Grover's Algorithm

Experiment Findings · July 2018

DOI: 10.13140/RG.2.2.30860.95366

2 authors:

Akanksha Singhal
Manipal University Jaipur
1 PUBLICATION 0 CITATIONS
SEE PROFILE

Arko Chatterjee
Shiv Nadar University
1 PUBLICATION 0 CITATIONS
SEE PROFILE

Grover's Algorithm

Akanksha Singhal, Arko Chatterjee [1]

July 27, 2018

[1]Internship Report on Grover's Algorithm

Contents

0.1 Abstract

Limited connectivity of qubits and hardware fidelity poses a challenge to implement large quantum algorithms on actual devices .This paper is targeted to run 3 qubit Grover's search on IBM QX4 platform and compare its results by implementing the same circuit sequence on the simulator offered on IBM Quantum experience i.e. IBM Q QASM Simulator. In this paper we have studied the IBM Q 5 Tenerife [ibmqx4] architecture i.e. arrangement and connectivity of qubits and modified the circuit for 3 Qubit Grover's Search so as to successfully run it on the hardware. In this paper we have first discussed the naïve approach to design a circuit similar to the theoretical one to implement the algorithm by employing few swap gates. Later we try to design modified circuits implementing the same logic with minimum number of swap gates by reducing the number of gate stages as much as possible to reduce the errors due to hardware and entanglement of qubits. We have also implemented 16 qubit Grover's Search on well known simulators such as Microsoft's LIQUID, Microsoft's Quantum Development Kit and Quirk: Quantum Circuit Simulator presented the approach to implement the search and shared the experience along with the comparision of the final results on these simulators in terms of accuracy and time required to execute them.

0.2 Quantum Computing

Quantum computing is computing using quantum-mechanical phenomena, such as superposition and entanglement.

Superposition : Superposition is essentially the ability of a quantum system to be in multiple states at the same time

Entanglement : Entanglement is an extremely strong correlation that exists between quantum particles, such that observing one of two entangled qubits causes it to behave randomly, but tells the observer exactly how the other qubit would act if observed in a similar manner, even if separated by great distances.

0.2.1 Quantum Gates[7]

0.2.1.1 Hadamard gate

Figure 1: Hadamard gate

The hadamard gate is used to create an uniform superposition of the input states. The Hadamard gate acts on a single qubit. It maps the basis state $|0\rangle$ to $\dfrac{|0\rangle + |1\rangle}{\sqrt{2}}$ and $|1\rangle$ to $\dfrac{|0\rangle - |1\rangle}{\sqrt{2}}$, which means that a measurement will have equal probabilities to become 1 or 0. It is represented by the matrix:

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

0.2.1.2 Pauli-X gate

Figure 2: X gate

X gate is known as a "bit-flip", since it flips the 0 to 1 and vice versa. This is similar to a classical NOT gate. It is also known as an Xrotation, since it rotates the state by π radians around the Xaxis. It is represented by the Pauli matrix:

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

0.2.1.3 Pauli-Z gate

Figure 3: Z gate

The Pauli-Z gate acts on a single qubit. It equates to a rotation around the Z-axis of the Bloch sphere π radians. It leaves the basis state $|0\rangle$ unchanged and maps $|1\rangle$ to $-|1\rangle$. Due to this nature, it is sometimes called phase-flip. It is represented by the Pauli Z matrix:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

0.2.1.4 S gate

Figure 4: S gate

S gate is a rotation of $\pi/2$ around Z. It is represented by the matrix:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$$

0.2.1.5 T gate

Figure 5: T gate

S gate is a rotation of $\pi/4$ around Z. It is represented by the matrix:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

0.2.1.6 $S^\dagger$ gate

Figure 6: $S^\dagger$ gate

$S^\dagger$ gate is the inverse of S and is a rotation of $-\pi/2$ around Z.

0.2.1.7 $T^\dagger$ gate

Figure 7: $T^\dagger$ gate

$T^\dagger$ is the inverse of T and is a rotation of $-\pi/4$ around Z.

0.2.1.8 Contolled Not gate

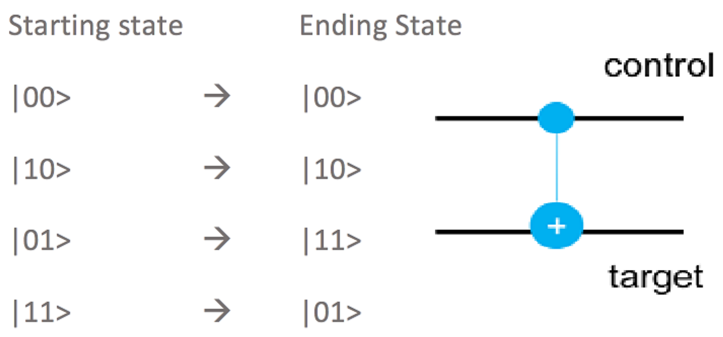


Figure 8: CNOT gate

[24]

The CNOT gate's action on classical basis states is to flip (apply a NOT or X gate to) the target qubit only if the control qubit is $|1\rangle$; otherwise it does nothing.

0.2.2 Implementaion of classical gates in quantum

1. **Classical AND Gate**
   In digital electronics, the AND gate outputs true only when both the inputs are true. In quantum computing, AND gate can be implemented using toffoli gate.

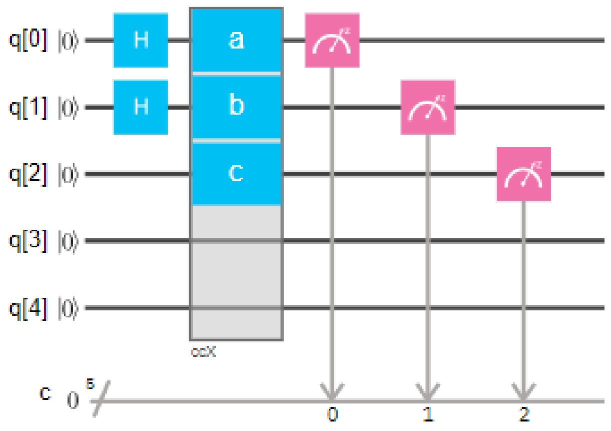


Figure 9: Quantum implementation of classical AND

An uniform superposition of all the four states { 00, 01, 10, 11} is prepared. The result shows that the output (most significant bit) is true only if both the inputs are true.

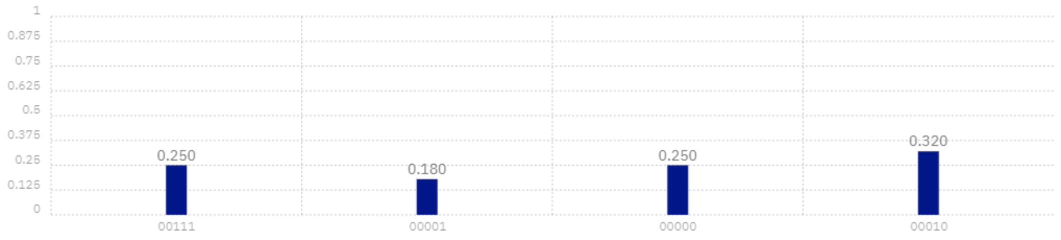


Figure 10: Output

2. **Classical OR Gate**
   The OR gate outputs true if at least one of the inputs is true.

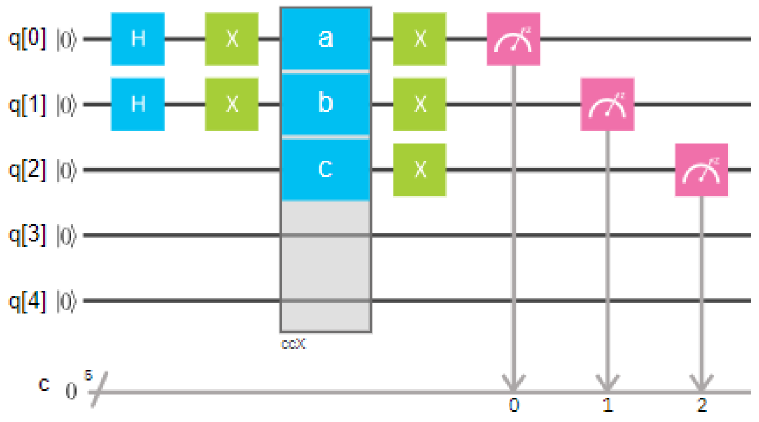


Figure 11: Quantum implementation of classical OR

It is implemented using toffoli gate based on De Morgan's law ($\overline{\bar{x}.\bar{y}}$ = x + y). The result shows that the output bit (most significant bit) is true if at least one of the inputs is true.

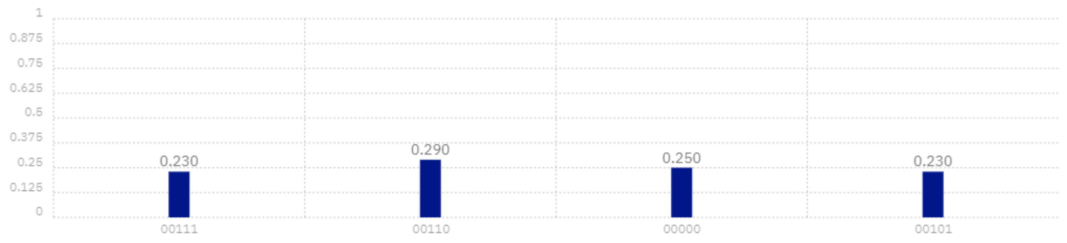


Figure 12: Output

3. **Classical XOR Gate**
   The XOR gate (Exclusive-OR gate) is a digital logic gate that gives a true output when the number of true inputs is odd.

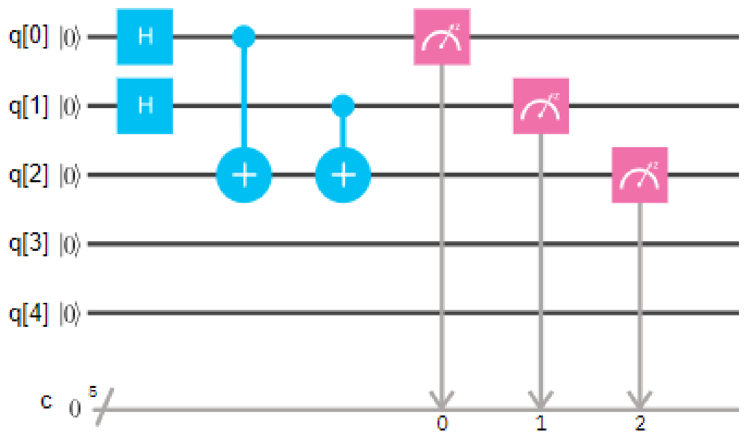


Figure 13: Quantum implementation of classical XOR

The result shows that, the output bit (most significant bit) is true only for the inputs { 01, 10 }.

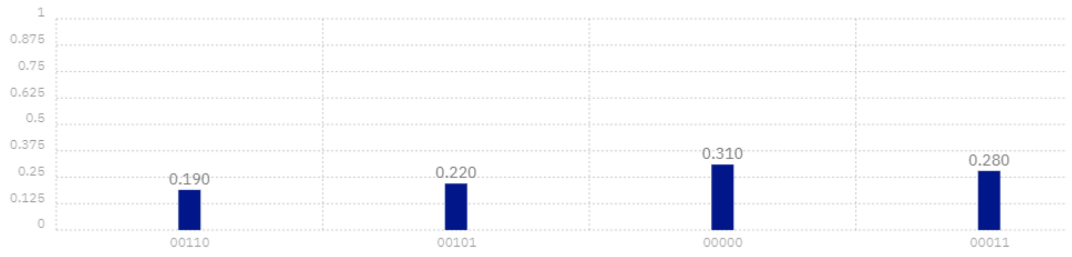


Figure 14: Output

0.3 Introduction to Grover's Algorithm

- Grover's algorithm is a quantum algorithm that solves the problem of unstructured search.
- It finds with high probability the unique input to a black box function that produces a particular output value, using just
- O $(\sqrt{N})$ evaluations of the function, where N is the size of the function' domain.
- By comparison, a classical search algorithm will get the correct answer after an average of N/2 queries of the oracle.
- It was devised by Lov Grover in 1996.

0.3.1 Stages of Grover's Algorithm

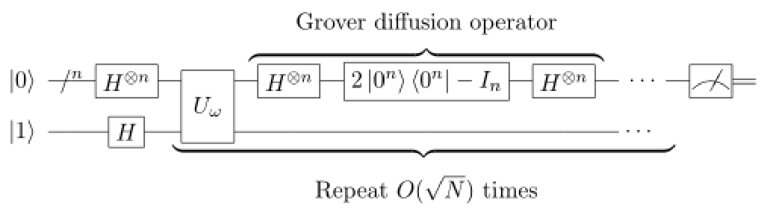

Grover diffusion operator
$|0\rangle$
n
$H^{\otimes n}$
$|1\rangle$
H
U_ω
$H^{\otimes n}$
$2|0^n\rangle\langle 0^n| - I_n$
$H^{\otimes n}$
Repeat $O(\sqrt{N})$ times

Figure 15: Initialization, oracle, amplification, and measurement

Step I: Initialization

Uniform Superposition $|s\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle$

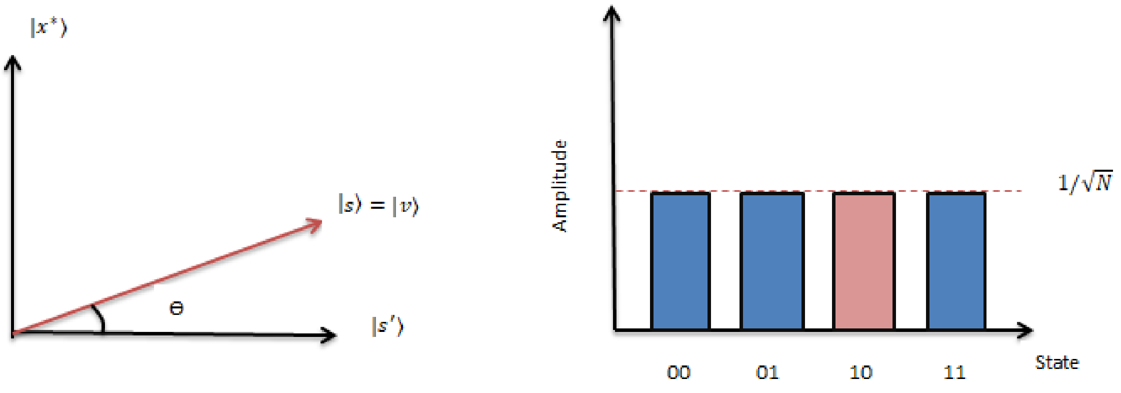


Figure 16: Initialization

Step II: Oracle

The oracle stage marks the solution(s) by flipping the sign of that state's amplitude.

Black Box Function

$$f(x^*) = 1$$

$$f(x) = 0$$

Oracle Function

$$U_f|x\rangle = (-1)^{f(x)}|x\rangle$$

$$(-1)^{f(x)}\,|x\rangle = \begin{cases} |x\rangle & \text{if } f(x)=0 \\ -\,|x\rangle & \text{if } f(x)=1. \end{cases}$$

The diagram of the gate is given below.

$$|x\rangle \;—\boxed{O_f^{\pm}}—\; (-1)^{f(x)}\,|x\rangle$$

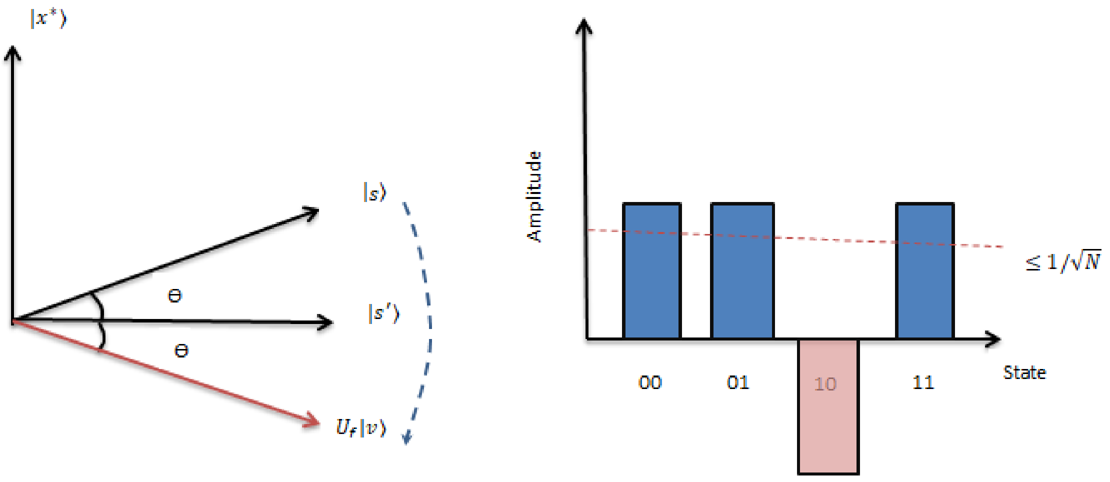


Figure 17: Oracle

Step III: Amplification

The amplification stage performs a reflection about the mean, thus increasing the amplitude of the marked state.

Diffusion Operation $\widehat{D} = 2|s\rangle\langle s| - \widehat{I}$

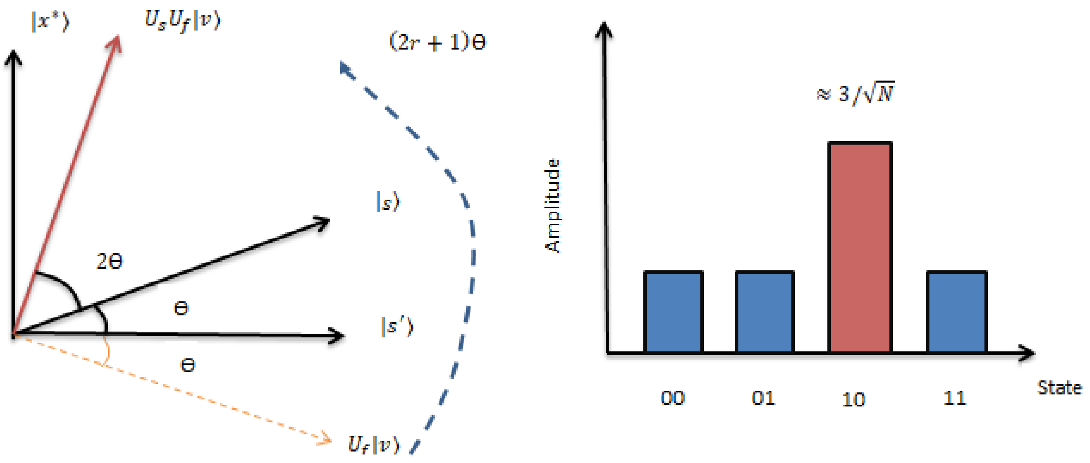


Figure 18: Diffusion Operation

Step IV: Measurement

Finally, the algorithm output is measured.

0.3.2 Potential of Grover's Algorithm

0.3.2.1 Threat to AES symmetric block cipher

Grover's algorithm could brute-force a 128-bit symmetric cryptographic key in roughly 264 iterations, or a 256-bit key in roughly 2128 iterations. As a result, it is sometimes suggested that symmetric key lengths be doubled to protect against future quantum attacks. Hence Grover's Search can be used to accelerate brute-force attacks on symmetric block ciphers like AES. However, it does not lead to a complete break but roughly halves the bit-security level of symmetric algorithms. Hence, the key length of algorithms with a 128-bit key and thus 128-bit security has to be doubled (moving to AES-256).
Post-quantum cryptography (PQC) refers to new cryptographic algorithms executed on classical computers that are expected to be efficiently secured against attackers using quantum computers. In contrast, quantum cryptography (QC) or more specifically quantum key distribution (QKD) uses properties of quantum mechanics to establish a secret key between two parties. This secret key can then be used to encrypt bulk data using classic symmetric ciphers like AES.
Grover's algorithm can also be used for estimating the mean and median of a set of numbers, and for solving the collision problem.

0.3.3 Optimal No. of Iterations

Optimal No. of Iterations = $\frac{\pi}{4}\sqrt{N}$
{ for proof refer to Appendix .1.1. } Classically one might think that applying more Grover Iterations will better the chance of success. This is not the case.
"the probability of success does not increase monotonically with the number of iterations." Performing half that number of iterations,$\frac{\pi}{8}\sqrt{N}$, will give about a 50/50 chance of success. However, doing twice that number of iterations, $\frac{\pi}{2}\sqrt{N}$, decreases the probability to measure the correct solution to a near negligible amount, close to 1

0.3.4 Gate Complexity

The number of gates applied will be in the order of $O(\sqrt{N}logN)$ for large values of N.
For proof refer to Appendix .1.2.

0.3.5 Applicability and Limitations

When applications of Grover's algorithm are considered, it should be emphasized that the database is not represented explicitly. Instead, an oracle is invoked to evaluate an item by its index. Reading a full database item by item and converting it into such a representation may take a lot longer than Grover's search. To account for such effects, Grover's algorithm can be viewed as solving an equation or satisfying a constraint. In such applications, the oracle is a way to check the constraint and is not related to the search algorithm. This separation usually prevents algorithmic optimizations, whereas conventional search algorithms often rely on such optimizations and avoid exhaustive search

0.4 Literature Survey

(1996) A fast quantum mechanical algorithm for database search

Lov K. Grover, 3C-404A, Bell Labs, 600 Mountain Avenue, Murray Hill NJ 07974, lkgrover@bell-labs.com

This is the main paper by Lov K. Grover, which described the Quantum search algorithm.In the paper,it is stated that, in an unsorted database of phone numbers, In order to find someone's phone number with a probability of 1/2 , any classical algorithm (whether deterministic or probabilistic) will need to look at a minimum of N/2 names. whereas, using Quantum search, the desired phone number can be obtained in only $\sqrt{N}$ steps.

(1996) Strengths and Weaknesses of Quantum Computing [1]

Charles H. Bennett ,Ethan Bernstein, Gilles Brassard

This paper addresses the question of whether all of NP can be efficiently solved in quantum polynomial time by a quantum computer, proving that relative to an oracle chosen uniformly at random, with probability 1, the class NP cannot be solved on a quantum Turing machine in time $o(2^{n/2})$. We also show that relative to a permutation oracle chosen uniformly at random, with probability 1, the class NP $\cap$ co–NP cannot be solved on a quantum Turing machine in time $o(2^{n/3})$.In the end,this paper states that the former bound is tight since recent work of Lov K. Grover shows how to accept the class NP relative to any oracle on a quantum computer in time $O(2^{n/2})$.

(1996) Tight bounds on quantum searching [2]

Boyer, Michel and Brassard, Gilles and Hoyer, Peter and Tapp, Alain

This paper provides a tight analysis of Grover's recent algorithm for quantum database searching. It gives a simple closed-form formula for the probability of success after any given number of iterations of the algorithm. Furthermore, the paper analyses the behaviour of the algorithm when the element to be found appears more than once in the table and provides a new algorithm to find such an element even when the number of solutions is not known ahead of time. Using techniques from Shor's quantum factoring algorithm in addition to Grover's approach, the paper introduces a new technique for approximate quantum counting, which allows to estimate the number of solutions. Finally a lower bound on the effciency of any possible quantum database searching algorithm is provided and the paper shows that Grover's algorithm nearly comes within a factor 2 of being optimal in terms of the number of probes required in the table.

(1998) Experimental Implementation of Fast Quantum Searching [3]

Chuang, Isaac L and Gershenfeld, Neil and Kubinec, Mark

This paper first discusses the problems with Quantum computing such as coherence and provides the solutions: 1. Quantum error correction can be used with imperfect computers. 2. Computing with mixed state ensembles rather than isolated systems in the pure state. These ideas have been applied in the first experimental realization of a significant quantum search algorithm, using nuclear magnetic resonance (NMR) techniques to perform Grover's search algorithm. The coherence times were measured to be T1 = 20 and T2 = 0.4 sec for the proton, and T1 = 21 sec and T2 = 0.3 sec for the carbon (the large ratio is due to C-Cl relaxation), and the coupling was measured to be J = 215 Hz. Grover search was perform to get the state $x_0 = 3$ (out of the states 0,1,2 for N = 4) The resulting density matrix of the experiment clearly revealed the $|1\rangle$ state corresponding to $x_0 = 3$.

(2000) Implementation of a three-quantum-bit search algorithm [5]

Vandersypen, Lieven MK and Steffen, Matthias and Sherwood, Mark H and Yannoni, Costantino S and Breyta, Gregory and Chuang, Isaac L

This paper reports the experimental implementation of Grover's quantum search algorithm on a quantum computer with three quantum bits. The computer consists of molecules of Clabeled CHFBr2, in which the three weakly coupled spin-1/2 nuclei behave as the bits and are initialized, manipulated, and read out using magnetic resonance techniques. This quantum computation is made possible by the introduction of two techniques which significantly reduce the complexity of the experiment and by the surprising degree of cancellation of systematic errors which have previously limited the total possible number of quantum gates. The first method to reduce the number of one- and two spin gates used to realize any given n-qubit unitary operation starts with a library of efficient implementations for often-used building blocks. The second method to reduce the complexity concerns the initialization of the qubits to the ground state, generally the first step in quantum algorithms.

(2002) Implementation of quantum search algorithm using classical Fourier optics [6]

Bhattacharya, N and van den Heuvell, HB van Linden and Spreeuw, RJC

This paper reports on an experiment on Grover's quantum search algorithm, showing that classical waves can search an N-item database as efficiently as quantum mechanics can. The transverse beam profile of a short laser pulse is processed iteratively as the pulse bounces back and forth between two mirrors. We directly observe the sought item being found in $\sim \sqrt{N}$ iterations, in the form of a growing intensity peak on this profile. The paper also described the problem with using classical waves: lack of quantum entanglement. Although the lack of quantum entanglement limits the size of the database, the results of this paper, show that entanglement is neither necessary for the algorithm itself, nor for its efficiency.

(2004) Quantum query complexity of some graph problems [10]

Dürr, Christoph and Heiligman, Mark and HOyer, Peter and Mhalla, Mehdi This paper shows the potential of Quantum algorithm to speed up some of the classical graph problems such as:

1. Minima Finding :
   Suppose we are given a function f defined on a domain of size n, and we want to find an index i so that f(i) is a minimum in the image of f

   (a) Initially let j ∈ [N] be an index chosen uniformly at random.

   (b) Repeat forever

      i. Find an index i ∈ [N] such that f(i) ¡ f(j).

      ii. Set j := i.

2. Minimum Spanning Tree:
   In this section undirected graphs with weighted edges are considered. In Minimum Spanning Tree we wish to compute a cycle-free edge set of maximal cardinality that has minimum total weight

   - Let T1,T2,...,Tk be a spanning forest. Initially, k = n and each tree Tj contains a single vertex.
   - Set l = 0.
   - Repeat until there is only a single spanning tree (i.e., k = 1).
     - Increment l.
     - Find edges e1,e2,...,ek satisfying that ej is a minimum weight edge leaving Tj. Interrupt when the total number of queries is (l + 2)c$\sqrt{(km)}$ for some appropriate constant c.
     - Add the edges e' j to the trees, merging them into larger trees.
   - Return the spanning tree T1.

3. Connectivity:
   A special case of Minimum Spanning Tree when all edge weights are equal, is Graph Connectivity. The input is an undirected graph and the output is a spanning tree, provided the graph is connected

(2005) Experimental One-Way Quantum Computing [9]

P.Walther, K.J.Resch, T.Rudolph, E.Schenck,*, H.Weinfurter, V.Vedral, M.Aspelmeyer and A.Zeilinger

This paper introduces a new model which requires qubits to be initialized in a highly-entangled cluster state. From this point, the quantum computation proceeds by a sequence of single-qubit measurements with classical feedforward of their outcomes A cluster state can be thought of as emerging from an array of equally prepared independent qubits, which then interact via controlled-phase (CPhase) gates with their nearest (connected) neighbours. Specifically, a cluster state can be built up by, first preparing an uniform superposition of a large number of Qubits ($|+\rangle = (|0\rangle + |1\rangle)/2$, where 0 and 1 are the computational basis of the physical qubits). Then, a CPhase operation $|j\rangle|k\rangle \rightarrow (-1)^{jk}|j\rangle|k\rangle$, with (j , k ∈ 0,1) is applied between pairs of neighbouring, connected qubits and effectively generates entanglement between them. According to the paper, the probability of the quantum computer determining the correct outcome was about 90%.

(2005) Implementation of Grover's Quantum Search Algorithm in a Scalable System [8]

Brickman, K-A and Haljan, PC and Lee, PJ and Acton, M and Deslauriers, L and Monroe, C

This paper reports the implementation of Grover's quantum search algorithm in the scalable system of trapped atomic ion quantum bits for two qubits. Any one of four possible states of a two-qubit memory is marked, and following a single query of the search space, the marked element is successfully recovered with an average probability of 60(2)%. This exceeds the performance of any possible classical search algorithm, which can only succeed with a maximum average probability of 50%.

(2008) On the Research of BDD Based Simulation of Grover's Algorithm [11]

Xue, Xiling and Chen, Hanwu and Chen, Kaizhong and Li, Zhiqiang

In this paper, they adopt an efficient data structure called the Binary Decision Diagram (BDD) that exploits the structure displayed in quantum computing to simulate Grover's quantum search algorithm. First they adapt the original BDD and implement a series of algorithms to represent and manipulate matrices and vectors. Then instances of Grover's algorithm are simulated using our programme written in C++. A schematic diagram plotting the probability of successfully finding the item against the number of Grover iterations. It can be seen that the probabilities of successful search as a function of the number of iterations follow a sinusoidal curve, after a certain point the probability of successful measurement starts fading.

(2009) Demonstration of Two-Qubit Algorithms with a Superconducting Quantum Processor [12]

DiCarlo, L and Chow, JM and Gambetta, JM and Bishop, Lev S and Johnson, BR and Schuster, DI and Majer, J and Blais, A and Frunzio, L and Girvin, SM and others

Simultaneously meeting the conflicting requirements of long coherence, state preparation, universal gate operations, and qubit readout makes building quantum processors challenging. Few-qubit processors have already been shown in nuclear magnetic resonance, cold ion trap and optical systems, but a solid-state realization has remained an outstanding challenge. This research paper demonstrates a two-qubit superconducting processor and the implementation of the Grover search and Deutsch–Jozsa quantum algorithms. It employs a novel two-qubit interaction, tunable in strength by two orders of magnitude on nanosecond time scales, which is mediated by a cavity bus in a circuit quantum electrodynamics (cQED) architecture. This interaction allows generation of highly-entangled states with concurrence up to 94Although this processor constitutes an important step in quantum computing with integrated circuits, continuing efforts to increase qubit coherence times, gate performance and register size will be required to fulfill the promise of a scalable technology.

(2011) Efficient Grover search with Rydberg blockade [13]

Mølmer, Klaus and Isenhower, Larry and Saffman, Mark

This paper presents effcient methods to implement the quantum computing Grover search algorithm using the Rydberg blockade interaction. It shows that simple π -pulse excitation sequences between ground and Rydberg excited states readily produce the key conditional phase shift and inversion-about-the mean unitary operations for the Grover search. Multi-qubit implementation schemes suitable for different properties of the atomic interactions were identifed and the error scaling of the protocols with system size was found to be promising for experimental investigation.

(2016) On the advantages of using relative phase Toffolis with an application to multiple control Toffoli optimization

Dmitri Maslov, * 1National Science Foundation, Arlington, Virginia, USA In this paper an approach for systematic optimization of quantum circuits via replacing suitable pairs of the multiple control Toffoli gates with their relative phase implementations was reported.This operation preserves the functional correctness. The advantage can be witnessed through the optimized resource counts. Our demonstrated optimizations include a simultaneous optimization of the T count by a factor of 4/3 in the leading constant, the CNOT count by a factor 2 in the leading constant, and the number of ancillary qubits by a factor of 2 in the leading constant.

(2016) Demonstration of a small programmable quantum computer with atomic qubits [16]

Debnath, Shantanu and Linke, Norbert M and Figgatt, Caroline and Landsman, Kevin A and Wright, Kevin and Monroe, Christopher

It demonstrates a five-qubit trappedion quantum computer that can be programmed in software to implement arbitrary quantum algorithms by executing any sequence of universal quantum logic gates. It compiles algorithms into a fully-connected set of gate operations that are native to the hardware and have a mean fidelity of 98 %. Unlike solid-state implementations , the quantum circuitry is determined by external fields, and hence can be programmed and reconfigured without altering the structure of the qubits themselves. As examples, it implements the Deutsch-Jozsa (DJ)h and Bernstein-Vazirani (BV) algorithms with average success rates of 95 % and 90 %, respectively. We also perform a coherent quantum Fourier transform (QFT) on five trappedion qubits for phase estimation and period finding with average fidelities of 62 % and 84 %, respectively. This small quantum computer can be scaled to larger numbers of qubits within a single register, and can be further expanded by connecting several such modules through ion shuttling or photonic quantum channels .

(2017)Complete 3-Qubit Grover search on a programmable quantum computer

C. Figgatt 1, D. Maslov1,2, K.A. Landsman1, N.M.Linke1, S.Debnath1 and C. Monroe1,3

It reports results for a complete three-qubit Grover search algorithm using the scalable quantum computing technology of trapped atomic ions, with better-than-classical performance. Two methods of state marking are used for the oracles: a phase-flip method employed by other experimental demonstrations, and a Boolean method requiring an ancilla qubit that is directly equivalent to the state marking scheme required to perform a classical search. It also reports the deterministic implementation of a Toffoli-4 gate, which is used along with Toffoli-3 gates to construct the algorithms; these gates have process fidelities of 70.5% and 89.6%, respectively.

(2017) Experimental comparison of two quantum computing architectures

Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe

This paper focuses on runnnig a selection of algorithms on two state-of-the-art 5-qubit quantum computers that are based on different technology platforms. One is a publicly accessible superconducting transmon device www.research.ibm.com/ibm-q with limited connectivity, and the other is a fully connected trapped-ion system. Even though the two systems have different native quantum interactions, both can be programed in a way that is blind to the underlying hardware, thus allowing a comparison of identical quantum algorithms between different physical systems. The research paper shows that quantum algorithms and circuits that use more connectivity clearly benefit from a better-connected system of qubits. Although the quantum systems here are not yet large enough to eclipse classical computers, this experiment exposes critical factors of scaling quantum computers, such as qubit connectivity and gate expressivity. In addition, the results suggest that codesigning particular quantum applications with the hardware itself will be paramount in successfully using quantum computers in the future.

0.5 *Our Contribution*

- Modified the circuit for 3 Qubit Grover's Search according to IBM Q 5 Tenerife's [ibmqx4] architecture i.e. arrangement and connectivity of qubits.
- We finally executed 3 qubit Grover's search on IBM QX4 platform IBM Q QASM Simulator.
- Multiple approaches for circuit designs are discussed with optimization by reducing the number of gate stages and the number of swap gates (basically the cnots) as much as possible to reduce the errors.
- We have also done an error analysis of the back of the envelope propagation of the number of involved gates and used the posted errors on the ibmqx4 device to estimate what final state fidelity we should have expected for our circuit.
- We have also implemented 16 qubit Grover's Search on well known simulators such as Microsoft's LIQUID, Microsoft's Quantum Development Kit and Quirk: Quantum Circuit Simulator presented the approach to implement the search and shared the experience along with the comparision of the final results on these simulators in terms of accuracy and time required to execute them.

0.6 IBM Platforms and Simulators

The Quantum Information Science Kit (Qiskit for short) is a software development kit (SDK) for working with OpenQASM and the IBM Q Experience (QX). Qiskit can be used to create quantum computing programs, compile them, and execute them on one of several backends (online Real quantum processors, online simulators, and local simulators). For the online backends, Qiskit uses our python API client to connect to the IBM Q Experience.
Quantum Information Science Kit (QISKit) is available to anyone interested in learning how to encode and simulate algorithms designed for a quantum computer.

QISKit is the software that sits between quantum algorithms from one side, and the physical quantum device from the other. It translates common programming languages like Python into quantum machine language. This means anyone outside of the IBM Q lab can program a quantum computer.

IBM Q QASM Simulator [ibmq_qasm_simulator] we can use max 32 qubits and and the device is available on QISKit.
Limitations : QASM Simulator allows the if conditions for less than 20 qubits. Fixed stages in composer makes it difficult to implement large algorithms. It is not possible to do a stage by stage analysis of the output. It lacks features to view the bloch sphere changes.

IBM Q 5 Yorktown [ibmqx2] and **IBM Q 5 Tenerife [ibmqx4]** are the 5 qubit devices available to users on QISKit
The connectivity is provided by two coplanar waveguide (CPW) resonators with resonances around 6.6 GHz (coupling Q2, Q3 and Q4) and 7.0 GHz (coupling Q0, Q1 and Q2). Each qubit has a dedicated CPW for control and readout. Currently ibmqx2 is under maintenance.

IBM Q 16 Rueschlikon [ibmqx5] is the 16 qubit device and is available on QiSKit
The connectivity on the device is provided by total 22 coplanar waveguide (CPW) ”bus” resonators, each of which connects two qubits. Three different resonant frequencies are used for the bus resonators. The resonant frequencies are 6.25 GHz, 6.45 GHz, 6.65 GHz.
IBM Q 20 Austin [QS1_1] and **IBM Q 20 Tokyo [ibmq_20_tokyo]** are 20 qubit devices which are available to hubs, partners, and members of the ibm network.

Limitations for Real Devices : Fixed stages in composer makes it difficult to implement large algorithms. Limited connectivity among the gates increases the need to rely on swap gates. Large errors are incorporated in the final answer due to cnots. Difference in frequency of qubits limits the direction of cnots. It is difficult to maintain coherence and prevent loss of state. Its difficult to avoid errors due to changes in device environment.

0.7 Other Simulators and their Pros and Cons

- **Quantum Kit (Q-KIT)**[19]
  Pros: It is a graphical quantum circuit simulator and works efficiently for less no.of qubits. Easy to add qubits while working and select gates. Allows stage by stage analysis of state of qubit. Allows adding custom gates. Shows the probability distributions and amplitude of the possible quantum states. Allows the bloch sphere analysis.
  Cons: Although it states to undertake operations for algorithms involving 20 qubits, it often hangs up and slows down around 16 qubits and the simulate feature does not work efficiently on laptop devices. The graphs generated for a large number of qubits are fuzzy and unreadable.

- **Quirk: Quantum Circuit Simulator**[23]
  Pros: Quirk offers a user – friendly environment to study quantum algorithms.It allows upto 16 qubits with an efficient toolbox to manage qubits efficiently with rotation gates and see their block sphere representation , the chance and amplitude of a particular quantum state, make custom gates and has an easy implementation of tofollis and swap gates. Results were obtained almost instantly and with a very high accuracy . Implementing 16 qubit grover’s search was easy and comfortable experience .
  Cons: Has a maximum qubit limitation of 16 qubits.

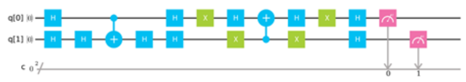
- **Microsoft’s Liqui$|\rangle$**[14]
  Pros: LIQUi$|\rangle$’s simulators are highly optimized, taking advantage of many available techniques including custom memory management, cache coherence analysis, parallelization, “gate growing”, and virtualization (running in the cloud). LIQUi$|\rangle$’s highly optimized simulation environment allows thorough investigation of quantum algorithms under noise, physical device constraints, and simulation. LIQUi$|\rangle$ is also a full optimizing compiler. A user’s input circuit definition may be massively rewritten (under user control) to generate compact, highly-optimized versions for simulation. We can compile any given unitary circuit with varying levels of optimization and can mathematically prove that the pre- and post-optimized unitary are identical even though the resulting circuits may appear very different. [18]
  Cons: Long simulation time.

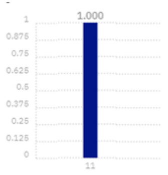
- **Microsoft’s Quantum Development Kit**[21]
  Pros: Microsoft’s Quantum Development Kit (QDK) uses Q#, which is a brand-new quantum-focused programming language with native type, operators, and other abstraction. Q# features rich integration with Visual Studio and VS Code and interoperability with the Python programming language. QDK allows us to simulate quantum solutions requiring up to 30 qubits with a local

simulator, or use the Azure simulator for large-scale quantum solutions requiring more than 40 qubits.
Cons: QDK does not have functions to visualize the circuits. It also does not have functions to optimize the circuit.

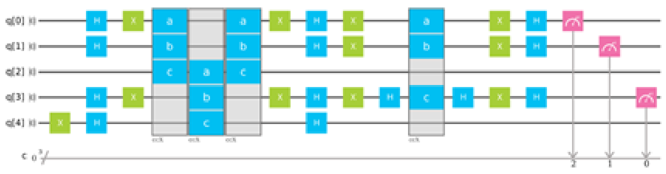
0.8 Implementation of Grover's Algorithm on IBM Simulator

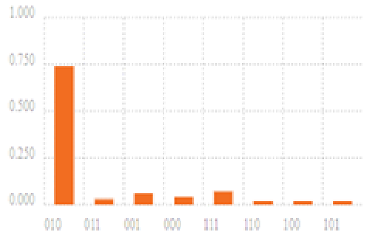
0.8.1 2 Qubit Implementation (IBM QX Simulator)



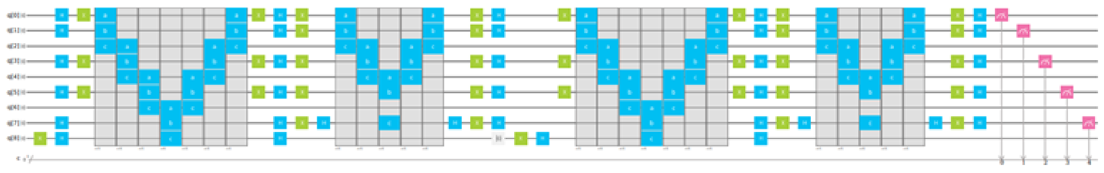


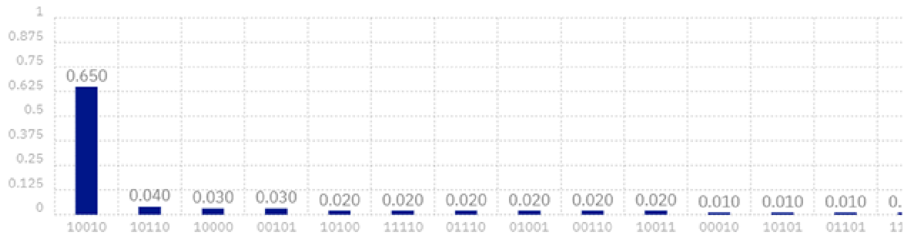
0.8.2 3 Qubit Implementation (IBM QX Simulator)





0.8.3 5 Qubit Implementation (IBM QX Simulator)





0.8.4 Basic Information about the 3 Qubit Grover's Search

[1]A 3 qubit grover search with n=3 bits can have $N = 2^n = 2^3 = 8$ possible correct solutions or $^8_2\text{C} = 28$ possible two result solutions The probability of measuring the correct solution after 1 interation of algorithm is $= t \cdot \left[\left[\frac{N-2t}{N} + \frac{2(N-t)}{N}\right] \frac{1}{\sqrt{N}}\right]^2 = 78.125\%$ (for N=8 and t=1)

The oracle marks the state using the Boolean method requiring one ancilla bit initialized as $|1\rangle$. The ancilla bit flips only if the input to oracle is one of the marked state.[17]

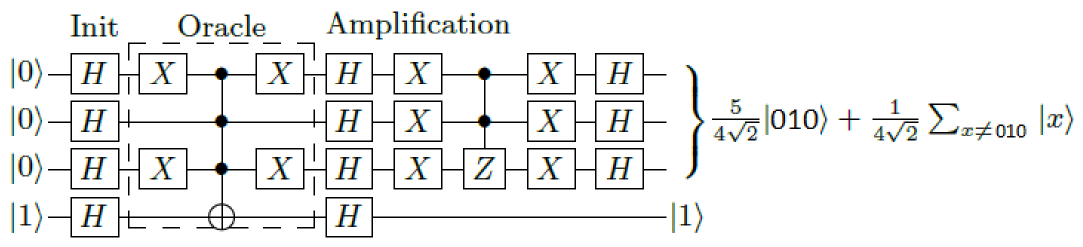

Init
Oracle
Amplification
$|0\rangle$
$|0\rangle$
$|0\rangle$
$|1\rangle$
$\frac{5}{4\sqrt{2}}|010\rangle + \frac{1}{4\sqrt{2}}\sum_{x\neq 010}|x\rangle$
$|1\rangle$

Figure 19: Circuit for Single Solution Boolean Oracle with Marked State $|010\rangle$ [17]

0.8.4.1 Device: Simulator

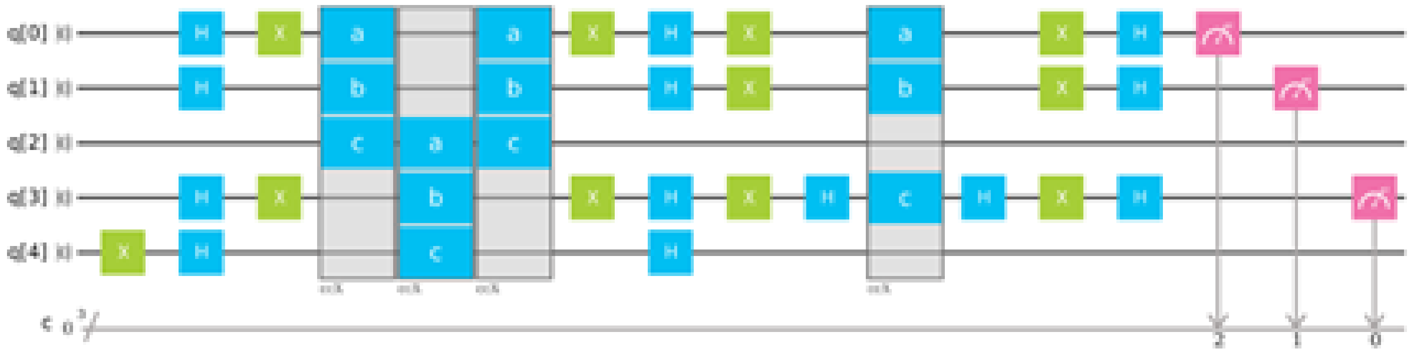


Figure 20: Implementation on IBM Q QASM Simulator

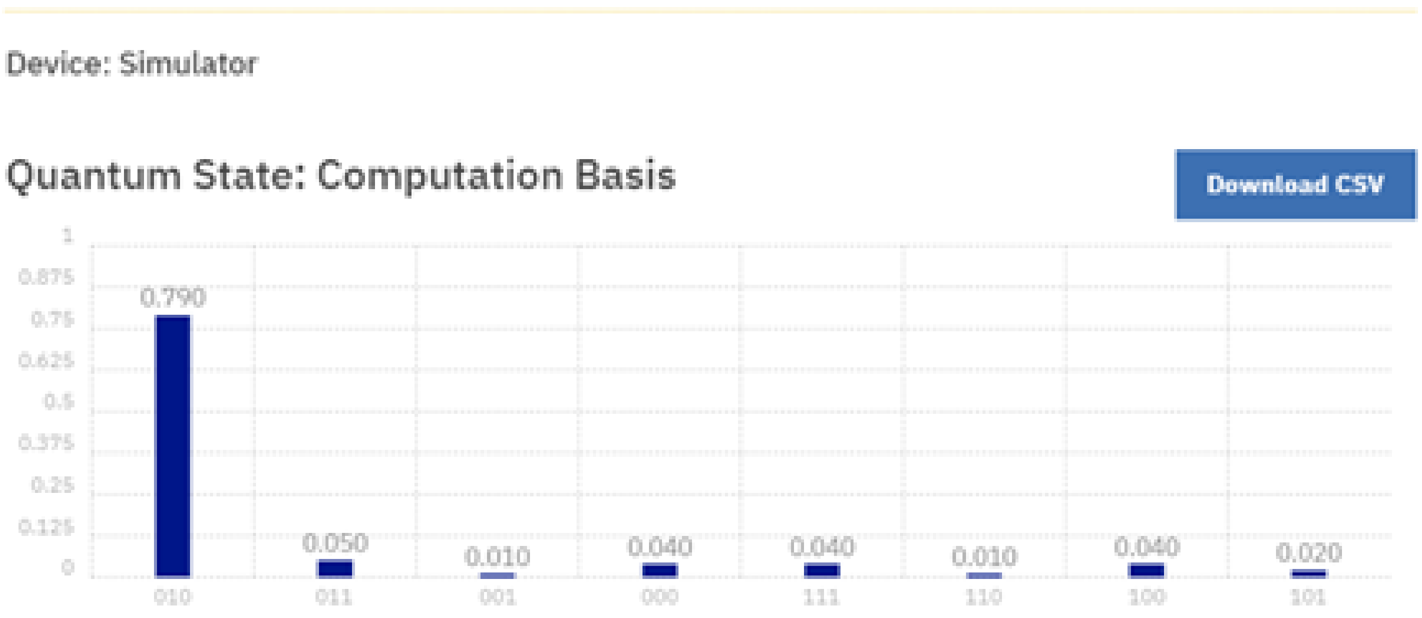
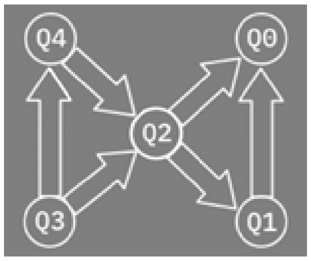


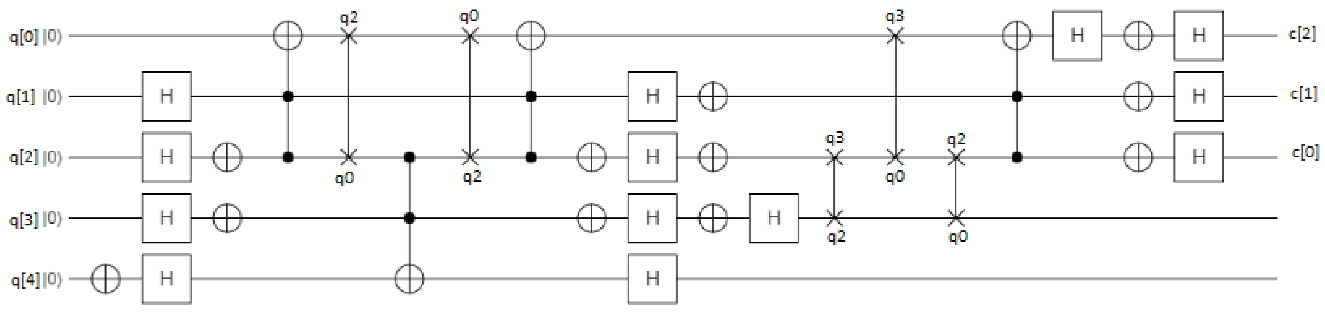
Figure 21: Results with Shots=8192

0.9 3 Qubit Grover's Search on IBM QX4

0.9.1 3 Qubit Grover's Search Implemented For IBM QX4 architecture



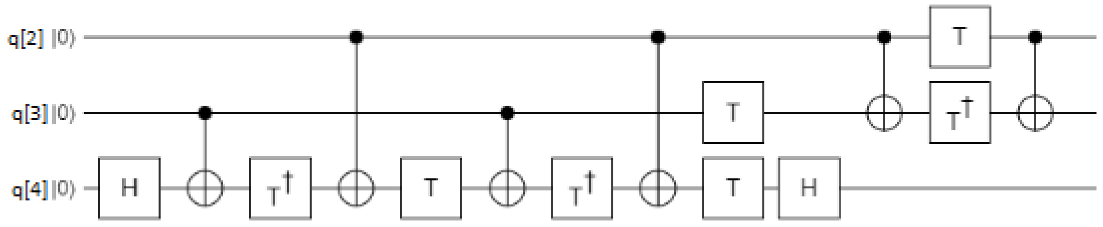
(a) IBM QX4



(b) Circuit for implemention of 3 Qubit Grover's Search Algorithm on IBM QX4

Figure 22: Modified circuit according to the direction of the C-nots using Swap gates

0.9.1.1 Normal Implementation of the 2^{nd} Tofolli Gate



0.9.1.2 Implementation of 2nd Tofolli gate for IBM QX4

Since CX q[2],q[4] and CX q[2],q[3] is not allowed in IBM QX4 we use swap gates to implement them.

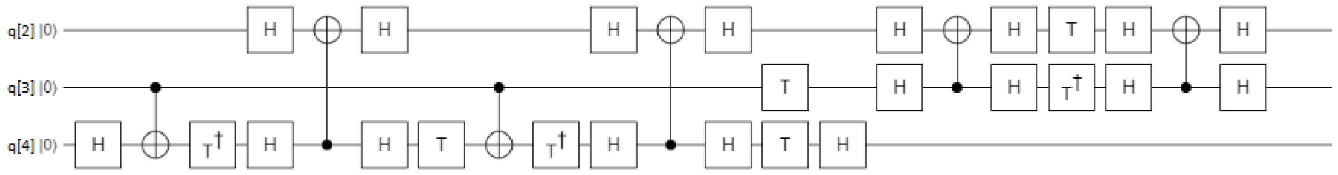


Figure 23: IBM QX4 Implementation of 2nd tofolli gate in Circuit given in Fig.14

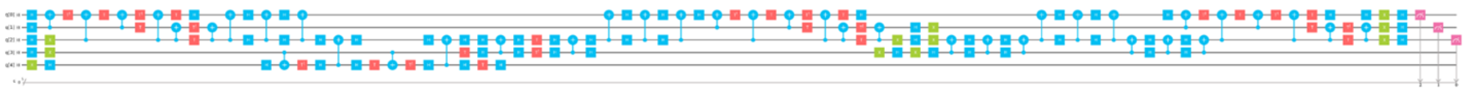


Figure 24: Circuit for implemention of 3 Qubit Grover's Search Algorithm on IBM QX4 on IBM Quantum Experience

0.9.1.3 Device: Simulator

Shots=8192

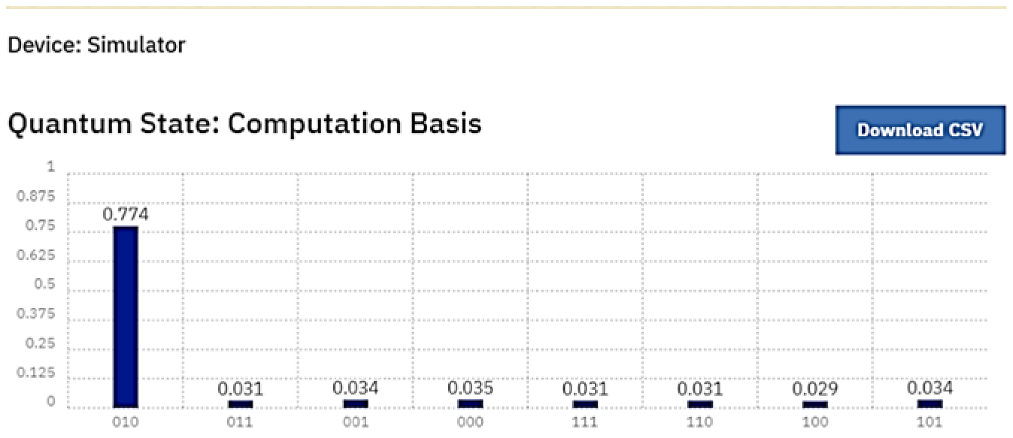
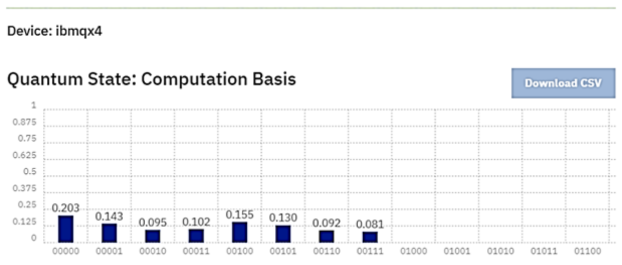
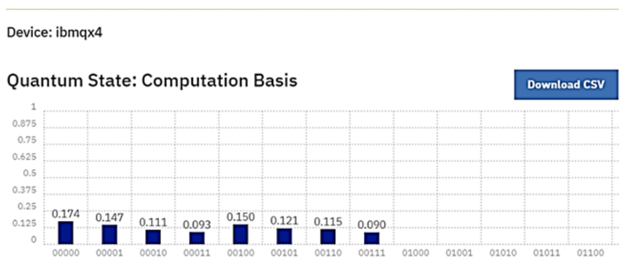


Figure 25: The probability of measuring the correct solution after 1 iteration of algorithm is =77.4%

0.9.1.4 Device: IBM QX4

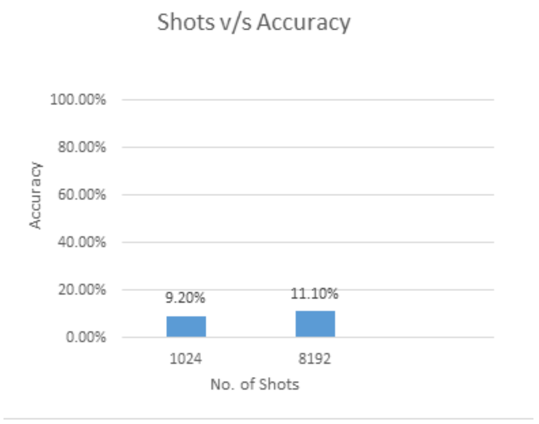


(a) 9.2% with 1024 shots



(b) 11.1% with 8192 shots

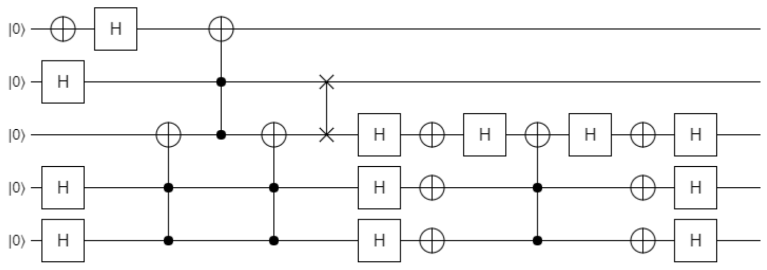
Figure 26: The probability of measuring the correct solution after 1 interation of algorithm

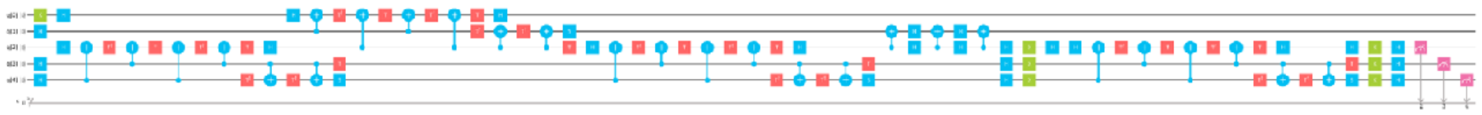


For code refer to Appendix .1.4.1

0.9.2 Optimized Approach for 3 Qubit Grover's Search Implemented for IBM QX4 Architecture

This approach implements the algorithm with the use 1 swap gate.





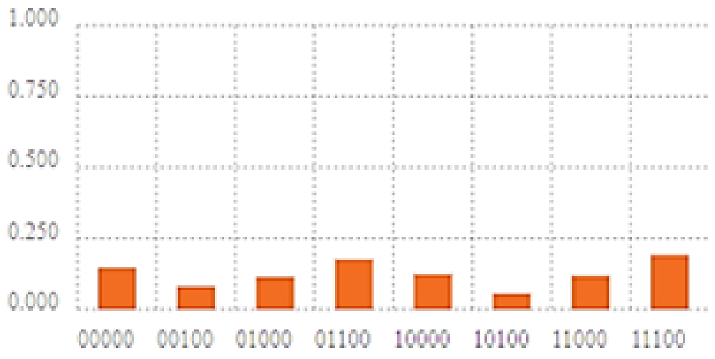


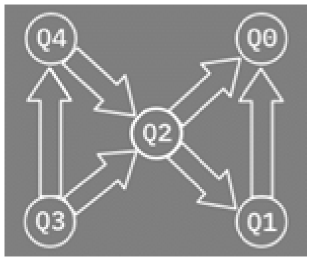
Figure 27: The probability of measuring the correct solution after 1 iteration of algorithm is =21.4% with 8192 shots. (Marked state= $|111\rangle$)

For code refer to Appendix 1.4.2

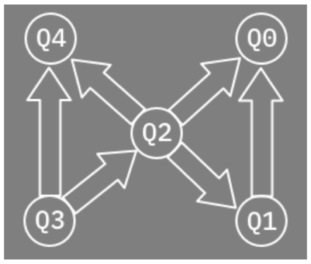
0.9.3 Working of C-Nots in IBM QX4 Architecture (IBM Q5 Tenerife)[20]

The gate directions are given by the following diagram.

0.9.3.1 Direction of 2 Qubit gates



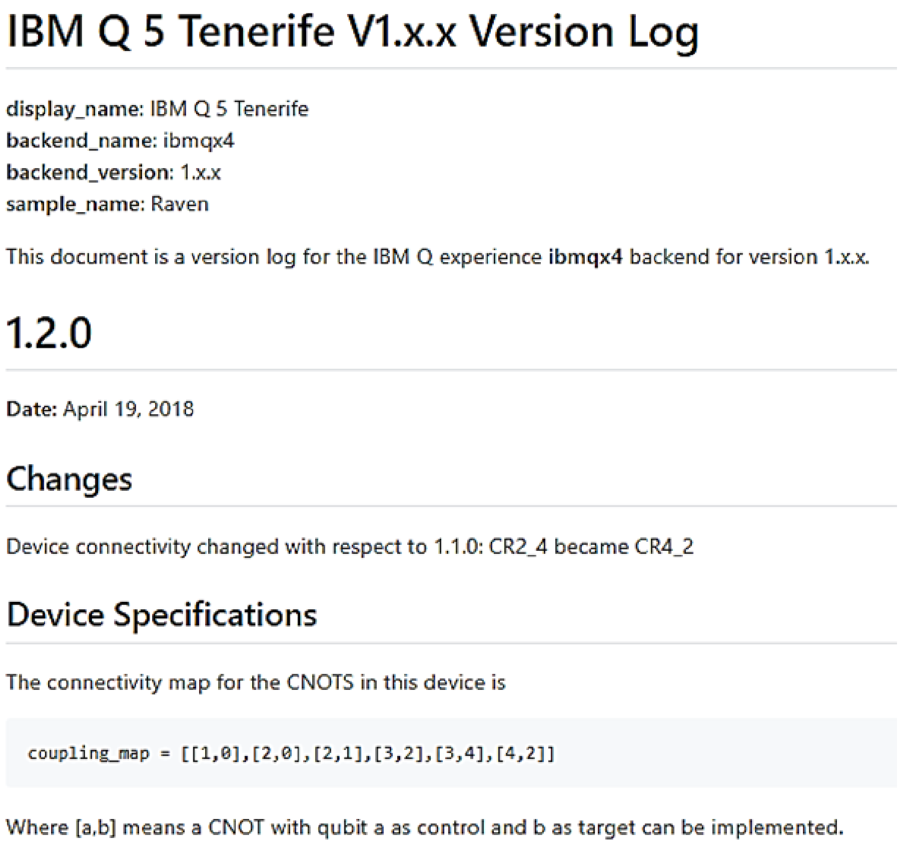
(a) Date: November 12, 2017 (change in direction of q2 to q4)



(b) Date: September 25, 2017

Figure 28: The directions in which the c-nots work on IBM QX4

According to the data obtained frequencies of the qubits are given as follows (in GHz):
Q4:5.18929 Q0: 5.24228 Q1 :5.30745 Q2: 5.35162 Q3:5.41037
Therefore in terms of frequencies: q4<q0<q1<q2<q3
Two-qubit gates are possible between neighboring qubits that are connected by a superconducting bus resonator. The IBM Q experience uses the cross-resonance interaction as the basis for the CX-gate. This interaction is stronger when choosing the qubit with higher frequency to be the control qubit and the lower frequency qubit to be the target, so the frequencies of the qubits determines the direction of the gate.
Since frequency of q4 is less than q2 CX q2, q4 should have been possible which is the not the case due to some exceptions These changes are stated as follows:



IBM Q 5 Tenerife V1.x.x Version Log

display_name: IBM Q 5 Tenerife
backend_name: ibmqx4
backend_version: 1.x.x
sample_name: Raven

This document is a version log for the IBM Q experience **ibmqx4** backend for version **1**.x.x.

1.2.0

Date: April 19, 2018

Changes

Device connectivity changed with respect to 1.1.0: CR2_4 became CR4_2

Device Specifications

The connectivity map for the CNOTS in this device is

```
coupling_map = [[1,0],[2,0],[2,1],[3,2],[3,4],[4,2]]
```

Where [a,b] means a CNOT with qubit a as control and b as target can be implemented.

0.9.3.2 Exceptions to the rule of high frequency control/low frequency target

- the gate direction must be reversed if the higher levels of the control qubit are degenerate with the target qubit,
- or if either qubit is coupled to a third (spectator) qubit that has the same frequency or a higher level with the same frequency as the target. In two cases on the QX5, no gate is possible between neighboring qubits because of degeneracies with spectator qubits that prevent the gate from working in either direction.

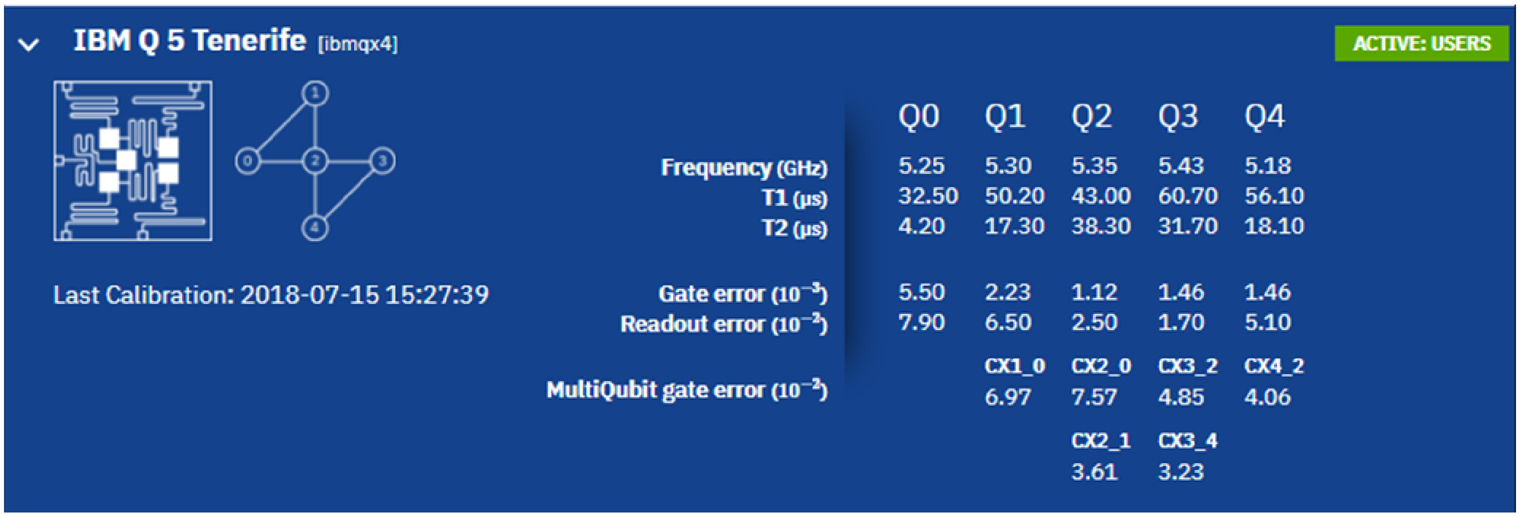
"spectator qubit": Using spectator qubits involves placing another qubit dedicated to spotting small deviations in the local environment close to the qubits actually carrying out the data processing.

0.10 Results obtained for 3 Qubit Grover's Search on IBM Platform

Device	Circuit Design	No of Grover Iterations	No of Gate Stages	Total No. of Gates	Total No. of C-Nots	No.of Shots	Accuracy
IBM Quantum Simulator	Using gates given in quantum experience composer	1	17 (stages are reduced due to derived ccx gates)	84	24	8192	79%
IBM Quantum Simulator	Basic Design modified according to IBM QX4 architecture	1	80	133	39	8192	77.4%
IBM QX4	Optimized Approach using 1 swap gate for IBM QX4	1	80	133	39	1024 8192	9.5 % 11.1 %
IBM QX4	Optimized Approach using 1 swap gate for IBM QX4	1	Minimum stages required=60 After reducing stages by compacting the circuit=49	90 86	27 27	8192 8192	19.05 % 21.4 %

IBM Simulators work fine and show the desired accuracy of 78.125% in one iteration of 3 qubit Grover's algorithm.We can see from the table that by optimizing the circuit and reducing the number of cnots we get a much higher accuracy. We have also removed consecutive hadamards wherever possible to avoid gate errors. Due to different frequencies of the qubits in IBM QX4, we have can apply cnots only in a fixed direction .Hence we have to use swap gates to apply cnots in the reverse direction and hence the required no. of cnots increases to a great deal and the error due to cnots also increases.We have compacted the gate stages as much as possible to improve the efficiency of the final result.

0.11 Back of Envelope Error Analysis of IBM QX4 Device for the 3 Qubit Grover's Search Circuit



IBM Q 5 Tenerife [ibmqx4]

ACTIVE: USERS

Last Calibration: 2018-07-15 15:27:39

	Q0	Q1	Q2	Q3	Q4
Frequency (GHz)	5.25	5.30	5.35	5.43	5.18
T1 (µs)	32.50	50.20	43.00	60.70	56.10
T2 (µs)	4.20	17.30	38.30	31.70	18.10
Gate error (10^{-3})	5.50	2.23	1.12	1.46	1.46
Readout error (10^{-2})	7.90	6.50	2.50	1.70	5.10
		CX1_0	CX2_0	CX3_2	CX4_2
MultiQubit gate error (10^{-2})		6.97	7.57	4.85	4.06
			CX2_1	CX3_4	
			3.61	3.23	

Figure 29: Errors on IBM QX4 Device (IBM Q 5 Tenerife) as posted on 15-07-2018

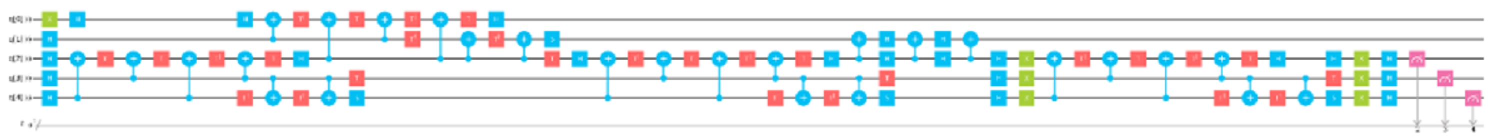


Figure 30: Optimized 3 Qubit Grover's Search Circuit after compacting the stages

No. of CX_1_0 = 2
Error due to CX_1_0= 0.0697
Accuracy in CX_1_0 = 1-0.0697 = 0.9303
Total accuracy due to CX_1_0 = $(0.9303)^2$

No. of CX_2_0 = 2
Error due to CX_2_0= 0.0757
Accuracy in CX_2_0 = 1-0.0757 = 0.9243
Total accuracy due to CX_2_0 =$(0.9243)^2$

No. of CX_3_2 = 6
Error due to CX_3_2= 0.0485
Accuracy in CX_3_2 = 1-0.0485 = 0.9515
Total accuracy due to CX_3_2 = $(0.9515)^6$

No. of CX_4_2 = 6
Error due to CX_4_2= 0.0406
Accuracy in CX_4_2 = 1-0.0406 0.9594
Total accuracy due to CX_4_2 = $(0.9594)^6$
No. of CX_2_1 = 5
Error due to CX_2_1= 0.0361
Accuracy in CX_2_1 = 1-0.0361 = 0.9639

Total accuracy due to CX_2_1 = $(0.9639)^5$

No. of CX_3_4 = 6
Error due to CX_3_4= 0.0323
Accuracy in CX_3_4 = 1-0.0323 = 0.9677
Total accuracy due to CX_3_4 = $(0.9677)^6$

Total accuracy due to cnots=
$(0.9303)^2 \cdot (0.9243)^2 \cdot (0.9515)^6 \cdot (0.9594)^6 \cdot (0.9639)^5 \cdot (0.9677)^6$
=0.29236631651

No. of gates applied to qubit[0] (other than cnots) = 6 (8 gates are applied but only 6 contribute to result)
Error due to q[0] gates= 0.0055
Accuracy in q[0] gates = 1-0.0055 = 0.9945
Total accuracy due to q[0] gates = $(0.9945)^6$

No. of gates applied to qubit[1] (other than cnots) = 6 Error due to q[1] gates= 0.00223
Accuracy in q[1] gates = 1-0.00223 = 0.99777
Total accuracy due to q[1] gates = $(0.99777)^6$

No. of gates applied to qubit[2] (other than cnots) = 25
Error due to q[2] gates= 0.00112
Accuracy in q[2] gates = 1-0.00112 = 0.99888
Total accuracy due to q[2] gates = $(0.99888)^{25}$

No. of gates applied to qubit[3] (other than cnots) = 8
Error due to q[3] gates= 0.00146
Accuracy in q[3] gates = 1-0.00146 = 0.99854
Total accuracy due to q[3] gates = $(0.99854)^8$

No. of gates applied to qubit[4] (other than cnots) = 14
Error due to q[4] gates= 0.00146
Accuracy in q[4] gates = 1-0.00146 = 0.99854
Total accuracy due to q[4] gates = $(0.99854)^{14}$

Total accuracy due to all the single qubit gates
=(Total accuracy due to q[0] gates)· (Total accuracy due to q[1] gates)· (Total accuracy due to q[2] gates)· (Total accuracy due to q[3] gates)· (Total accuracy due to q[4] gates)

$=(0.9945)^6 \cdot (0.99777)^6 \cdot (0.99888)^{25} \cdot (0.99854)^8 \cdot (0.99854)^{14}$
=0.89877695775
Accuracy due to all the gates = (Accuracy due to cnots)· (accuracy due to all single qubit gates)
=0.29236631651· 0.89877695775
=**0.2627721085**

Considering the Readout Errors
Readout Error due to qubit[2] =0.025
Accuracy due to Measure gates applied to qubit[2] = 1-0.0250=0.975

Readout Error due to qubit[3] =0.0170
Accuracy due to Measure gates applied to qubit[3] = 1-0.0170 = 0.983
Readout Error due to qubit[4] =0.0510
Accuracy due to Measure gates applied to qubit[4] = 1-0.0510 = 0.949

Accuracy to all measure gates = 0.975· 0.983· 0.949
=**0.909545325**

Total Accuracy = Accuracy due to all the gates · Accuracy to all measure gates
= 0.2627721085 · 0.909545325
=**0.239003143**

Highest answer predicted by the simulator = 0.791
Least answer expected from device ibmqx4 = 0.7910 · 239003143 = 0.189051486
≈ **19%**
Obtained ans=21.4%

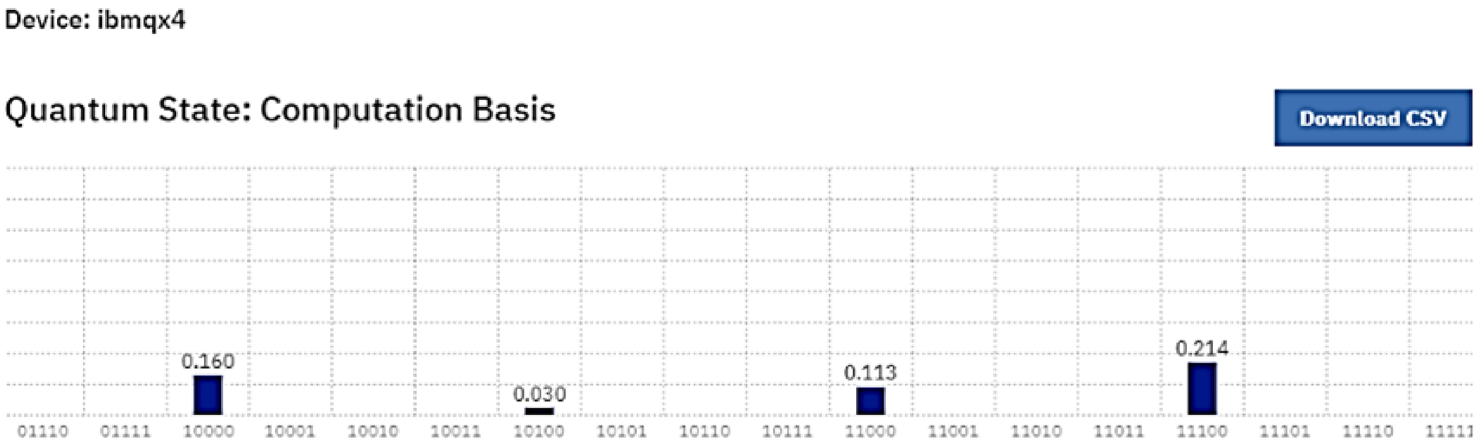


Figure 31: Result obtained on IBM QX4 device for the optimized 3 Qubit Grover's Search Circuit after compacting the stages

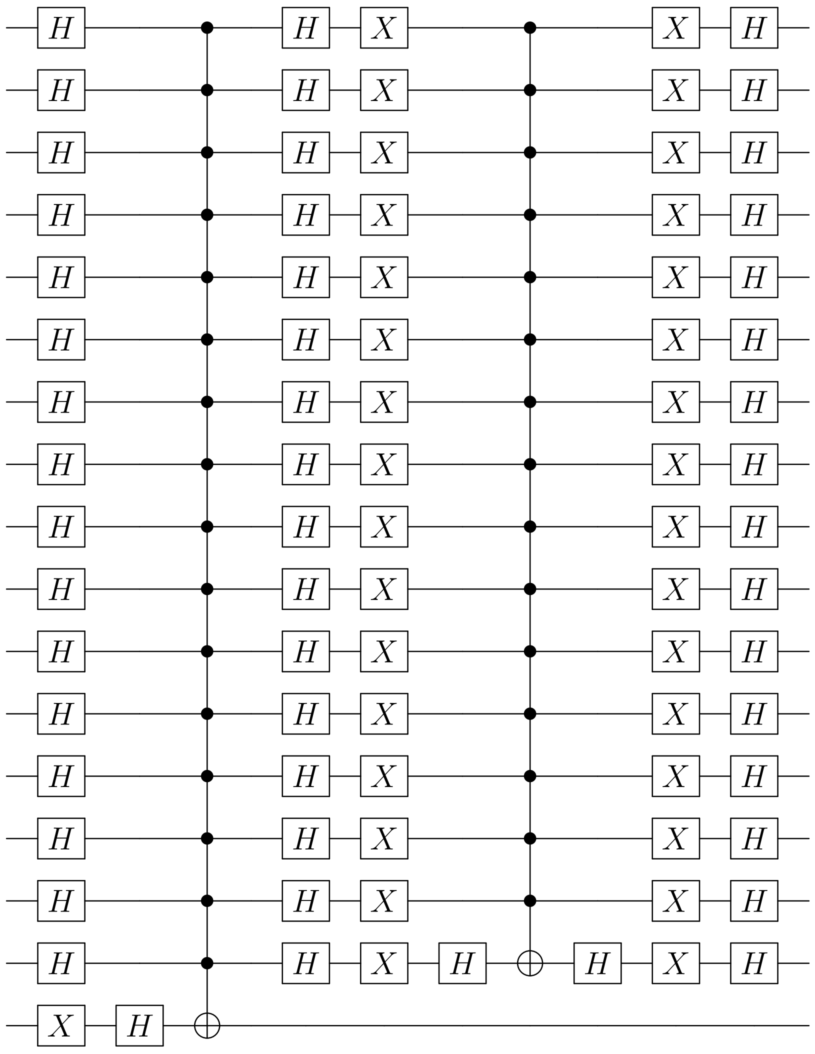
0.12 Implementing 16 Qubits Grover's Search

Simulators used:

1. Microsoft's LIQ$Ui|\rangle$
2. Microsoft's Quantum Development Kit
3. Quirk: Quantum Circuit Simulator

0.12.1 Circuit diagram

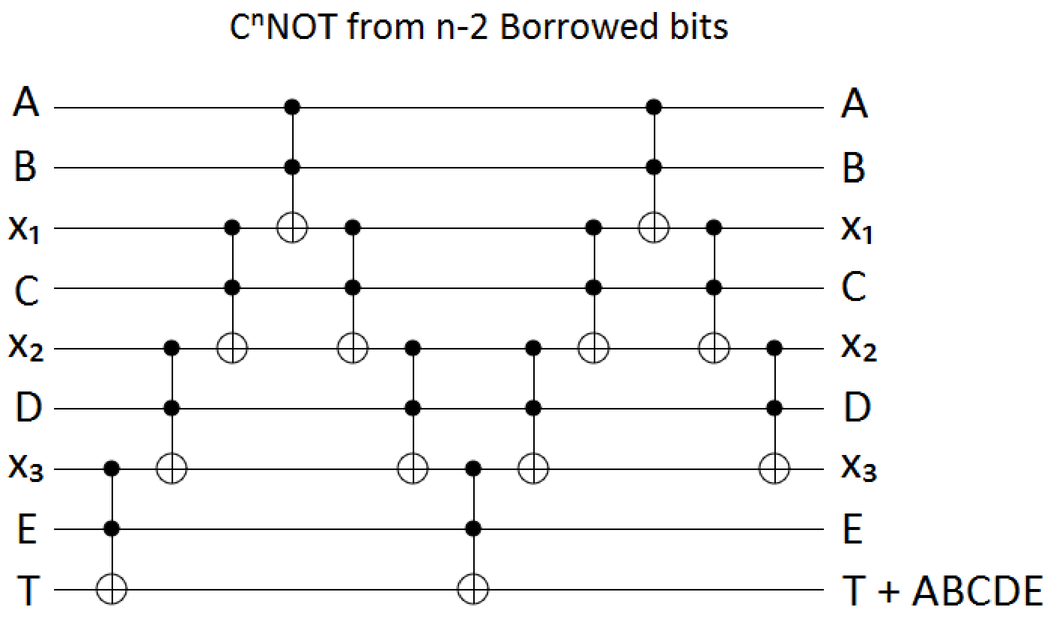
- Total number of possible states = N = $2^n = 2^{16} = 65,536$
- No. of iterations = $\frac{\pi}{4}\sqrt{N} = \frac{\pi}{4}\sqrt{(2^{16})} = 201.06$
- Required state = 1111111111111111 = 65,535



0.12.2 Creating larger CNOTs using toffoli gates [15]

Ancilla bits are extra bits, not involved in the logical operation being performed, that give circuit constructions "room to move". In addition to making constructions possible in the first place, ancilla bits can allow for simpler or more efficient constructions.

The type of Ancilla bits used is Borrowed Bits, where the bits can be in any state beforehand, and must be restored to that same state afterwards. The advantage of using Borrowed Bits is that they can be reused and hence, reduce the number of Qubits required. But, this implementation increases the number of toffoli gates required.



0.12.3 The Oracle

The black box function (f(x)), in this implementation, is designed such that when all the inputs are 1 i.e. state is 1111111111111111, it returns the value 1.

$$f(x^*) = 1$$

where x^* is the desired state.
Using this function, the oracle matrix U_f is designed to act on any of the simple, standard basis states $|x$ *rangle* by

$$U_f|x\rangle = (-1)^{f(x)}|x\rangle$$

Implementation of C^{16}NOT gate (Oracle function) :

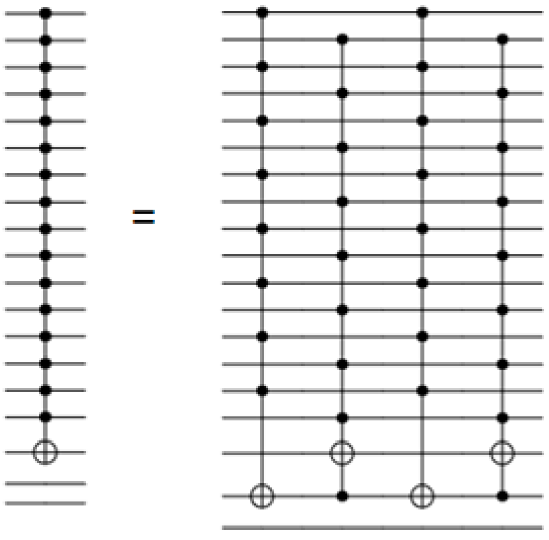


Figure 32: Implementation of $C^{16}NOT$ using smaller CNOT gates

Implementation of C^8NOT and C^9NOT using toffoli gates:

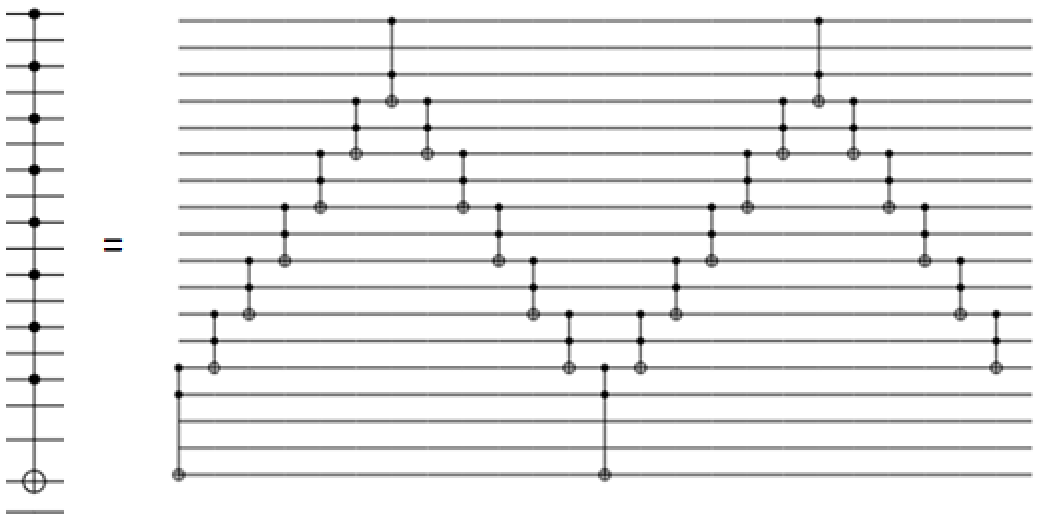


Figure 33: Implementation of C^8NOT using toffoli gates

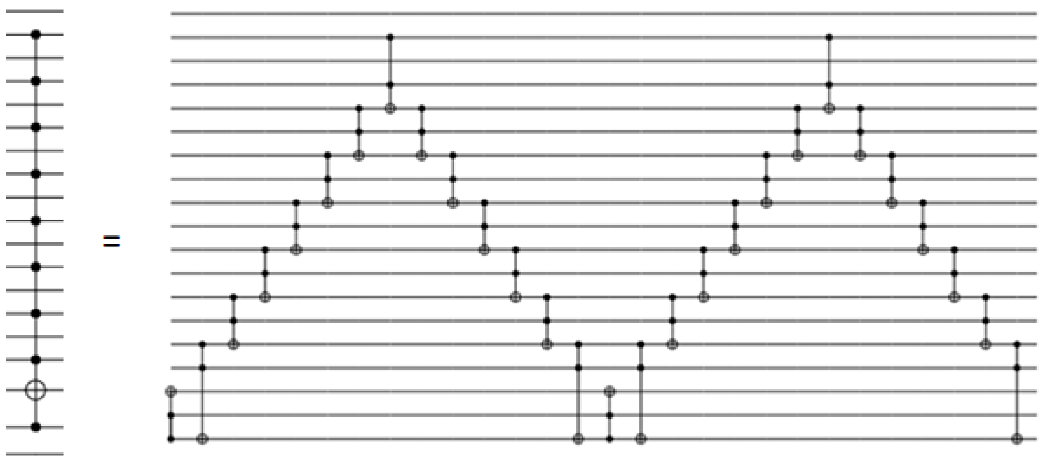
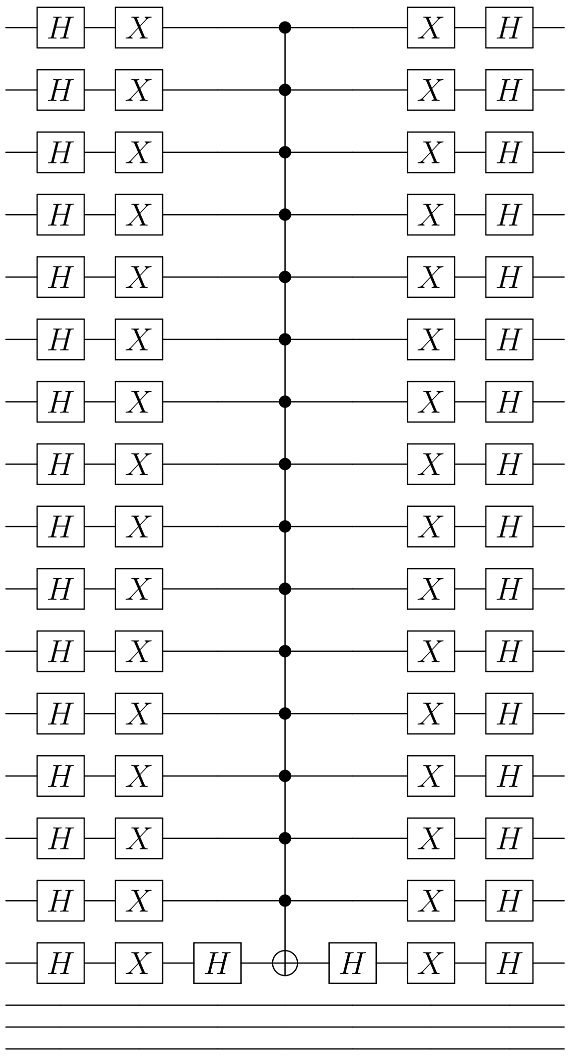


Figure 34: Implementation of C^9NOT using toffoli gates

0.12.4 The Diffusion Operation

Diffusion (Amplitude Amplification) is the procedure by which the quantum computer significantly enhances the probability of the required state

Circuit diagram:



Implementation of $C^{15}NOT$:

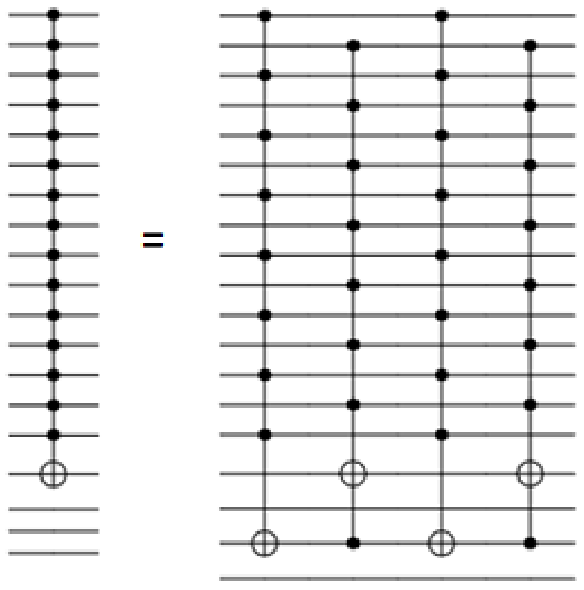


Figure 35: Implementation of $C^{15}NOT$ using smaller CNOT gates

Implementation of C^8NOT using toffoli gates:

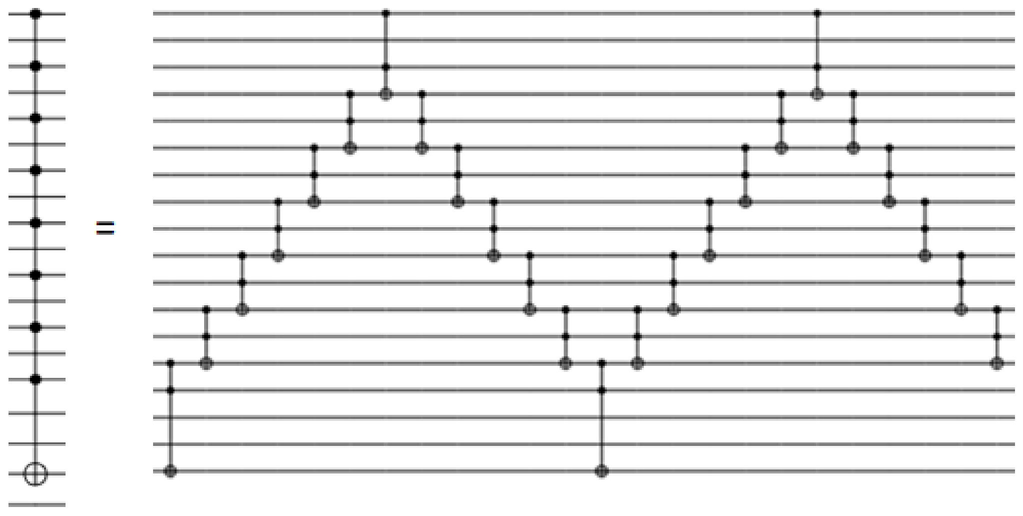


Figure 36: Implementation of C^8NOT using toffoli gates

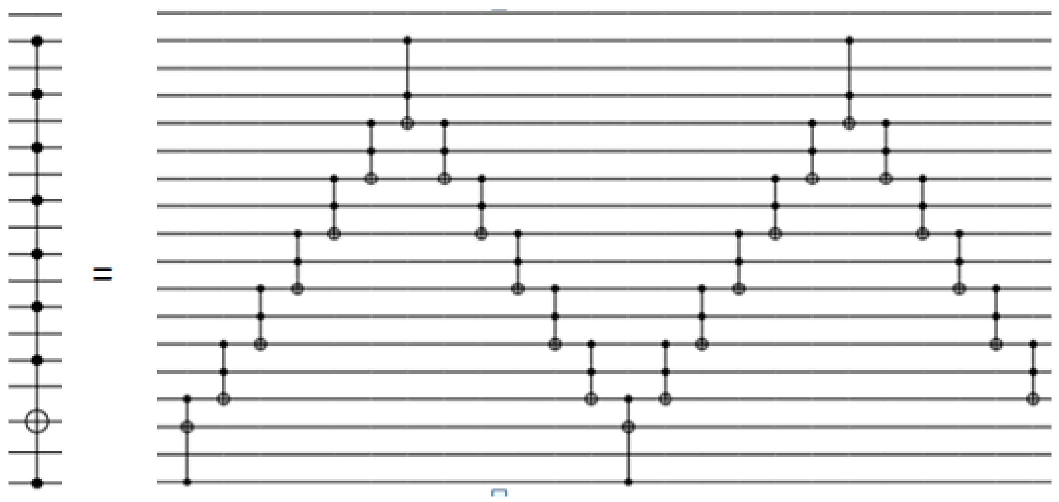


Figure 37: Implementation of C^8NOT using toffoli gates

0.12.5 Implementation of 16 Qubits Grover's search on Microsoft's LIQ$\boldsymbol{Ui}|\rangle$ simulator

LIQ$Ui|\rangle$[14] is a simulation platform developed by the Quantum Architectures and Computation team at Microsoft Research to aid in the exploration of quantum computation. LIQ$Ui|\rangle$ stands for "Language Integrated Quantum Operations".

Programming language used : F#

0.12.5.1 Result on compiling and rendering the circuit

{For code refer to Appendix .1.5 }

Circuit diagrams :

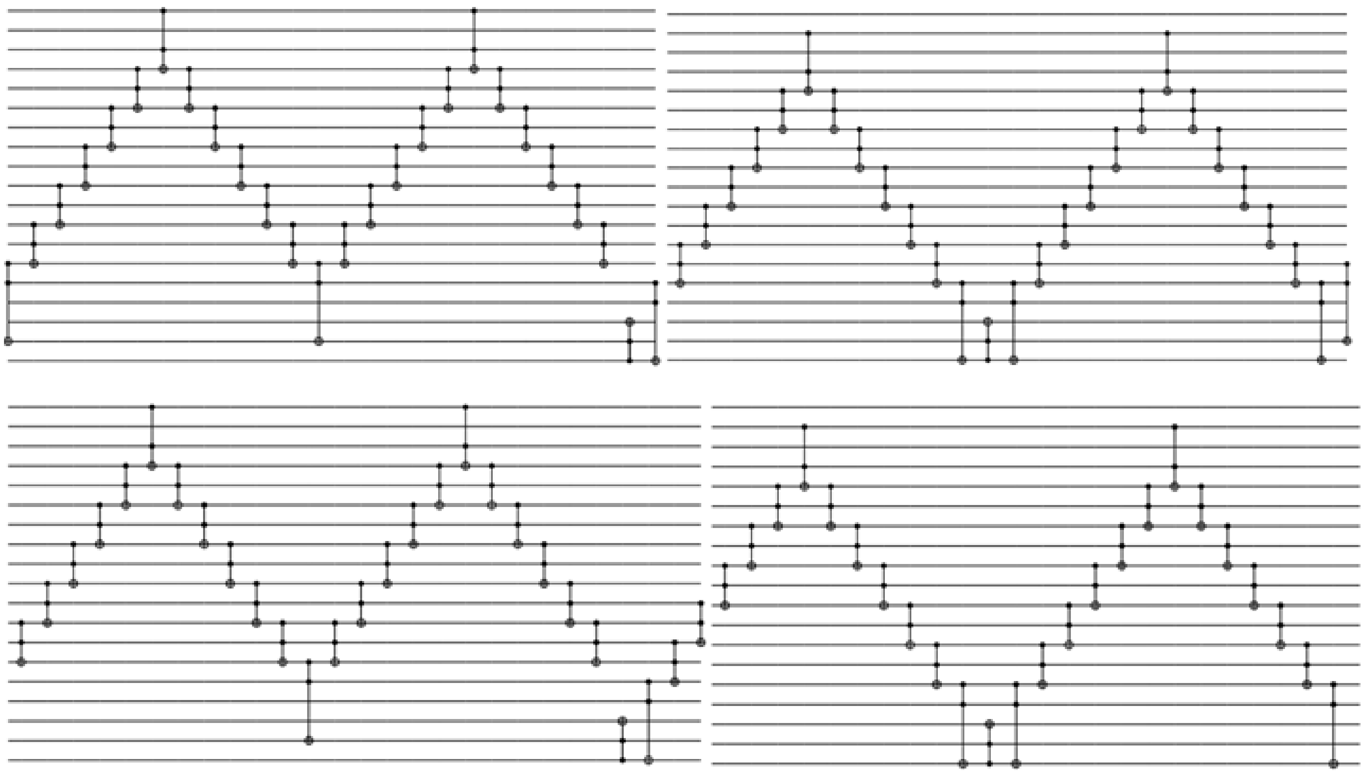


Figure 38: Oracle circuit diagram

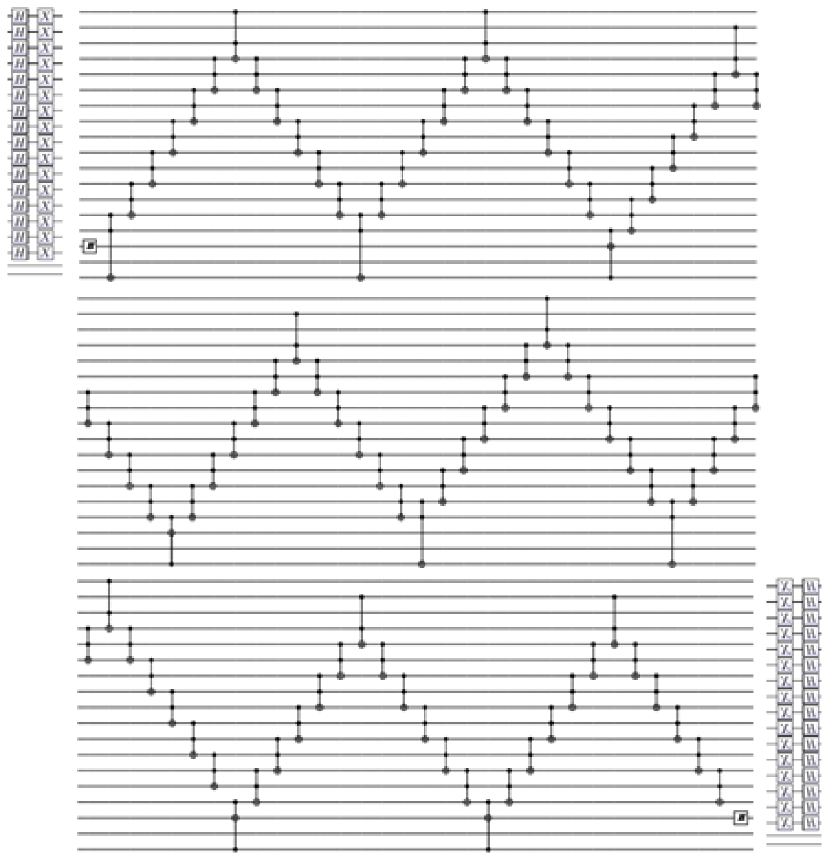
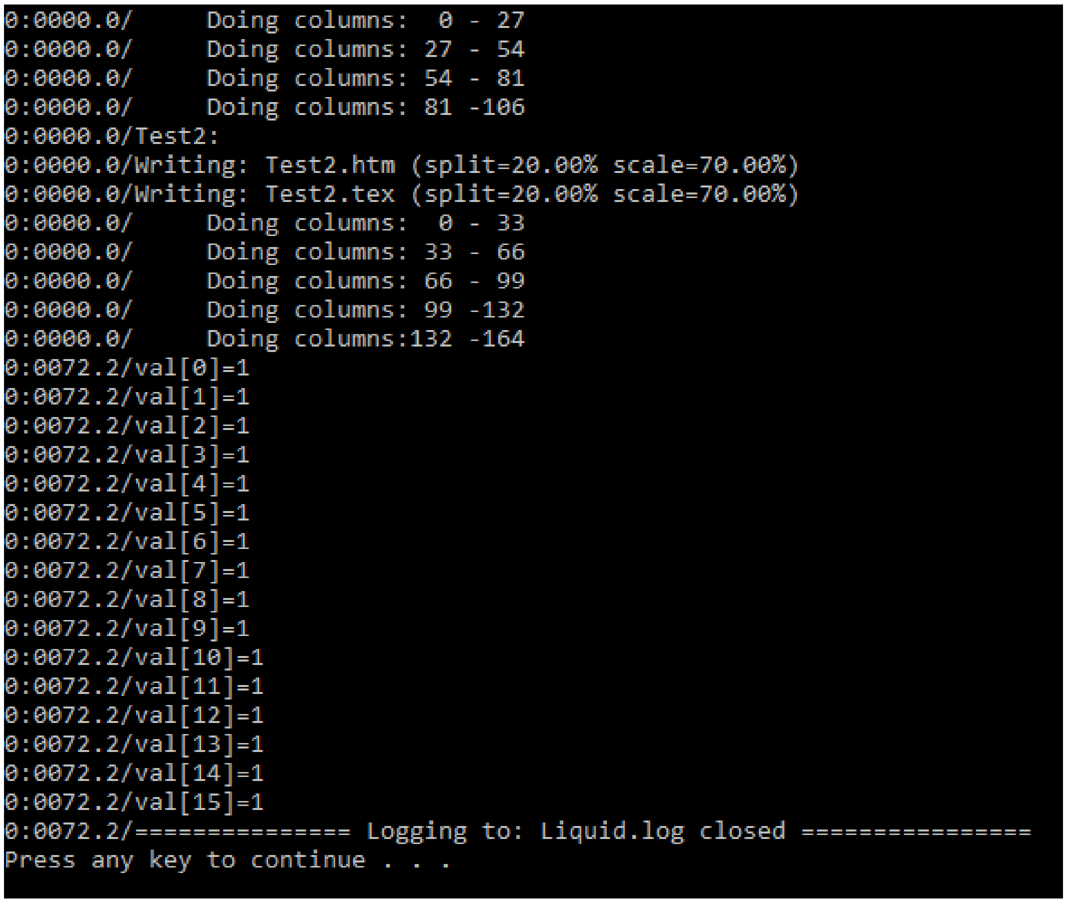


Figure 39: Diffusion circuit diagram

Output :

Desired result : 1111111111111111
Time taken : 72.2 minutes



```
0:0000.0/     Doing columns:  0 - 27
0:0000.0/     Doing columns: 27 - 54
0:0000.0/     Doing columns: 54 - 81
0:0000.0/     Doing columns: 81 -106
0:0000.0/Test2:
0:0000.0/Writing: Test2.htm (split=20.00% scale=70.00%)
0:0000.0/Writing: Test2.tex (split=20.00% scale=70.00%)
0:0000.0/     Doing columns:  0 - 33
0:0000.0/     Doing columns: 33 - 66
0:0000.0/     Doing columns: 66 - 99
0:0000.0/     Doing columns: 99 -132
0:0000.0/     Doing columns:132 -164
0:0072.2/val[0]=1
0:0072.2/val[1]=1
0:0072.2/val[2]=1
0:0072.2/val[3]=1
0:0072.2/val[4]=1
0:0072.2/val[5]=1
0:0072.2/val[6]=1
0:0072.2/val[7]=1
0:0072.2/val[8]=1
0:0072.2/val[9]=1
0:0072.2/val[10]=1
0:0072.2/val[11]=1
0:0072.2/val[12]=1
0:0072.2/val[13]=1
0:0072.2/val[14]=1
0:0072.2/val[15]=1
0:0072.2/============== Logging to: Liquid.log closed ================
Press any key to continue . . .
```

Figure 40: Result on compiling and executing the circuit

0.12.5.2 Result on optimizing the circuit

LIQ$Ui|\rangle$ has method, GrowGates() to optimize the circuit by transforming it into more efficient unitary matrices.

Circuit diagrams :

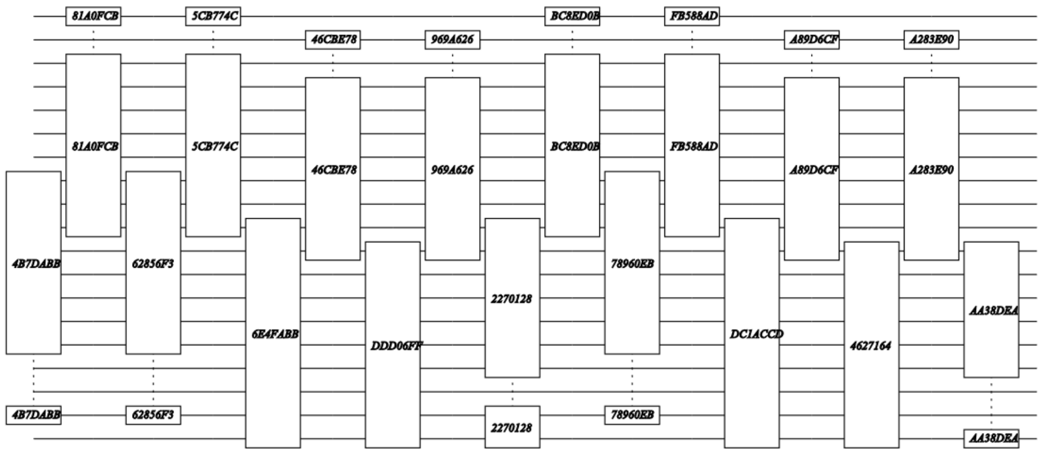


Figure 41: Optimized oracle circuit diagram

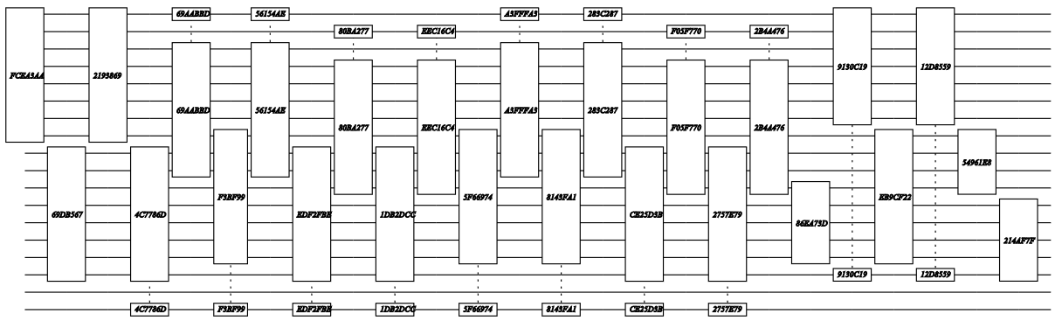
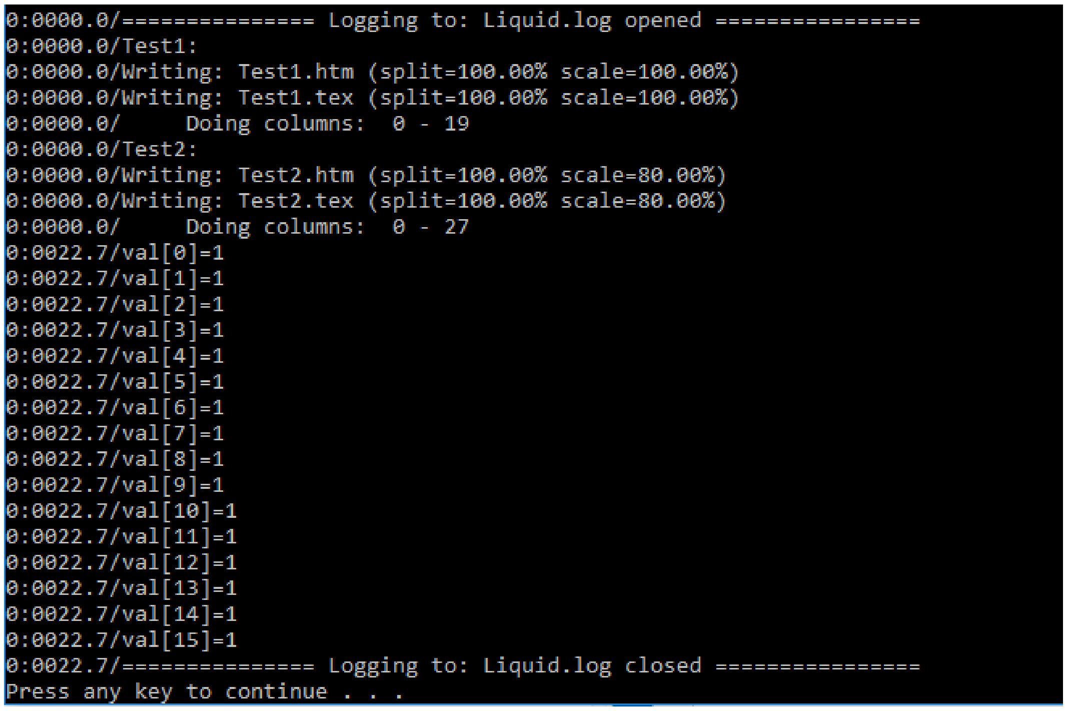


Figure 42: Optimized diffusion circuit diagram

Output :

Desired result : 1111111111111111
Time taken : 22.7 minutes



```
0:0000.0/=============== Logging to: Liquid.log opened ================
0:0000.0/Test1:
0:0000.0/Writing: Test1.htm (split=100.00% scale=100.00%)
0:0000.0/Writing: Test1.tex (split=100.00% scale=100.00%)
0:0000.0/     Doing columns:  0 - 19
0:0000.0/Test2:
0:0000.0/Writing: Test2.htm (split=100.00% scale=80.00%)
0:0000.0/Writing: Test2.tex (split=100.00% scale=80.00%)
0:0000.0/     Doing columns:  0 - 27
0:0022.7/val[0]=1
0:0022.7/val[1]=1
0:0022.7/val[2]=1
0:0022.7/val[3]=1
0:0022.7/val[4]=1
0:0022.7/val[5]=1
0:0022.7/val[6]=1
0:0022.7/val[7]=1
0:0022.7/val[8]=1
0:0022.7/val[9]=1
0:0022.7/val[10]=1
0:0022.7/val[11]=1
0:0022.7/val[12]=1
0:0022.7/val[13]=1
0:0022.7/val[14]=1
0:0022.7/val[15]=1
0:0022.7/=============== Logging to: Liquid.log closed ================
Press any key to continue . . .
```

Figure 43: Result on optimizing the circuit

0.12.6 Implementation of 16 Qubits Grover's search on Microsoft's Quantum Development Kit (QDK)

The Quantum Development Kit , developed by Microsoft, is based on Q# language coding and includes a Q# compiler, Q# libraries, a local quantum computing simulator, a quantum trace simulator and a Visual Studio extension for quantum computing projects.

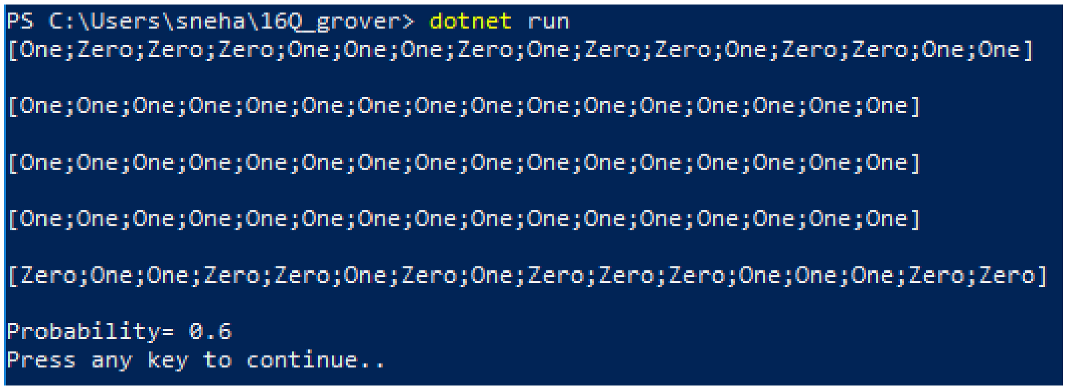
Programming languages used: Q# and C#

{ For code refer to Appendix .1.6}

Output :

The circuit is simulated five times and based on the outputs, the probability of getting the desired state (1111111111111111) is calculated.

Desired result : 1111111111111111
Time taken : 1.34 minutes



```
PS C:\Users\sneha\16Q_grover> dotnet run
[One;Zero;Zero;Zero;One;One;One;Zero;One;Zero;Zero;One;Zero;Zero;One;One]

[One;One;One;One;One;One;One;One;One;One;One;One;One;One;One;One]

[One;One;One;One;One;One;One;One;One;One;One;One;One;One;One;One]

[One;One;One;One;One;One;One;One;One;One;One;One;One;One;One;One]

[Zero;One;One;Zero;Zero;One;Zero;One;Zero;Zero;Zero;One;One;One;Zero;Zero]

Probability= 0.6
Press any key to continue..
```

Figure 44: Result on executing the circuit

0.12.7 Implementation on Quirk quantum simulator

0.12.7.1 Grover gate :

Oracle : The 16 bit controlled Z-gate flips the amplitude of the required state (here $|1111111111111011\rangle$)
Diffusion : It is implemented using 16 bit controlled-not gate [1]

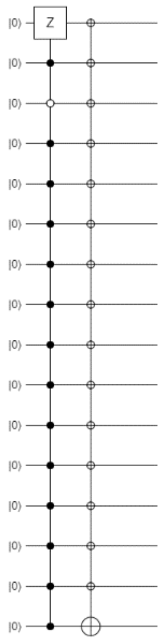


Figure 45: Grover gate

0.12.7.2 Circuit diagram :

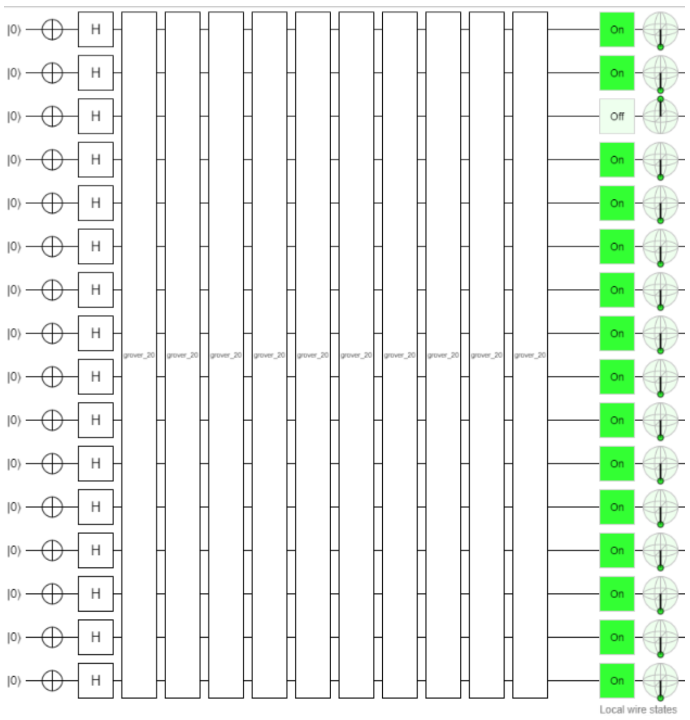


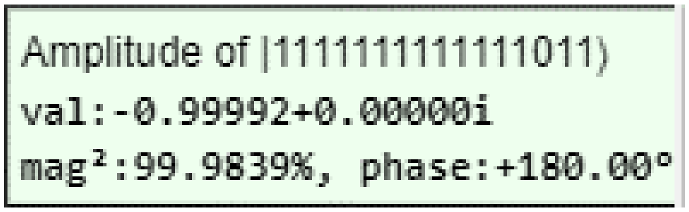
Figure 46: Circuit diagram

[1]Note that HXH = Z and HZH = X, and diffusion operator = HZH

0.12.7.3 Result :

grover_20 gate is 20 times the Grover gate applied repeatedly

Results after 400 iterations :



```
Amplitude of |1111111111111011⟩
val:-0.99992+0.00000i
mag²:99.9839%, phase:+180.00°
```

Figure 47: simulation output

0.13 Conclusions

While quantum computing has dramatically advanced through the last decade, its potential applications have not yet been demonstrated at full scale. Such demonstrations are likely to require breakthroughs in physics, computer science and engineering[4].The Quantum computers are prone to errors due to quantum coherence and environmental conditions. To this end, we have analyzed the potential of Grover's search algorithm to provide quadratic speed up over classical search algorithm, for certain problems and have executed it for 3 Qubits on IBMQX4 , developed by IBM. We got the desired state with a probability of 22.4% (for one iteration), which is much less than the theoretical probability of 78.125%. Thus, at the moment, the gate errors on the public devices made available by IBM are quite high. Finally, we have executed Grover's search for 16 Qubits on Microsoft's LIQ$Ui|\rangle$, Microsoft's Quantum Development Kit and Quirk simulator and observed the following:

Results obtained for Implementation of 16 Qubit Grover's Search on various Platforms			
Device	**No. of Grover Iterations**	**Time Taken**	**Accuracy**
Microsoft's LIQUi⟩ Simulator	201	22.7 minutes	100%
Microsoft Quantum Development Kit	201	1.34 minutes	60%
Quirk Quantum Simulator	200	instant	99.9839%

Simple compilation and execution of 16 Qubits Grover's search took 72.2 minutes on Microsoft's LIQ$Ui|\rangle$ simulator. But, on optimizing the circuit using the inbuilt method GrowGates(), the simulation took merely 22.7 minutes, that is, the circuit executes at a 31.44% faster rate. Also, the simulator provides a very high accuracy rate of almost cent percent and returned the desired state ($|1111111111111111\rangle$) as the result at each test run. On the other hand, Microsoft's Quantum development kit provided less accurate results. After simulating the circuit on five consecutive trials, the desired state ($|1111111111111111\rangle$) was obtained only three times which translates accuracy to just 60%. It is, however, observed that operations take much less time while using this simulator if one can compromise with low accuracy. Compared to these Quirk Simulator gave almost instant results but it was restricted to a maximum of 16 qubits. Since the Quirk toolbox provided an easy tofolli implementation we did not have to worry about its equivalent circuit using basic quantum gates. However this implementation was close to the theoritical model of the circuit rather than being close to actual hardware.

0.14 Limitations

Coherence issues where device must constantly fight against the environment which acts to degrade the coherence of the system. And hence delicate quantum information stored in a quantum computer is extremely susceptible to noise. Qubits must be kept cold or else they collapse easily. Error due to qubit entanglement are high. Multiqubit gate errors were much higher than single qubit gate errors.
Limited Connectivity among the qubits is another major limitation. Hence there is a need for employing a large no. of swap gates which increases the gate count (thus adding to gate errors) especially the cnots which have a quite high errors and thus reducing the expected state fidelity by a great deal and also circuits become complex and difficult to comprehend and debug. IBM Simulators already have limited code length (comparatively LIQ$Ui|\rangle$ and Quirk Quantum Simulator have relaxed code lengths) hence it becomes difficult to implement large circuits or enlarge the circuits to add any error correction gates or resolve the problem of limited connectivity by adding more swap gates.
Due to difference in the frequency of qubits in IBM devices the direction of applying cnots is limited to specific qubits which also increases our dependency on swap gates to a great deal.Also we cannot do stage by stage analysis on IBM devices. While working on the real IBM devices sending our code to run on actual devices , a long queue of execution generally results in a timeout error while working with QISKit. While the composer ensures results even though if they are delayed it has limited stages due to which quantum algorithms with large number of stages cannot be executed. Implementing the tofolli gate for 16 qubits required many extra qubits or a large number of stages both of which is not possible on the current IBM devices and hence implementing 16 qubit Grover's search on the IBM simulators and real devices was not realistic.

References

[1] Charles H Bennett et al. "Strengths and weaknesses of quantum computing". In: *SIAM journal on Computing* 26.5 (1997), pp. 1510–1523.

[2] Michel Boyer et al. "Tight bounds on quantum searching". In: *Fortschritte der Physik: Progress of Physics* 46.4-5 (1998), pp. 493–505.

[3] Isaac L Chuang, Neil Gershenfeld, and Mark Kubinec. "Experimental implementation of fast quantum searching". In: *Physical review letters* 80.15 (1998), p. 3408.

[4] John Preskill. "Quantum computing: pro and con". In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*. Vol. 454. 1969. The Royal Society. 1998, pp. 469–486.

[5] Lieven MK Vandersypen et al. "Implementation of a three-quantum-bit search algorithm". In: *Applied Physics Letters* 76.5 (2000), pp. 646–648.

[6] N Bhattacharya, HB van Linden van den Heuvell, and RJC Spreeuw. "Implementation of quantum search algorithm using classical Fourier optics". In: *Physical review letters* 88.13 (2002), p. 137901.

[7] Quantum logic gate. *Quantum logic gate — Wikipedia, The Free Encyclopedia*. Accessed: 2018-07-20. 2004. URL: `https://en.wikipedia.org/wiki/Quantum_logic_gate`.

[8] K-A Brickman et al. "Implementation of Grover's quantum search algorithm in a scalable system". In: *Physical Review A* 72.5 (2005), p. 050306.

[9] Philip Walther et al. "Experimental one-way quantum computing". In: *Nature* 434.7030 (2005), p. 169.

[10] Christoph Dürr et al. "Quantum query complexity of some graph problems". In: *SIAM Journal on Computing* 35.6 (2006), pp. 1310–1328.

[11] Xiling Xue et al. "On the Research of BDD Based Simulation of Grover's Algorithm". In: *Genetic and Evolutionary Computing, 2008. WGEC'08. Second International Conference on*. IEEE. 2008, pp. 459–462.

[12] L DiCarlo et al. "Demonstration of two-qubit algorithms with a superconducting quantum processor". In: *Nature* 460.7252 (2009), p. 240.

[13] Klaus Mølmer, Larry Isenhower, and Mark Saffman. "Efficient Grover search with Rydberg blockade". In: *Journal of Physics B: Atomic, Molecular and Optical Physics* 44.18 (2011), p. 184016.

[14] Dave Wecker and Krysta M. Svore. *LIQUi—¿: A Software Design Architecture and Domain-Specific Language for Quantum Computing*. 2014. eprint: `1402.4467`. URL: `arXiv:1402.4467v1`.

[15] Craig Gidney. *Constructing Large Controlled Nots*. `http://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Nots.html`. Blog. 2015.

[16] Shantanu Debnath et al. "Demonstration of a small programmable quantum computer with atomic qubits". In: *Nature* 536.7614 (2016), p. 63.

[17] C Figgatt et al. “Complete 3-qubit Grover search on a programmable quantum computer”. In: *Nature communications* 8.1 (2017), p. 1918.

[18] Norbert M Linke et al. “Experimental comparison of two quantum computing architectures”. In: *Proceedings of the National Academy of Sciences* 114.13 (2017), pp. 3305–3310.

[19] Archana Tankasala, Hesameddin Ilatikhameneh. *Q-Kit*. `https://sites.google.com/view/quantum-kit/`.

[20] *IBM Q 5 Tenerife V1.x.x Version Log*. `https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=beginners-guide&page=005-Single-Qubit_Gates~2F006-Summary_of_quantum_gates`. Accessed: 2018-21-07.

[21] Microsoft. *Quantum Development Kit*. `https://www.microsoft.com/en-in/quantum/development-kit`. Accessed: 2018-06-23.

[22] *Proof of Gate Complexity*. `https://www.cs.cmu.edu/~odonnell/quantum15/lecture04.pdf`.

[23] *Quirk Quantum Simulator*. `https://algassert.com/quirk`.

[24] *Summary of Quantum Gates IBM Q Experience*. `https://quantumexperience.ng.bluemix.net/qx/tutorial?sectionId=beginners-guide&page=005-Single-Qubit_Gates~2F006-Summary_of_quantum_gates`. Accessed: 2018-19-07.

.1 Appendix

.1.1 Proof of Optimal No. of Iterations

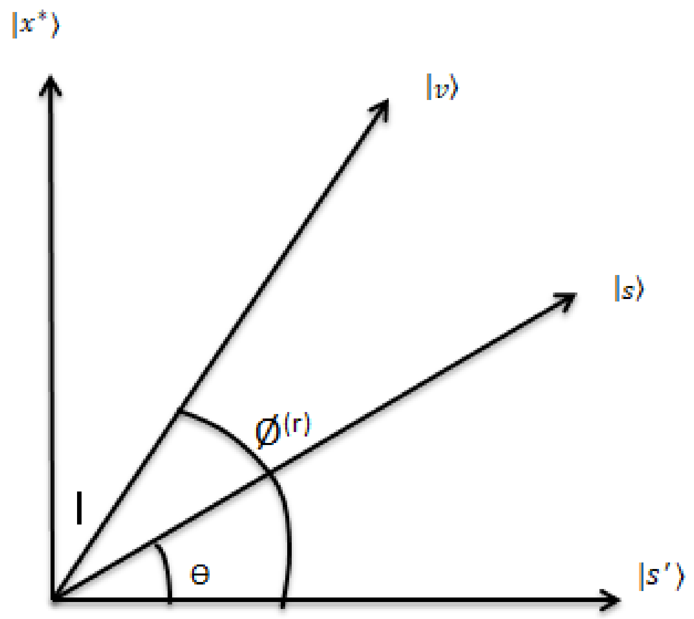


Figure 48: Geometric representation of Grover's search

After r iterations of Oracle and Diffusion subroutines, we get the relation (Figure: 48)

$$\phi^{(r)} = (2r+1)\theta \tag{1}$$

Now, for any vector $|v\rangle$ spanned by the orthonormal bases $|x^*\rangle$ and $|s^{'}\rangle$, can be represented by,

$$|v\rangle = \cos\phi^{(r)}|s^{'}\rangle + \sin\phi^{(r)}|x^*\rangle$$

So, the probability of $|x^*\rangle$ after r iterations of the Oracle and diffusion subroutines is given by,

$$P(|x^*\rangle) = \sin^2\phi^{(r)}$$

Or, we need to maximize $\sin\phi^{(r)}$. We get the maximum value

$$\sin\phi^{(r)} = 1$$

at

$$\phi^{(r)} = \pi/2 \tag{2}$$

Using equations (1) and (2),

$$(2r+1)\theta = \pi/2 \tag{3}$$

we get,

$$\theta = \sin^{-1}(\sqrt{1/2^n})^2 \tag{4}$$

Using (4) in (1), we get

$$\phi^{(r)} = (2r+1)\sin^{-1}(\sqrt{1/2^n}) = \pi/2 \tag{5}$$

For very large n, $(n \to \infty)$,

$$\sin^{-1}(\sqrt{1/2^n}) \approx \sqrt{1/2^n} \tag{6}$$

Using (6) we get,

$$(2r+1)\sqrt{1/2^n} = \pi/2$$

or,

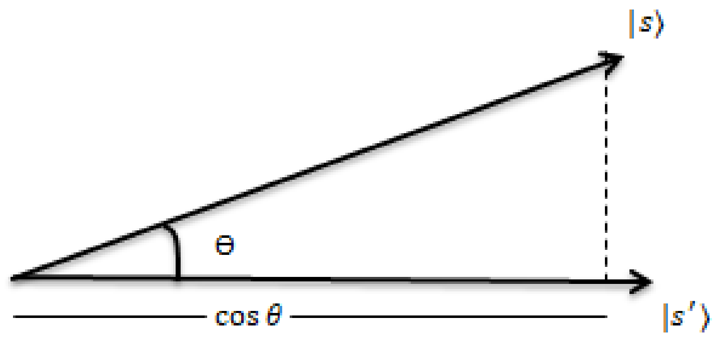
$$r = \frac{(\frac{\pi}{2}\sqrt{2^n} - 1)}{2}$$

for large n,

$$\boxed{r = \frac{\pi}{4}\sqrt{2^n}}$$

Thus, $\frac{\pi}{4}\sqrt{2^n}$ iterations of oracle and diffusion operations are required to achieve maximum probability of desired result.

.1.2 Derivation of angle θ between uniform superposition $|s\rangle$ and component of uniform superposition $|s'\rangle$ normal to $|x^*\rangle$



Calculating the projection of $|s\rangle$ onto $|s'\rangle$,

$$= (|s'\rangle\langle s|')|s\rangle$$

$$= \langle s'|s\rangle|s'\rangle$$

Now, from the figure, the inner product $\langle s'|s\rangle$ is given by,

$$\cos\theta = \langle s'|s\rangle \tag{7}$$

[2]Proof in Appendix .1.2

The inner product $\langle s'|s\rangle$ can be calculated as,

$$\langle s'|s\rangle = (2^n - 1)(\frac{1}{\sqrt{2^n - 1}})(\frac{1}{\sqrt{2^n}}) \tag{8}$$

using equations (7) and (8), we get

$$\cos\theta = \sqrt{\frac{2^n - 1}{2^n}}$$

or,

$$\sin\theta = \sqrt{\frac{1}{2^n}}$$

or,

$$\boxed{\theta = \sin^{-1}\sqrt{\frac{1}{2^n}}} \tag{9}$$

.1.3 Proof of Gate Complexity [22]

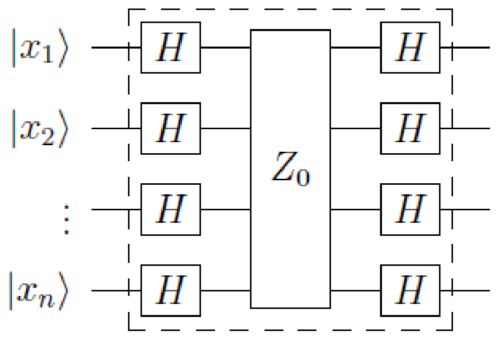
We also promised an $O(\sqrt{N}logN)$ bound on the number of gates used in the algorithm. To prove this bound, it suffices to construct the Grover diffusion gate. To do so, we first define the Z_0 defned by

$$Z_0|x\rangle = \begin{Bmatrix} |x\rangle & if\ |x\rangle = |0^n\rangle \\ -|x\rangle & otherwise \end{Bmatrix}$$

In unitary matrix form, $Z_0 = 2|0^n\rangle\langle 0^n| - I$ where I is the n x n identity matrix.
Now we can construct the Grover diffusion gate, D, with a Z_0 gate and O(n) = O(logN) Hadamard gates. In matrix form, knowing that the Hadamard gate is its own inverse, we can write the diffusion gate as

$$\begin{aligned} D =& H^{\bigotimes n} Z_0 H^{\bigotimes n} \\ =& H^{\bigotimes n}\left(2|0^n\rangle\langle 0^n| - I\right) H^{\bigotimes n} \\ =& 2\left((H|0^n\rangle)^{\bigotimes n} \left((H|0^n\rangle)^{\bigotimes n} \right)^{\dagger} \right) - H^{\bigotimes n} H^{\bigotimes n} \\ =& 2|+^n\rangle\langle +^n| - I \end{aligned}$$

It is the same as Z_0 but with $|+\rangle$ instead of $|0\rangle$. In diagram form, it is



Let us try giving the matrix an arbitrary input $|\psi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x|x\rangle$ to make sure it works. We get

$$D|\psi\rangle = \left(2|+^n\rangle\langle +^n| - I\right)|\psi\rangle$$

$$= (2|+^n\rangle\langle +^n|\psi\rangle - |\psi\rangle)$$

Notice that we can rewrite $|+\rangle$ to get

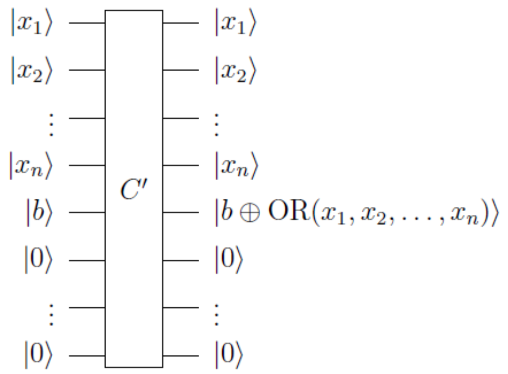
$$\begin{aligned}\langle +^n| &= \left(\frac{\langle 0| + \langle 1|}{\sqrt{2}}\right)^{\otimes n}\\ &= \sum_{x\in\{0,1\}^n} \frac{1}{\sqrt{N}}\langle x|,\end{aligned}$$

$$\begin{aligned}\langle +^n|\psi\rangle &= \sum_{x\in\{0,1\}^n} \frac{\alpha_x}{\sqrt{N}}\langle x|x\rangle\\ &= \sum_{x\in\{0,1\}^n} \frac{\alpha_x}{\sqrt{N}}\\ &=\mu\sqrt{N}\end{aligned}$$

That means that we can continue simplifying $D\,|\psi\rangle$ to get

$$\begin{aligned}D|\psi\rangle &=2|+^n\rangle\langle +^n|\psi\rangle - |\psi\rangle\\ &=2|+^n\rangle\mu\sqrt{N} - |\psi\rangle\\ &=2\left(\frac{\langle 0| + \langle 1|}{\sqrt{2}}\right)\mu\sqrt{N} - |\psi\rangle\\ &=2\left(\sum_{x\in\{0,1\}^n} \frac{1}{\sqrt{N}}|x\rangle\right)\mu\sqrt{N} - |\psi\rangle\\ &=2\left(\sum_{x\in\{0,1\}^n} \mu|x\rangle\right) - |\psi\rangle\\ &=\sum_{x\in\{0,1\}^n} (2\mu - \alpha_x)\,|x\rangle\end{aligned}$$

which is precisely what we expect from the diffusion operator. Lastly, we need to show that we actually can construct the Z_0 gate. What does Z_0 do? It negates $|x\rangle$ if and only if the logical OR of $\{x_1, x_2, ...x_n\}$ is 1, where x_i is the i-th qubit that makes up $|x\rangle$. These kinds of logical operations are easy to compute. In particular,there is a classical circuit that computes the logical OR of n bits with $O(n) = O(logN)$ Toffoli or CNOT gates. This circuit is not reversible, but of course we can fix that. Given a circuit C that computes the logical OR of n bits, we can construct a circuit C' with $m \in N$ ancillary bits that does the following:



Now if we send $|x\rangle \otimes |-\rangle \otimes |0^m\rangle$ into C', we get $(-1)^{OR(x)} (|x\rangle \otimes |-\rangle \otimes |0^m\rangle)$ out, which is exactly what Z_0 does. There are some extra garbage bits, but those can be ignored. Ultimately, what this all means is that the Grover diffusion gate can be constructed with O(logN) gates. That means that Grover's algorithm takes $O(\sqrt{N} logN)$ gates to implement.

.1.4 QASM Code for 3 qubit Grover's Algortitm

.1.4.1 QASM Code for Basic 3 Qubit Grover's Algorithm in IBM Simulator

```
include ”qelib1.inc”;
qreg q[5];
creg c[5];
h q[0];
h q[1];
h q[2];
h q[3];
x q[4];
cx q[1],q[0];
x q[2];
x q[3];
h q[4];
tdg q[0];
cx q[2],q[0];
t q[0];
cx q[1],q[0];
tdg q[0];
t q[1];
cx q[2],q[0];
t q[0];
cx q[2],q[1];
h q[0];
tdg q[1];
t q[2];
cx q[2],q[1];
cx q[2],q[0];
h q[0];
h q[2];
cx q[2],q[0];
h q[4];
h q[0];
h q[2];
cx q[3],q[4];
cx q[2],q[0];
tdg q[4];
h q[2];
h q[4];
cx q[4],q[2];
h q[2];
h q[4];
t q[4];
cx q[3],q[4];
tdg q[4];
h q[2];
h q[4];
cx q[4],q[2];
h q[2];
t q[3];
h q[4];
h q[2];
h q[3];
t q[4];
cx q[3],q[2];
h q[4];
h q[2];
h q[3];
t q[2];
tdg q[3];
h q[2];
h q[3];
cx q[3],q[2];
h q[2];
h q[3];
cx q[2],q[0];
h q[0];
h q[2];
cx q[2],q[0];
h q[0];
h q[2];
cx q[2],q[0];
h q[0];
cx q[1],q[0];
tdg q[0];
cx q[2],q[0];
t q[0];
cx q[1],q[0];
tdg q[0];
t q[1];
cx q[2],q[0];
t q[0];
cx q[2],q[1];
h q[0];
tdg q[1];
t q[2];
cx q[2],q[1];
x q[3];
x q[2];
h q[3];
h q[1];
h q[2];
x q[3];
x q[1];
x q[2];
h q[3];
cx q[3],q[2];
h q[2];
h q[3];
cx q[3],q[2];
h q[2];
h q[3];
cx q[3],q[2];
cx q[2],q[0];
h q[0];
h q[2];
cx q[2],q[0];
h q[0];
h q[2];
cx q[2],q[0];
cx q[3],q[2];
h q[2];
h q[3];
h q[0];
cx q[3],q[2];
cx q[1],q[0];
h q[2];
h q[3];
tdg q[0];
cx q[3],q[2];
cx q[2],q[0];
t q[0];
cx q[1],q[0];
tdg q[0];
cx q[2],q[0];
```

```
t q[0];
t q[1];
h q[0];
cx q[2],q[1];
tdg q[1];
t q[2];
h q[0];
cx q[2],q[1];
x q[0];
x q[1];
x q[2];
h q[0];
h q[1];
h q[2];
measure q[0] -> c[2];
measure q[1] -> c[1];
measure q[2] -> c[0];
```

.1.4.2 QASM Code for Optimized 3 Qubit Grover's Algorithm in IBM Simulator

```
include "qelib1.inc";
qreg q[5];
creg c[5];
x q[0];
h q[1];
h q[3];
h q[4];
h q[0];
h q[2];
cx q[4],q[2];
tdg q[2];
cx q[3],q[2];
t q[2];
cx q[4],q[2];
tdg q[2];
cx q[3],q[2];
t q[2];
tdg q[4];
h q[2];
cx q[3],q[4];
h q[0];
tdg q[4];
cx q[1],q[0];
cx q[3],q[4];
tdg q[0];
t q[3];
s q[4];
cx q[2],q[0];
t q[0];
cx q[1],q[0];
tdg q[0];
cx q[2],q[0];
t q[0];
tdg q[1];
h q[0];
cx q[2],q[1];
tdg q[1];
cx q[2],q[1];
s q[1];
t q[2];
h q[2];
cx q[4],q[2];
tdg q[2];
cx q[3],q[2];
t q[2];
cx q[4],q[2];
tdg q[2];
cx q[3],q[2];
t q[2];
tdg q[4];
h q[2];
cx q[3],q[4];
tdg q[4];
cx q[3],q[4];
t q[3];
s q[4];
cx q[2],q[1];
h q[1];
h q[2];
cx q[2],q[1];
h q[1];
h q[2];
cx q[2],q[1];
h q[2];
h q[3];
h q[4];
x q[2];
x q[3];
x q[4];
h q[2];
h q[2];
cx q[4],q[2];
tdg q[2];
cx q[3],q[2];
t q[2];
cx q[4],q[2];
tdg q[2];
cx q[3],q[2];
t q[2];
tdg q[4];
h q[2];
cx q[3],q[4];
tdg q[4];
cx q[3],q[4];
h q[2];
t q[3];
s q[4];
x q[2];
x q[3];
x q[4];
h q[2];
h q[3];
h q[4];
measure q[2] -> c[2];
measure q[3] -> c[3];
measure q[4] -> c[4];
```

.1.5 Code for implementing 16 qubit Grover's search in Microsoft's LIQ$Ui|\rangle$ simulator

.1.5.1 Oracle function:

```
let oracle (qs:Qubits) =
        let h i      = H [qs.[i]]
        let m i      = M [qs.[i]]
        let x i      = X [qs.[i]]
        let ccx i j k = CCNOT[qs.[i];qs.[j];qs.[k]]
        let cx i j    =CNOT[qs.[i];qs.[j]]

        ccx 13 14 17
        ccx 11 12 13
        ccx 9 10 11
        ccx 7 8 9
        ccx 5 6 7
        ccx 3 4 5
        ccx 0 2 3
        ccx 3 4 5
        ccx 5 6 7
        ccx 7 8 9
        ccx 9 10 11
        ccx 11 12 13
        ccx 13 14 17
        ccx 11 12 13
        ccx 9 10 11
        ccx 7 8 9
        ccx 5 6 7
        ccx 3 4 5
        ccx 0 2 3
        ccx 3 4 5
        ccx 5 6 7
        ccx 7 8 9
        ccx 9 10 11
        ccx 11 12 13
        ccx 17 18 16
        ccx 14 15 18
        ccx 12 13 14
        ccx 10 11 12
        ccx 8 9 10
        ccx 6 7 8
        ccx 4 5 6
        ccx 1 3 4
        ccx 4 5 6
        ccx 6 7 8
        ccx 8 9 10
        ccx 10 11 12
        ccx 12 13 14
        ccx 14 15 18
        ccx 18 17 16
        ccx 14 15 18
        ccx 12 13 14
        ccx 10 11 12
        ccx 8 9 10
        ccx 6 7 8
        ccx 4 5 6
        ccx 1 3 4
        ccx 4 5 6
        ccx 6 7 8
        ccx 8 9 10
        ccx 10 11 12
        ccx 12 13 14
        ccx 14 15 18
        ccx 13 14 17
        ccx 11 12 13
        ccx 9 10 11
        ccx 7 8 9
        ccx 5 6 7
        ccx 3 4 5
        ccx 0 2 3
        ccx 3 4 5
        ccx 5 6 7
        ccx 7 8 9
        ccx 9 10 11
        ccx 11 12 13
        ccx 13 14 17
        ccx 11 12 13
        ccx 9 10 11
        ccx 7 8 9
        ccx 5 6 7
        ccx 3 4 5
        ccx 0 2 3
        ccx 3 4 5
        ccx 5 6 7
        ccx 7 8 9
        ccx 9 10 11
        ccx 11 12 13
        ccx 17 18 16
        ccx 14 15 18
        ccx 12 13 14
        ccx 10 11 12
        ccx 8 9 10
        ccx 6 7 8
        ccx 4 5 6
        ccx 1 3 4
        ccx 4 5 6
        ccx 6 7 8
        ccx 8 9 10
        ccx 10 11 12
```

```
        ccx 12 13 14
        ccx 14 15 18
        ccx 18 17 16
        ccx 14 15 18
        ccx 12 13 14
        ccx 10 11 12
        ccx 8 9 10
        ccx 6 7 8
        ccx 4 5 6
        ccx 1 3 4
        ccx 4 5 6
        ccx 6 7 8
        ccx 8 9 10
        ccx 10 11 12
        ccx 12 13 14
        ccx 14 15 18
```

.1.5.2 Diffusion function:

```
let diffusion (qs:Qubits) =
        let h i      = H [qs.[i]]
        let m i      = M [qs.[i]]
        let x i      = X [qs.[i]]
        let ccx i j k = CCNOT[qs.[i];qs.[j];qs.[k]]
        let cx i j    = CNOT[qs.[i];qs.[j]]
        for i in 0..15 do
            h i
        for i in 0..15 do
            x i
        h 15

        ccx 13 14 17
        ccx 11 12 13
        ccx 9 10 11
        ccx 7 8 9
        ccx 5 6 7
        ccx 3 4 5
        ccx 0 2 3
        ccx 3 4 5
        ccx 5 6 7
        ccx 7 8 9
        ccx 9 10 11
        ccx 11 12 13
        ccx 13 14 17
        ccx 11 12 13
        ccx 9 10 11
        ccx 7 8 9
        ccx 5 6 7
        ccx 3 4 5
ccx 0 2 3
ccx 3 4 5
ccx 5 6 7
ccx 7 8 9
ccx 9 10 11
ccx 11 12 13
ccx 14 17 15
ccx 12 13 14
ccx 10 11 12
ccx 8 9 10
ccx 6 7 8
ccx 4 5 6
ccx 1 3 4
ccx  4 5 6
ccx 6 7 8
ccx 8 9 10
ccx 10 11 12
ccx 12 13 14
ccx 14 17 15
ccx 12 13 14
ccx 10 11 12
ccx 8 9 10
ccx 6 7 8
ccx 4 5 6
ccx 1 3 4
ccx  4 5 6
ccx 6 7 8
ccx 8 9 10
ccx 10 11 12
ccx 12 13 14
ccx 13 14 17
ccx 11 12 13
ccx 9 10 11
ccx 7 8 9
ccx 5 6 7
ccx 3 4 5
ccx 0 2 3
ccx 3 4 5
ccx 5 6 7
ccx 7 8 9
ccx 9 10 11
ccx 11 12 13
ccx 13 14 17
ccx 11 12 13
ccx 9 10 11
ccx 7 8 9
ccx 5 6 7
ccx 3 4 5
ccx 0 2 3
ccx 3 4 5
ccx 5 6 7
ccx 7 8 9
ccx 9 10 11
ccx 11 12 13
ccx 14 17 15
ccx 12 13 14
ccx 10 11 12
ccx 8 9 10
ccx 6 7 8
ccx 4 5 6
ccx 1 3 4
ccx  4 5 6
ccx 6 7 8
ccx 8 9 10
ccx 10 11 12
ccx 12 13 14
ccx 14 17 15
ccx 12 13 14
ccx 10 11 12
ccx 8 9 10
ccx 6 7 8
ccx 4 5 6
ccx 1 3 4
ccx  4 5 6
ccx 6 7 8
ccx 8 9 10
ccx 10 11 12
ccx 12 13 14


h 15
for i in 0..15 do
    x i
for i in 0..15 do
    h i
```

.1.5.3 Function to optimize and compile the circuit:

```
[<LQD>]
let __GroverSearch() =
     let stats      = Array.create 16 0
     let k          = Ket(19)
     let circ1      = Circuit.Compile oracle k.Qubits
     let optCirc1   = circ1.GrowGates(k)
     show "Test1:"
     optCirc1.Dump()
     optCirc1.RenderHT("Test1")
     let circ2      = Circuit.Compile diffusion k.Qubits
     let optCirc2   = circ2.GrowGates(k)

     show "Test2:"
     optCirc2.Dump()
     optCirc2.RenderHT("Test2")
     let qs = k.Reset(19)
     let h i     = H [qs.[i]]
     let m i     = M [qs.[i]]
     let x i     = X [qs.[i]]
     for i in 0..15 do
            h  i
     x 16
     h 16
     for i in 0..200 do
        optCirc1.Run qs
        optCirc2.Run qs
     for i in 0..15 do
        m i
     for i in 0..15 do
        stats.[i] <- qs.[i].Bit.v
     for i in 0..15 do
        show "val[%d]=%d" i stats.[i]
```

.1.6 Code for implementing 16 qubits Grover's search in Microsoft's Quantum Development Kit (QDK)

.1.6.1 Oracle operation:

```
operation Oracle (q:Qubit[]) : ()
    {
        body
        {
            H(q[16]);
            CCNOT(q[13],q[14],q[17]);
            CCNOT(q[11],q[12],q[13]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[0],q[2],q[3]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[13],q[14],q[17]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[0],q[2],q[3]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[17],q[18],q[16]);
            CCNOT( q[14],q[15],q[18]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[8],q[9],q[10]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[1],q[3],q[4]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[8],q[9],q[10]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[14],q[15],q[18]);
            CCNOT( q[18],q[17],q[16]);
            CCNOT( q[14],q[15],q[18]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[8],q[9],q[10]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[1],q[3],q[4]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[9],q[8],q[10]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[14],q[15],q[18]);
            CCNOT(q[13],q[14],q[17]);
            CCNOT(q[11],q[12],q[13]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[0],q[2],q[3]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[13],q[14],q[17]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[0],q[2],q[3]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[17],q[18],q[16]);
            CCNOT( q[14],q[15],q[18]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[8],q[9],q[10]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[1],q[3],q[4]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[8],q[9],q[10]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[14],q[15],q[18]);
```

```
            CCNOT( q[18],q[17],q[16]);
            CCNOT( q[14],q[15],q[18]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[8],q[9],q[10]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[1],q[3],q[4]);
            CCNOT( q[4],q[5],q[6]);
            CCNOT( q[6],q[7],q[8]);
            CCNOT( q[9],q[8],q[10]);
            CCNOT( q[10],q[11],q[12]);
            CCNOT( q[12],q[13],q[14]);
            CCNOT( q[14],q[15],q[18]);
            H(q[16]);
        }
    }
```

.1.6.2 Diffusion operation:

```
operation Diffusion (q:Qubit[]) : ()
    {
        body
        {
            for(i in 0..15)
            {
                H(q[i]);
            }
            for(i in 0..15)
            {
                X(q[i]);
            }

            H(q[15]);
            CCNOT(q[13],q[14],q[17]);
            CCNOT(q[11],q[12],q[13]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[0],q[2],q[3]);
            CCNOT( q[3],q[4],q[5]);
            CCNOT( q[5],q[6],q[7]);
            CCNOT( q[7],q[8],q[9]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[13],q[14],q[17]);
            CCNOT( q[11],q[12],q[13]);
            CCNOT( q[9],q[10],q[11]);
            CCNOT( q[7],q[8],q[9]);
CCNOT( q[5],q[6],q[7]);
CCNOT( q[3],q[4],q[5]);
CCNOT( q[0],q[2],q[3]);
CCNOT( q[3],q[4],q[5]);
CCNOT( q[5],q[6],q[7]);
CCNOT( q[7],q[8],q[9]);
CCNOT( q[9],q[10],q[11]);
CCNOT( q[11],q[12],q[13]);
CCNOT(q[14],q[17],q[15]);
CCNOT(q[12],q[13],q[14]);
CCNOT(q[10],q[11],q[12]);
CCNOT(q[8],q[9],q[10]);
CCNOT(q[6],q[7],q[8]);
CCNOT(q[4],q[5],q[6]);
CCNOT(q[1],q[3],q[4]);
CCNOT(q[4],q[5],q[6]);
CCNOT(q[6],q[7],q[8]);
CCNOT(q[8],q[9],q[10]);
CCNOT(q[10],q[11],q[12]);
CCNOT(q[12],q[13],q[14]);
CCNOT(q[14],q[17],q[15]);
CCNOT(q[12],q[13],q[14]);
CCNOT(q[10],q[11],q[12]);
CCNOT(q[8],q[9],q[10]);
CCNOT(q[6],q[7],q[8]);
CCNOT(q[4],q[5],q[6]);
CCNOT(q[1],q[3],q[4]);
CCNOT(q[4],q[5],q[6]);
CCNOT(q[6],q[7],q[8]);
CCNOT(q[8],q[9],q[10]);
CCNOT(q[10],q[11],q[12]);
CCNOT(q[12],q[13],q[14]);
CCNOT(q[13],q[14],q[17]);
CCNOT(q[11],q[12],q[13]);
CCNOT( q[9],q[10],q[11]);
CCNOT( q[7],q[8],q[9]);
CCNOT( q[5],q[6],q[7]);
CCNOT( q[3],q[4],q[5]);
CCNOT( q[0],q[2],q[3]);
CCNOT( q[3],q[4],q[5]);
CCNOT( q[5],q[6],q[7]);
CCNOT( q[7],q[8],q[9]);
CCNOT( q[9],q[10],q[11]);
CCNOT( q[11],q[12],q[13]);
CCNOT( q[13],q[14],q[17]);
CCNOT( q[11],q[12],q[13]);
CCNOT( q[9],q[10],q[11]);
CCNOT( q[7],q[8],q[9]);
CCNOT( q[5],q[6],q[7]);
CCNOT( q[3],q[4],q[5]);
```

```
        CCNOT( q[0],q[2],q[3]);
        CCNOT( q[3],q[4],q[5]);
        CCNOT( q[5],q[6],q[7]);
        CCNOT( q[7],q[8],q[9]);
        CCNOT( q[9],q[10],q[11]);
        CCNOT( q[11],q[12],q[13]);
        CCNOT( q[14],q[17],q[15]);
        CCNOT( q[12],q[13],q[14]);
        CCNOT( q[10],q[11],q[12]);
        CCNOT( q[8],q[9],q[10]);
        CCNOT( q[6],q[7],q[8]);
        CCNOT( q[4],q[5],q[6]);
        CCNOT( q[1],q[3],q[4]);
        CCNOT( q[4],q[5],q[6]);
        CCNOT( q[6],q[7],q[8]);
        CCNOT( q[8],q[9],q[10]);
        CCNOT( q[10],q[11],q[12]);
        CCNOT( q[12],q[13],q[14]);
        CCNOT( q[14],q[17],q[15]);
        CCNOT( q[12],q[13],q[14]);
        CCNOT( q[10],q[11],q[12]);
        CCNOT( q[8],q[9],q[10]);
        CCNOT( q[6],q[7],q[8]);
        CCNOT( q[4],q[5],q[6]);
        CCNOT( q[1],q[3],q[4]);
        CCNOT( q[4],q[5],q[6]);
        CCNOT( q[6],q[7],q[8]);
        CCNOT( q[8],q[9],q[10]);
        CCNOT( q[10],q[11],q[12]);
        CCNOT( q[12],q[13],q[14]);

        H(q[15]);

        for(i in 0..15)
        {
            X(q[i]);
        }

        for(i in 0..15)
        {
            H(q[i]);
        }
    }
}
```

.1.6.3 Set operation:

```
    operation Set (desired: Result, q1 :Qubit) : ()
        {
            body
            {
                let current=M(q1);
                if(desired!=current)
                {
                    X(q1);
                }
            }
        }
```

.1.6.4 Search operation:

```
    operation Search () : (Result[])
        {
            body
            {
                mutable resultSet=new Result[16];
                using(q=Qubit[19])
                {
                    let databaseRegister=q[0..15];
                    for (i in 0..16)
                    {
                        H(q[i]);
                    }

                    for(i in 0..200){
                        Oracle(q);
                        Diffusion(q);
                    }
                    set resultSet=MultiM(databaseRegister);

                    for (i in 0..18)
                    {
                        Set(Zero,q[i]);
                    }
                }

                return (resultSet);
            }
        }

    }
```

.1.6.5 C# Driver class:

```
class Driver
   {

       static void Main(string[] args)
       {
           int num = 5;
           int count = 0,flag = 0;
            using (var sim = new QuantumSimulator())
           {
               for(int i=0;i<num;i++){
                   flag = 0;
                   var res = Search.Run(sim).Result;
                   var register = res;
                   System.Console.WriteLine($"{register.ToString()}\n
                      ");
                   foreach(Result r in register){
                       if(r.Equals(Result.Zero)){
                           flag = 1;
                           break;
                       }
                   }
                   if(flag == 0){
                       count++;
                   }
               }
           }
           System.Console.WriteLine($"Probability = {(double)count/
              num}");
           System.Console.WriteLine("Press any key to continue..");
           System.Console.ReadKey();
       }
   }
```