cryptosystem used for data transmission. This approach primarily relies on the complexity of how to crack the communication. This cryptosystem is still considered unconditionally secure since there does not exist a mathematical theorem that prevents eavesdroppers from creating sophisticated revolutionary algorithms to interfere the communication.

Quantum cryptography provides an elegant solution to the dilemma using Quantum Key Distribution (QKD) to exchange a symmetric key over a quantum channel. The security of data transmission can be ensured since quantum error correction codes allow all parties of the communication channel to be able to detect the presence of potential eavesdroppers.

4.2 Algorithms

It is wildly acknowledged that potential quantum-mechanic applications are primarily based on quantum algorithms, computational algorithms run on a quantum computer with highly efficiency improvement over any classical algorithm. This section intends to deliver an overview of quantum algorithmics, focusing on Shor and Grover's algorithms.

4.2.1 Shor's Algorithm

On a classical computer, factoring a large integer is considered a highly time-intensive task. However, Shor indicates that factoring a large integer in polynomial relies on the capacity to find the order of that integer *mod N*. Shor's algorithm follows the processes outlined below:

1. Select a random x, such that $x < N$

2. Compute $f = gcd(x, N)$. This may be done using the Euclidean algorithm. If $f \neq 1$, then return f since it is a prime factor.

3. Find the least r such that $x^r \equiv 1 \ mod \ N$, where r is a repetition period.

4. If either $gcd(x^{r/2} - 1, N) \neq 1$ or $gcd(x^{r/2} + 1, N) \neq 1$, then return it as a prime factor.

5. Otherwise, repeat the process from the first step.

This algorithm consists of two parts, including the classical and quantum parts. According to the algorithm, computing x^r in the third step is necessary. However, this cannot be implemented classically. In this case, Shor utilised the quantum Fourier transform which relies on quantum parallelism to solve the problem.

4.2.2 Grover's Algorithm

Grover's algorithm is one of the most well-known quantum algorithms. It is extremely beneficial to any large technology-based organisation which has its data stored in a database where searching is required. This process of searching is incredibly time-efficient compared with any other classical algorithms. For a search over a set of unordered data, the best classical algorithm requires $O(N)$ time, while Grover's algorithm performs a search in only $O(\sqrt{N})$ operations to find the unique element that satisfies a particular condition, which is a quadratic speedup. Grover's algorithm is as follows:

1. Prepare a quantum register of n qubits, where n is a number of qubits necessary to represent the search space of size $2^n = N$. All qubits need to be initialised to the state $|0\rangle$.

2. Compute $P(x_i)$ where the superposition is calculated.

$$\frac{1}{\sqrt{2^n}} \sum_{x=0}^{n-1} |x, P(x)\rangle$$

3. Invert amplitude of a_j to $-a_j$ such that $P(x_j) = 1$. Then apply inversion to increase amplitude of x_j with $P(x_j) = 1$.

4. Repeat steps 2 to 3 $\frac{\pi}{4}\sqrt{2^n}$ times.

5. Measure the result.

In order to achieve such performance, Grover's algorithm relies on the superposition of qubit states. Additionally, it also utilises the amplitude amplification algorithms which are unique to quantum computing to solve the problem.

5. LIQ$Ui|\rangle$

5.1 Introduction to LIQ$Ui|\rangle$

Language-Integrated Quantum Operations (LIQ$Ui|\rangle$) is a modular software platform for quantum computing developed by the Quantum Architectures and Computation Group (QuArC). It includes an embedded and domain-specific language designed essentially for quantum algorithms with F# as the primary programming language. The software also allows the simulation of quantum algorithms and circuits along with supporting the extraction of a circuit data structure which is useful for circuit optimisation, gate replacement, quantum error correction, rendering or export.

LIQ$Ui|\rangle$ allows the circuit simulation up to 30 qubits on a classical computer with 32 GB of RAM. According to the Microsoft Research's statistics, the largest integer factored on LIQ$Ui|\rangle$, based on Beauregards circuit for Shor's algorithm, is a 13-bit integer. The process required 27 qubits, approximately 0.5 million gates and 5 days runtime to produce the prime factoring result.

5.2 LIQ$Ui|\rangle$ Experiment

This experiment explores functionalities of LIQ$Ui|\rangle$ with Visual Studio as the main Integrated Development Environment (IDE) since it provides a full compilation environment linked to the LIQ$Ui|\rangle$ library (dll), IntelliSense editing and a full debugging environment.

5.2.1 Template

The easiest approach to create a new executable file for LIQ$Ui|\rangle$ is to use the Visual Studio solution template provided by Microsoft Research. Any function with the attribute `[<LQD>]` can be called from terminal with `liquid <function>(<arg>,...)`.

```
[<LQD>]
let __UserSample() =
    show "This module is a good place to put compiled user code"
```

This simple F# code only display the message on the command prompt using the command `liquid __UserSample()`. The following message should be displayed after compiling and running the code:

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/This module is a good place to put compiled user code
0:0000.0/=============== Logging to: Liquid.log closed ================
```

5.2.2   Qubit Measurement

This program simply measures the state of a qubit for a thousand iterations. The following code creates an array of size two to store the results of both states, repetitively measures it and prints the final output.

```
let quantumFunction (qubits:Qubits) =
    M qubits // M: Built-in measurement function

[<LQD>]
let Liquid() =
    let stats = Array.create 2 0
    let ket = Ket(1)
    for i in 0..999 do
        let qubits = ket.Reset(1)
        quantumFunction qubits
        let value = qubits.Head.Bit.v
        stats.[value] <- stats.[value] + 1
    show "Measured: 0 = %d 1 = %d" stats.[0] stats.[1]
```

The result should be obvious since the initial state of the qubit is 0 and there is no other quantum gate involved in the circuit except for its own quantum gate (identity).

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.1/Measured: 0 = 1000 1 = 0
0:0000.1/=============== Logging to: Liquid.log closed ================
```

5.2.3   The Hadamard Gate

This program is the first piece of actual quantum code. It uses the Hadamard gate to transform a given input state into its superposition. The code should remain the same as the previous section but the quantum function should be modified as follows:

```
let quantumFunction (qubits:Qubits) =
    H qubits // H: Built-in Hadamard gate function
    M qubits // M: Built-in measurement function
```

The result is different with 50% probability of the state 0 or 1.

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.1/Measured: 0 = 527 1 = 473
0:0000.1/=============== Logging to: Liquid.log closed ================
```

5.2.4 User-defined Gate

This program uses the user-defined gate, namely *rotationX*. This gate can be used to rotate the qubit around any arbitrary angle in the *X* axis.

```
let rotationX (theta:float) (qs:Qubits) =
    let gate (theta:float) =
        let name = "Rx" + theta.ToString("F2")
        new Gate(
            Name = name,
            Help = sprintf "Rotate in X by: %f" theta,
            Mat = (
                let phi = theta / 2.0
                let c = Math.Cos phi
                let s = Math.Sin phi
                CSMat(2,[0,0,c,0.;0,1,-s,0.;1,0,s,0.;1,1,c,0.])),
            Draw = "\\gate{" + name + "}"
            )
    (gate theta).Run qs
```

The code should remain the same as the previous section but the quantum function should be modified as follows:

```
let quantumFunction (qubits:Qubits) =
    rotationX (Math.PI/2.0) qubits
    M qubits
```

The result should be similar with the result from the circuit with the Hadamard gate.

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.1/Measured: 0 = 501 1 = 499
0:0000.1/=============== Logging to: Liquid.log closed ================
```

5.2.5 Entanglement

All the previous programs only perform the quantum function on a single qubit. However, two or more qubits are required in this case in order to simulate the quantum entanglement. The program will take *n* qubits as its input, transform each qubit to its superposition, then perform entanglement with the CNOT gate and measure the output state (See Appendix A.1 for the source code).

```
let quantumFunction (qubits:Qubits) =
    rotationX (Math.PI/2.0) qubits
    for qubit in qubits.Tail do
        CNOT [qubits.Head; qubit]
    M >< qubits // Measures all the qubits

[<LQD>]
let Liquid(n:int) =
    let stats = Array.create 2 0
```

14

```
let ket = Ket(n)
let circuit = Circuit.Compile quantumFunction ket.Qubits
show "Circuit 1:"
circuit.Dump()
circuit.RenderHT("Circuit1")
let circuit = circuit.GrowGates(ket)
show "Circuit 2:"
circuit.Dump()
circuit.RenderHT("Circuit2")
for i in 0..999 do
    let qubits = ket.Reset(n)
    circuit.Run qubits
    let value = qubits.Head.Bit.v
    stats.[value] <- stats.[value] + 1
show "Measured: 0 = %d 1 = %d" stats.[0] stats.[1]
```

The Entangle function is the extended version of the LIQUID function (the function with the [<LQD>] attribute). This function defines two additional quantum circuits. The first circuit is used to compile the quantum function with the defined ket as an input. The second circuit optimises the program efficiency by combining all the gates on the first circuits into a single gate.

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.1/Circuit 1:
0:0000.1/Writing: Circuit1.htm (split=100.00% scale=100.00%)
0:0000.1/Writing: Circuit1.tex (split=100.00% scale=100.00%)
0:0000.1/    Doing columns:  0 - 22
0:0000.1/Circuit 2:
0:0000.1/Writing: Circuit2.htm (split=100.00% scale=100.00%)
0:0000.1/Writing: Circuit2.tex (split=100.00% scale=100.00%)
0:0000.1/    Doing columns:  0 - 14
0:0000.1/Measured: 0 = 484 1 = 516
0:0000.1/=============== Logging to: Liquid.log closed ================
```

The Dump function creates a log file containing the sequence of the applications of gates (See Appendix A.2 for the sample log file).

The RenderHT function produce a diagram of the current quantum circuit. In this case, it will create `htm` and `tex` files illustrating two diagrams, including the initial state of the quantum circuit and its state after being optimised.
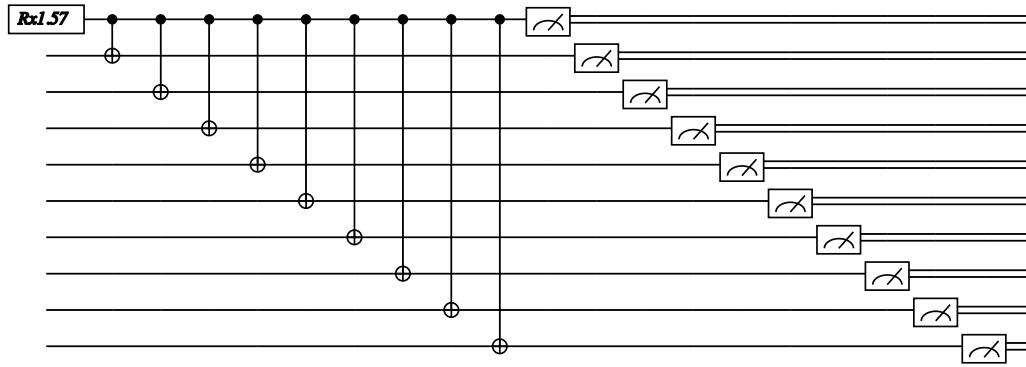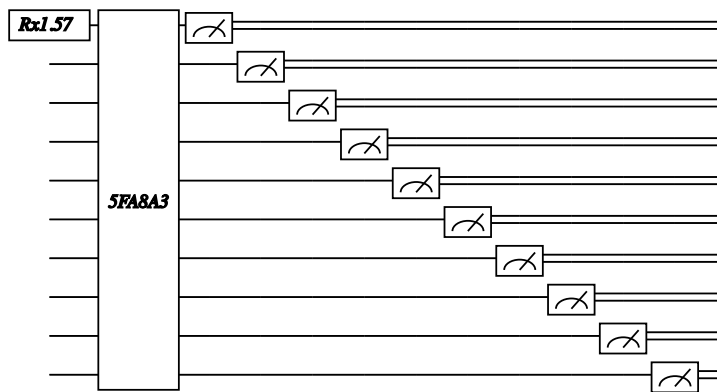
Figure 5.1: The initial state of the circuit

Figure 5.2: The circuit after being optimised

5.3   Built-in Samples

LIQ$Ui|\rangle$ contains a number of built-in samples for some of the specific quantum algorithms, including Shors algorithm, quantum teleportation, quantum error correction, quantum linear algebra, quantum associated memory and the simulation of the ground state energy of a molecule. These executable samples gives users a glimpse of what LIQ$Ui|\rangle$ can do and application areas that it has been applied to. This section focuses on three built-in examples which are relevant to the concepts and applications explained in this report, including entanglement, Shors algorithm and quantum associative memory (utilising Grovers algorithm).

5.3.1   Entanglement

This entanglement function produces detailed statistics the duration the simulator takes to perform various operations. It can be called with liquid __Entangle1(n), where *n* is a number of qubits to be entangled (typically 10 to 22). The sample result is as the following:

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/ Secs/Op  S/Qubit  Mem(GB) Operation
0:0000.0/ -------  -------  ------- ---------
0:0000.0/   0.035    0.035    0.019 Created single state vector
```

```
0:0000.0/    0.039    0.039    0.019 Did Hadamard
0:0000.0/    0.004    0.004    0.019   Did CNOT:  1
0:0000.0/    0.009    0.005    0.019   Did CNOT:  2
0:0000.0/    0.019    0.006    0.020   Did CNOT:  3
0:0000.0/    0.026    0.007    0.020   Did CNOT:  4
0:0000.0/    0.038    0.008    0.020   Did CNOT:  5
0:0000.0/    0.047    0.008    0.020   Did CNOT:  6
0:0000.0/    0.054    0.008    0.020   Did CNOT:  7
0:0000.0/    0.061    0.008    0.020   Did CNOT:  8
0:0000.0/    0.068    0.008    0.020   Did CNOT:  9
0:0000.0/    0.013    0.001    0.020 Did Measure
0:0000.0/=============== Logging to: Liquid.log closed ================
```

Apart from above example, LIQU$i|\rangle$ also provides another two functions which perform quantum entanglement on qubits, namely __Entangle2 and __Entangles. The __Entangle2 function operates in the same way with __Enytangle1. However, it also produces diagrams illustrating the circuits in the form of htm and tex files. The __Entangles function runs 100 entanglement tests on 16 qubits. At the end of the process, half of the qubits should be in state 1 and another half should be in the opposite state.

5.3.2   Shor's Algorithm

Shor's polynomial-time prime factorisation algorithm is one of the sophisticated built-in tests among other samples. The command liquid __Shor(n,true) can be used to run the algorithm. The command takes two parameters. The first is the integer to be factored and the second is the option whether to optimise (by combining) the quantum circuit. The following is the result of prime factorisation using Shor's algorithm with 21 as an input:

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/======== Doing Shor Round =========
0:0000.0/      21 = N = Number to factor
0:0000.0/       2 = a = coPrime of N
0:0000.0/       5 = n = number of bits for N
0:0000.0/      32 = 2^n
0:0000.0/      13 = total qubits
0:0000.0/      22 = starting memory (MB)
0:0000.0/  96.00% = prob of random result (983/1024)
0:0000.0/  43.07% = prob of Shor (worst case)
0:0000.0/         - Compiling circuit
0:0000.0/0.001330 = mins for compile
0:0000.0/   13890 = cnt of gates
0:0000.0/    3873 = cache hits
0:0000.0/     121 = cache misses
0:0000.0/      22 = compiled memory (MB)
0:0000.0/         - Wrapping circuit pieces
0:0000.1/       8 = wires have possibles: 66 (prv= 0GB did= 0 big= 0)
0:0000.1/       9 = wires have possibles: 57 (prv= 0GB did= 9 big= 4)
0:0000.1/      10 = wires have possibles: 52 (prv= 0GB did= 14 big= 39)
```

```
0:0000.1/          11 = wires have possibles: 51 (prv= 0GB did= 15 big= 81)
0:0000.2/          12 = wires have possibles: 48 (prv= 0GB did= 18 big= 124)
0:0000.2/          13 = wires have possibles: 47 (prv= 0GB did= 19 big= 166)
0:0000.2/     14 = Ran out of wires
0:0000.2/          MM: g:    19 b:   212  12=1 11=3 10=1 9=5 8=9
0:0000.2/0.140439 = mins for growing gates
0:0000.2/    470 = cnt of gates
0:0000.2/     52 = grown memory (MB)
0:0000.2/        Bit:   9 [MB:   49 m=1]
0:0000.3/        Bit:   8 [MB:   50 m=0]
0:0000.4/        Bit:   7 [MB:   49 m=1]
0:0000.6/        Bit:   6 [MB:   50 m=0]
0:0000.6/... compiling MB=     47 cache(15806,126) GC:8519
0:0000.8/        Bit:   5 [MB:   52 m=1]
0:0001.1/        Bit:   4 [MB:   52 m=0]
0:0001.2/        Bit:   3 [MB:   52 m=1]
0:0001.2/... compiling MB=     49 cache(27741,126) GC:6400
0:0001.4/        Bit:   2 [MB:   51 m=0]
0:0001.6/        Bit:   1 [MB:   53 m=1]
0:0001.8/        Bit:   0 [MB:   55 m=0]
0:0001.8/0.046182 = mins for running
0:0001.8/ 107.464 = Total Elapsed time (seconds)
0:0001.8/     13 = Max Entangled
0:0001.8/      0 = Gates Permuted
0:0001.8/    371 = State Permuted
0:0001.8/    128 = None  Permuted
0:0001.8/    341 = m = quantum result
0:0001.8/0.333008 = c = 341/1024 =~ 1/3
0:0001.8/        Odd denominator, expanding
0:0001.8/      3 = 6/2 = exponent
0:0001.8/      9 = 2^3 + 1 mod 21
0:0001.8/      7 = 2^3 - 1 mod 21
0:0001.8/      7 = factor = max(3,7)
0:0001.8/CSV N a m den f1 f2 good,21,2,341,6,7,3,1
0:0001.8/GOT:  21=   7x   3 co=    2 n,q=21,13 mins=1.79 SUCCESS!!
0:0001.8/=============== Logging to: Liquid.log closed ================
```

According to the result, the quantum circuit had 13,890 gates that were reduced to 470 grown gates. The execution time is approximately 107 seconds on a Windows virtual machine with only 4 GB of RAM. However, the algorithm run successfully with the result $21 = 7 \times 3$.

5.3.3   Quantum Associative Memory

This built-in sample is the quantum associative memory algorithm developed by Dan Ventura and Tony Martinez. The algorithm applies the theory of quantum physics in creating a quantum associative memory with an exponentially expanded capacity in the number of neurons. The first set of output is a number of keyvalue pairs:

```
0:0000.0/Writing: QuAM1.htm (split=10.00% scale=70.00%)
0:0000.3/======= DUMPSTATE: Loaded patterns
0:0000.3/0x0a:  6.3% (0x001400)
0:0000.3/0x1a:  6.3% (0x003400)
0:0000.3/0x24:  6.3% (0x004800)
0:0000.3/0x3d:  6.3% (0x007a00)
0:0000.3/0x4b:  6.3% (0x009600)
0:0000.3/0x5e:  6.3% (0x00bc00)
0:0000.3/0x64:  6.3% (0x00c800)
0:0000.3/0x75:  6.3% (0x00ea00)
0:0000.3/0x8c:  6.3% (0x011800)
0:0000.3/0x9d:  6.3% (0x013a00)
0:0000.3/0xa6:  6.3% (0x014c00)
0:0000.3/0xb9:  6.3% (0x017200)
0:0000.3/0xcf:  6.3% (0x019e00)
0:0000.3/0xdd:  6.3% (0x01ba00)
0:0000.3/0xeb:  6.3% (0x01d600)
0:0000.3/0xf9:  6.3% (0x01f200)
```

It is obvious that the circuit stored all the elements with equal probability. The next set of result demonstrates how the system used Grover's search algorithm to search for the element with key 6:

```
0:0000.3/Writing: QuAM2.htm (split=33.33% scale=70.00%)
0:0000.5/    Grover[ 0]: 0x64:  7.7% (0x00c800)
0:0000.5/    Grover[ 1]: 0x64: 12.4% (0x00c800)
0:0000.5/    Grover[ 2]: 0x64: 16.9% (0x00c800)
0:0000.5/    Grover[ 3]: 0x64: 19.2% (0x00c800)
0:0000.5/    Grover[ 4]: 0x64: 18.2% (0x00c800)
0:0000.5/    Grover[ 5]: 0x64: 14.5% (0x00c800)
0:0000.5/    Grover[ 6]: 0x64:  9.6% (0x00c800)
0:0000.5/    Grover[ 7]: 0x64:  5.5% (0x00c800)
0:0000.5/    Grover[ 8]: 0x64:  2.9% (0x00c800)
0:0000.5/    Grover[ 9]: 0x1a:  3.7% (0x003400)
0:0000.5/    Grover[10]: 0x1a:  3.8% (0x003400)
0:0000.5/    Grover[11]: 0x64:  3.2% (0x00c800)
0:0000.5/    Grover[12]: 0x64:  6.0% (0x00c800)
0:0000.5/    Grover[13]: 0x64: 10.3% (0x00c800)
0:0000.5/    Grover[14]: 0x64: 15.1% (0x00c800)
0:0000.5/    Grover[15]: 0x64: 18.6% (0x00c800)
0:0000.5/    Grover[16]: 0x64: 19.1% (0x00c800)

0:0000.5/======= DUMPSTATE: Searched for key: 6
0:0000.5/0x64: 19.1% (0x00c800)
0:0000.5/0x0a:  3.1% (0x001400)
0:0000.5/0x1a:  3.1% (0x003400)
0:0000.5/0x24:  3.1% (0x004800)
0:0000.5/0x3d:  3.1% (0x007a00)
```

```
0:0000.5/0x4b:  3.1% (0x009600)
0:0000.5/0x5e:  3.1% (0x00bc00)
0:0000.5/0x75:  3.1% (0x00ea00)
0:0000.5/0x8c:  3.1% (0x011800)
0:0000.5/0x9d:  3.1% (0x013a00)
0:0000.5/0xa6:  3.1% (0x014c00)
0:0000.5/0xb9:  3.1% (0x017200)
0:0000.5/0xcf:  3.1% (0x019e00)
0:0000.5/0xdd:  3.1% (0x01ba00)
0:0000.5/0xeb:  3.1% (0x01d600)
0:0000.5/0xf9:  3.1% (0x01f200)
0:0000.5/0x60:  1.7% (0x00c000)
0:0000.5/0x61:  1.7% (0x00c200)
0:0000.5/0x62:  1.7% (0x00c400)
0:0000.5/0x63:  1.7% (0x00c600)
```

The result shows how Grover's algorithm performed the search through the optimal probability of finding the targeted key/value pair.

References

Bhunia, C. T. (2010). *Introduction to quantum computing*. New Age International.

Deutsch, D. (1982). Quantum computation. *Physics World*.

Deutsch, D. (1985). Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London, Series A*, *400*(1818), 97-117.

Deutsch, D. (1989). Quantum computational networks. *Proceedings of the Royal Society of London, Series A*, *425*(1868), 73-90.

IBM. (2014). *Quantum teleportation*. `http://researcher.watson.ibm.com/researcher/view_group.php?id=2862`.

Nielsen, M. A., & Chuang, I. L. (2000). *Quantum computation and quantum information*. Cambridge University Press.

Thompson, S. E., & Parthasarathy, S. (2006). Moore's law: the future of si microelectronics. *Materials Today*, *9*(6), 20-25.

Wecker, D., & Svore, K. M. (2014). *LIQUi—¿: A Software Design Architecture and Domain-Specific Language for Quantum Computing*. Retrieved from `arXiv:1402.4467v1`

Yanofsky, N. S., & Mannucci, M. A. (2008). *Quantum computing for computer scientists*. Cambridge University Press.

Yao, A. C.-C. (1993). Quantum circuit complexity. *Proceedings of 34th Annual IEEE Symposium on Foundations of Computer Science*, 352-361.

Appendix A – LIQ*Ui*|⟩ Experiment

A.1 Entanglement Source Code

```
#if INTERACTIVE
#r @"..\bin\Liquid1.dll"
#else
namespace Microsoft.Research.Liquid // Namespace
#endif

open System                         // Support libraries
open Microsoft.Research.Liquid      // Liquid libraries
open Util                           // General utilites
open Operations                     // Basic gates and operations

// Rotates a qubit around any arbitrary angle in the X axis
let rotationX (theta:float) (qs:Qubits) =
    let gate (theta:float) =
        let name = "Rx" + theta.ToString("F2")
        new Gate(
            Name = name,
            Help = sprintf "Rotate in X by: %f" theta,
            Mat = (
                let phi = theta / 2.0
                let c = Math.Cos phi
                let s = Math.Sin phi
                CSMat(2,[0,0,c,0.;0,1,-s,0.;1,0,s,0.;1,1,c,0.])),
            Draw   = "\\gate{" + name + "}"
            )
    (gate theta).Run qs

// Quantum funtion
let quantumFunction (qubits:Qubits) =
    rotationX (Math.PI/2.0) qubits
    for qubit in qubits.Tail do
        CNOT [qubits.Head; qubit]
    M >< qubits // Measures all the qubits
```

```
[<LQD>]
let Liquid(n:int) =
    let stats = Array.create 2 0
    let ket = Ket(n)
    let circuit = Circuit.Compile quantumFunction ket.Qubits
    show "Circuit 1:"
    circuit.Dump()
    circuit.RenderHT("Circuit1")
    let circuit = circuit.GrowGates(ket)
    show "Circuit 2:"
    circuit.Dump()
    circuit.RenderHT("Circuit2")
    for i in 0..999 do
        let qubits = ket.Reset(n)
        circuit.Run qubits
        let value = qubits.Head.Bit.v
        stats.[value] <- stats.[value] + 1
    show "Measured: 0 = %d 1 = %d" stats.[0] stats.[1]
```

A.2 LIQ*Ui*|⟩ Log File

```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.1/Circuit 1:
0:0000.1/SEQ
0:0000.1/  APPLY
0:0000.1/    GATE Rx1.57 is a Rotate in X by: 1.570796 (Normal)
0:0000.1/      0.7071 -0.7071
0:0000.1/      0.7071 0.7071
0:0000.1/    WIRE(Id:0)
0:0000.1/    WIRE(Id:1)
0:0000.1/    WIRE(Id:2)
0:0000.1/    WIRE(Id:3)
0:0000.1/    WIRE(Id:4)
0:0000.1/    WIRE(Id:5)
0:0000.1/    WIRE(Id:6)
0:0000.1/    WIRE(Id:7)
0:0000.1/    WIRE(Id:8)
0:0000.1/    WIRE(Id:9)
0:0000.1/  APPLY
0:0000.1/    GATE CNOT is a Controlled NOT (Normal)
0:0000.1/      1 0 0 0
0:0000.1/      0 1 0 0
0:0000.1/      0 0 0 1
0:0000.1/      0 0 1 0
```

```
0:0000.1/    WIRE(Id:0)
0:0000.1/    WIRE(Id:1)
0:0000.1/  APPLY
0:0000.1/    GATE CNOT is a Controlled NOT (Normal)
0:0000.1/     1 0 0 0
0:0000.1/     0 1 0 0
0:0000.1/     0 0 0 1
0:0000.1/     0 0 1 0
0:0000.1/    WIRE(Id:0)
0:0000.1/    WIRE(Id:2)
.
.
.

0:0000.1/Circuit 2:
0:0000.1/SEQ
0:0000.1/  APPLY
0:0000.1/    GATE Rx1.57 is a Rotate in X by: 1.570796 (Normal)
0:0000.1/     0.7071 -0.7071
0:0000.1/     0.7071 0.7071
0:0000.1/    WIRE(Id:0)
0:0000.1/    WIRE(Id:1)
0:0000.1/    WIRE(Id:2)
0:0000.1/    WIRE(Id:3)
0:0000.1/    WIRE(Id:4)
0:0000.1/    WIRE(Id:5)
0:0000.1/    WIRE(Id:6)
0:0000.1/    WIRE(Id:7)
0:0000.1/    WIRE(Id:8)
0:0000.1/    WIRE(Id:9)
0:0000.1/  APPLY
0:0000.1/    GATE 5FA8A3 is a 5FA8A3 (Normal)
0:0000.1/      0  0 1
0:0000.1/      1  1 1
0:0000.1/      2  2 1
0:0000.1/      3  3 1
0:0000.1/      4  4 1
0:0000.1/      5  5 1
0:0000.1/      6  6 1
0:0000.1/      7  7 1
.
.
.
0:0000.1/Measured: 0 = 484 1 = 516
0:0000.1/=============== Logging to: Liquid.log closed ================
```