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 (Rieffel & Polak, 2000). 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 (Slingerland, 2016). However, quantum computers are not a replacement for conventional computers since 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, proposed in 1993 (IBM, 2014), 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 (Bone & Castro, 1997), 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 (Quantiki, 2015), meaning the qubit can be in both 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 the computer system, information is represented in binary form since it is stored in the registers (Ekert, Hayden, & Inamori, 2008). For example, the non-negative numbers can be represented in binary form as

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

The number of bits can determine how many configurations that binary string can represent since $2^n = y$ where n is the number of bits and y is the 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 is able to represent positive integers from 0 to 7 as the following:

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

According to the theory of two-state quantum system in quantum mechanics, each qubit can be in both states at the same time. 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 preserves the unit length of the state vector $|\Psi\rangle$ after matrix multiplication (Strubell, 2011) 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:

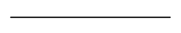

Figure 3.1: A quantum gate

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, the result will be 1 and if the initial state of the input is 1, the result will be 0. 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:

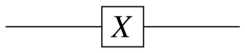

Figure 3.2: The NOT gate

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 $((|0\rangle + |1\rangle)/\sqrt{2})$. In contrast, if the input is 1, the output will be the normalised difference of both basis states $((|0\rangle - |1\rangle)/\sqrt{2})$.

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

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

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

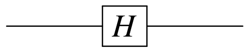

Figure 3.3: The Hadamard gate

3.2.4 Entanglement

In order to explain the CNOT gate, it is necessary to understand the concept of quantum entanglement. As mentioned before, 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 (Dahlsten, 2005). When qubit states are entangled, it needs to be treated as the entire system or overall state, instead of an 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.

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

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

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

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

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

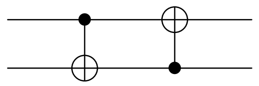

Figure 3.4: The CNOT gate

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:

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

In this case $|\Psi\rangle$ 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 (Steinruecken, 2004).

3.4 Measurements

Contrary to the concept of unitary transformations, measurements are not reversible and therefore information destructive (Steinruecken, 2004). This kind of operation is effective in retrieving classical information back from quantum information but may also interfere or destroy the quantum state (Gagen, 1993).

For example, given the quantum state $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the state $|0\rangle$ with a probability α^2 and the state $|1\rangle$ with a 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 ($|0\rangle$ or $|1\rangle$) until a new transformation occurs. To make the example more concrete, given the qubit state $|\Psi\rangle = (+0.6) + (-0.8)$, the probability of the result being $|0\rangle$ is $\alpha^2 = 0.6^2 = 36\%$ and the probability of the result being $|1\rangle$ is $\beta^2 = -0.8^2 = 64\%$.

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 (Salomaa, 1990). 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 (CQC, 2016).

Quantum cryptography provides an elegant solution to the dilemma using Quantum Key Distribution (QKD) to exchange a symmetric key over a quantum channel (CSA, 2015). 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

Factoring a large integer on a classical computer is considered a highly time-intensive task. However, it would be different in the case of a quantum computer since each qubit used for factoring can be in a superposition, which implies that n qubits can handle 2^n states simultaneously. By increasing the number of qubits used in the prime factorisation process, this quantum quality can dramatically reduce the time used to factor an integer. However, the algorithm needs at least $2l$ qubits, where $l = (\log_2(N))$ and N is the integer to be factored (Cao

& Liu, 2015). According to the research on quantum computation and Shor's factoring algorithm (Wolf, 1999), the complexity of Shor's algorithm is $O((\log n)^2(\log\log n)(\log\log\log n))$, which can be alternatively expressed as $\tilde{O}((\log n)^2)$. Shor also indicates that the efficiency of factoring a large integer in polynomial time relies on the efficiency of modular exponentiation and the quantum Fourier transform (QFT) (See Appendix A for the definition and an example of the QFT). 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 by 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. The classical part includes modular exponentiation using repeated squarings and the quantum part includes the concept of the QFT, which heavily relies on quantum parallelism to solve the problem (Lin, 2013).

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 (Strubell, 2011), which is a quadratic speedup. Grover's algorithm can be summarised 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 the steps 2 and 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 (Strubell, 2011).

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) (Microsoft, 2016). 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 (Microsoft, 2016; Wecker & Svore, 2014).

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 Beauregard's 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 (Microsoft, 2016).

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 actual quantum program. It uses the Hadamard gate to transform a given input state into a 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 to 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 B.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
    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 Liquid 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 B.2 for a sample log file).

The RenderHT function produces 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.

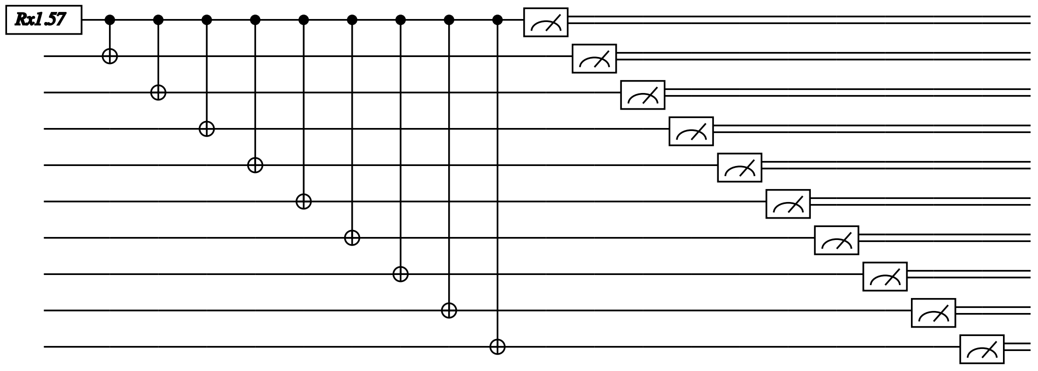

Figure 5.1: The initial state of the circuit

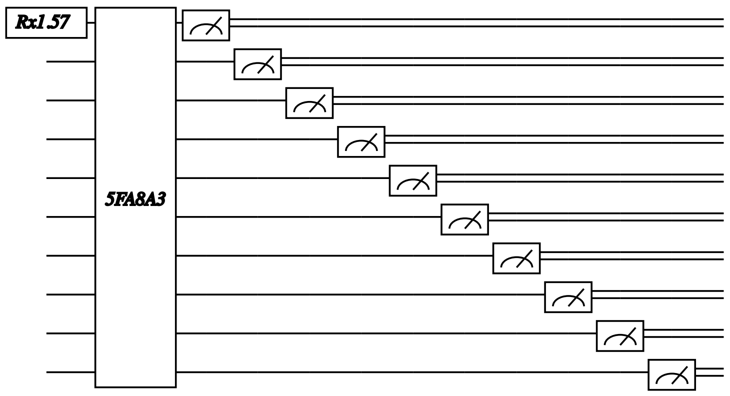

Figure 5.2: The circuit after being optimised

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 (Wecker, 2016). 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) (See Appendix C for the list of all built-in samples).

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, LIQ$Ui|\rangle$ also provides another two functions which perform quantum entanglement on qubits, namely `__Entangle2` and `__Entangles`. The `__Entangle2` function operates in the same way with `__Entangle1`. However, it also produces diagrams illustrating the circuits in the form of `htm` and `tex` files. The `__Entangles` function runs a hundred 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 (Ventura & Martinez, 1998). 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.

6. Evaluation

Practical quantum computing requires a toolchain extending from the level of abstract algorithm down to physical particles (Valiron, Ross, Selinger, Alexander, & Smith, 2015). It is evident that experimenting quantum computing on a real quantum computer falls beyond the scope of this research report. This is the reason why LIQ$Ui|\rangle$ has been selected as a software architecture for quantum computing experiment since it allows users to express a quantum algorithm written in a high-level programming language (F# in this case) into the low-level machine-readable instructions for a quantum computer (Microsoft, 2016).

Ideally, a programming language to be translated into the machine instructions for a quantum device ought to allow developers to implement quantum algorithms in a human-comprehensible way (Valiron et al., 2015). This means the language should support a level of abstraction that is close to the way one naturally thinks about how the algorithm should be implemented. For example, if the quantum algorithm is primarily expressed by a computationally mathematical formula, the programming language used to implement that kind of algorithm should support such expression. Fortunately, F# is a multi-paradigm programming language with a sufficient level of flexibility to fulfill that requirement.

By using LIQ$Ui|\rangle$ as a quantum computing simulator, users are able to simulate various quantum algorithms along with other quantum operations, such as expressing quantum circuits using a high-level functional language (F#), extracting circuit data structure for circuit optimisation, gate replacement, quantum error correction, circuit rendering and export (Microsoft, 2016). In essence, LIQ$Ui|\rangle$ allows users to experiment and understand how quantum computing can be applied to various real-world applications.

7. Reflection

The computational power of quantum computing is based on the laws of quantum physics and various quantum phenomena, such as quantum entanglement, quantum parallelism or quantum interference (Valiron et al., 2015). Those events are radically different from what can be encountered in classical computing. Consequently, in order to comprehend these attributes and designs algorithms that outperform the well-known classical counterparts, it is mandatory to understand the fundamental principles of quantum computing as well as why some certain problems cannot be solved by a conventional digital computer.

Apart from the comprehension of the principles which quantum mechanic is based on, it is also necessary to understand the programming aspect of how to express a quantum algorithm in the form of a high-level program which will be translated into a set of instructions for a quantum computer. Even though LIQ$Ui|\rangle$ already enables easy programming and compilation for the simulation of quantum algorithms and circuits, having knowledge of the functional programming paradigm also results in increased ease in the algorithm implementation process since the simulator adopts a high-level functional programming language (F#) as its host language.

Most of the modern quantum programming languages may be taxonomically classified into the imperative paradigm, the functional paradigm and others (Sofge, 2008). However, many of recent quantum programming developments have focused on utilising functional quantum programming languages instead of imperative quantum programming languages due to its succinctness (Altenkirch & Grattage, 2005), which allows developers to express algorithms more clearly. Hence, it is valuable to understand the concept of functional programming as an alternative way of thinking and expressing algorithms for either classical or quantum computation, apart from programming in the imperative style which most of the developers have been accustomed.

References

Altenkirch, T., & Grattage, J. (2005). *A Functional Quantum Programming Language.* Nottingham University.

Bacon, D. (2006). *The Quantum Fourier Transform and Jordans Algorithm.* University of Washington.

Bhunia, C. T. (2010). *Introduction to Quantum Computing*. New Age International.

Bone, S., & Castro, M. (1997). *A Brief History of Quantum Computing.* `http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol4/spb3/`.

Cao, Z., & Liu, L. (2015). *Comment on Realization of a Scalable Shor Algorithm.* Shanghai University.

circuit Tutorial, Q. (2004). *Quantum Associative Memory.* `arXiv:quant-ph/0406003v2`.

CQC. (2016). *Secure Quantum Communications.* `http://www.cqc2t.org/research/secure_quantum_communications`.

CSA. (2015). *What Is Quantum Key Distribution?* `http://www.quintessencelabs.com/wp-content/uploads/2015/08/CSA_What-is-Quantum-Key-Distribution_QSS.pdf`.

Dahlsten, O. (2005). *An Introduction to Entanglement Theory.* Imperial College London.

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.

Ekert, A., Hayden, P., & Inamori, H. (2008). Basic Concepts in Quantum Computation. *Les Houches - Ecole dEte de Physique Theorique Coherent atomic matter waves*, 661-701.

Gagen, M. J. (1993). *Quantum Measurement Theory and the Quantum Zeno Effect* (Unpublished doctoral dissertation). University of Queensland.

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

Lin, F. X. (2013). *Shors Algorithm and the Quantum Fourier Transform.* McGill University.

Microsoft. (2016). *Language-Integrated Quantum Operations: LIQUi$|\rangle$.* `http://research.microsoft.com/en-US/projects/liquid/default.aspx`.

Nielsen, M. A., & Chuang, I. L. (2000). *Quantum Computation and Quantum Information*. Cambridge University Press.

Quantiki. (2015). *What Is Quantum Computation?* `https://quantiki.org/wiki/what-quantum-computation`.

Rieffel, E., & Polak, W. (2000). An Introduction to Quantum Computing for Non-Physicists. *CSUR ACM Comput. Surv. ACM Computing Surveys*, *32*(3), 300-335.

Salomaa, A. (1990). *Public-Key Cryptography*. Springer.

Slingerland, J. (2016). *Quantum vs Classical Computation.* `http://www.thphys.may.ie/staff/joost/TQM/QvC.html`.

Sofge, D. A. (2008). A Survey of Quantum Programming Languages: History, Methods, and Tools. In *Proceedings of the Second International Conference on Quantum, Nano, and Micro Technologies (ICQNM 2008)* (p. 66-71).

Steinruecken, C. (2004). *Quantum Computation on Non-Quantum Computers.* Kings College.

Strubell, E. (2011). *An Introduction to Quantum Algorithms.* University of Massachusetts Amherst.

Thompson, S. E., & Parthasarathy, S. (2006). Moore's Law: The Future of Si Microelectronics. *Materials Today*, *9*(6), 20-25.

Valiron, B., Ross, N. J., Selinger, P., Alexander, D. S., & Smith, J. M. (2015). Programming the Quantum Future. *Communications of the ACM*, *58*(8), 52-61.

Ventura, D., & Martinez, T. (1998). *Quantum Associative Memory.* `arXiv:quant-ph/9807053v1`.

Wecker, D. (2016). Language-Integrated Quantum Operations (LIQUi$|\rangle$) Simulator User's Manual [Computer software manual].

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

Wolf, R. D. (1999). *Quantum Computation and Shor's Factoring Algorithm.* CWI and University of Amsterdam.

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 – Quantum Fourier Transform

According to the definition of the quantum Fourier transform (QFT), it is the discrete Fourier transform (DFT) applied to the amplitudes of a quantum state that performs a linear transformation on qubits (Bacon, 2006). The DFT is a version of Fourier transform that performs a transformation on a discrete number of frequencies and discrete-time signal. The DFT of a vector can be defined as:

$$\tilde{f}(k) = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi i(\frac{jk}{N})} f(j)$$

The QFT is considered the same transformation as the DFT. However, since the transformation occurs on the qubit states, a unitary operator F on n qubits $(N = 2^n)$ must be defined as:

$$F\,|j\rangle = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{2\pi i(\frac{jk}{2^n})}\,|k\rangle$$

For example, the transformation coefficient of a two-qubit quantum register can be expressed as the following:

$$|j\rangle \to \frac{1}{2}(|00\rangle + e^{2\pi i\frac{j}{4}}\,|01\rangle + e^{2\pi i\frac{j2}{4}}\,|10\rangle + e^{2\pi i\frac{j3}{4}}\,|11\rangle)$$

Thus, the QFT of $|j\rangle$ can be represented as:

$$|00\rangle \to \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle$$

$$|01\rangle \to \frac{1}{2}(|00\rangle + \omega\,|01\rangle + \omega^2\,|10\rangle + \omega^3\,|11\rangle$$

$$|10\rangle \to \frac{1}{2}(|00\rangle + \omega^2\,|01\rangle + \omega^4\,|10\rangle + \omega^6\,|11\rangle$$

$$|11\rangle \to \frac{1}{2}(|00\rangle + \omega^3\,|01\rangle + \omega^6\,|10\rangle + \omega^9\,|11\rangle$$

Essentially, the QFT maps the standard basis state $(|j\rangle)$ to the Fourier basis stage. The transformation is also unitary, which allows high efficiency when performing on a quantum computer.

Appendix B – LIQ$Ui|\rangle$ Experiment

B.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]
```

B.2 LIQ$Ui|\rangle$ 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 ================
```

Appendix C – Built-in Samples

```
==================================================================================
=   The Language-Integrated Quantum Operations (LIQUi|>) Simulator              =
=   Copyright (c) 2015,2016 Microsoft Corporation                               =
=   If you use LIQUi|> in your research, please follow the guidelines at        =
=   https://github.com/msr-quarc/Liquid for citing LIQUi|> in your publications.  =
==================================================================================

TESTS (all start with two underscores):
__Big()             Try to run large entanglement tests (16 through 33 qubits)
__Chem(m)           Solve Ground State for molecule m (e.g., H2O)
__ChemFull(...)     See QChem docs for all the arguments
__Correct()         Use 15 qubits+random circs to test teleport
__Entangle1(cnt)    Run n qubit entanglement circuit (for timing purposes)
__Entangle2(cnt)    Entangle1 with compiled and optimized circuits
__Entangles()       Draw and run 100 instances of 16 qubit entanglement test
__EntEnt()          Entanglement entropy test
__EIGS()            Check eigevalues using ARPACK
__EPR()             Draw EPR circuit (.htm and .tex files)
__Ferro(false,true) Test ferro magnetic coupling with true=full, true=runonce
__JointCNOT()       Run CNOTs defined by Joint measurements
__Noise1(d,i,p)     d=# of idle gates, i=iters, p=probOfNoise
__NoiseAmp()        Amplitude damping (non-unitary) noise
__QECC()            Test teleport with errors and Steane7 code (gen drawing)
__QFTbench()        Benchmark QFT used in Shor (func,circ,optimized)
__QLSA()            Test of HHL linear equation solver
__QuAM()            Quantum Associative Memory
__QWalk(typ)        Walk tiny,tree,graph or RMat file with graph information
__Ramsey33()        Try to find a Ramsey(3,3) solution
__SG()              Test spin glass model
__Shor(N,true)      Factor N using Shor's algo false=direct true=optimized
__show("str")       Test routine to echo str and then exit
__Steane7()         Test basic error injection in Steane7 code
__Teleport()        Draw and run original, circuit and grown versions
__TSP(5)            Try to find a Traveling Salesman solution for 5 to 8 cities
```