Project Dissertation: Practical Quantum Computing

January 2017

Lennart Hirsch
ID: 200800033

Abstract

Quantum computers have been hailed as the Holy Grail of computing since the conception of the idea of a quantum computer in the early 1980s. They are thought to be able to perform many computations at blindingly fast speeds, outperforming classical computers by several orders of magnitude in problems such as large integer factorisation, Fourier Transformations and optimisation problems, making them formidable machines not to be taken lightly: a properly functioning universal quantum computer could break conventional encryption techniques in seconds compared to the decades it would take even the most powerful classical computer. This project explores the potential strengths and advantages of quantum computing, as well as the problems and obstacles that must be overcome in order to construct a practical quantum computer. One such problem is the absence of higher languages in quantum computing. This problem may be solved by creating a system that can communicate with a quantum computer directly; a quantum version of MS-DOS, opening the door to higher level quantum development.

Contents

1 Introduction

Since the dawn of modern computing, classical computers have gotten smaller and more powerful, following Moore's Law almost religiously: the number of transistors we can fit onto a given surface area has been roughly doubling every two years since the conception of Moore's Law in 1965. The number of transistors we can fit in a given circuit correlates directly with the amount of computing power we can achieve. Thus, computational power has been steadily increasing since the creation of microprocessors. It is thought, however, that Moore's Law will soon fail: nowadays classical components are often manufactured at the nanoscale and this introduces new challenges: soon we will be building components at the atomic level, where quantum phenomena have radical affects on systems. This introduces the idea of a size limit to classical computing, and thus predicts the eventual failure of Moore's Law: if we physically cannot reduce the size of transistors any further, how can we increase computational power without quantum mechanical effects wreaking havoc?

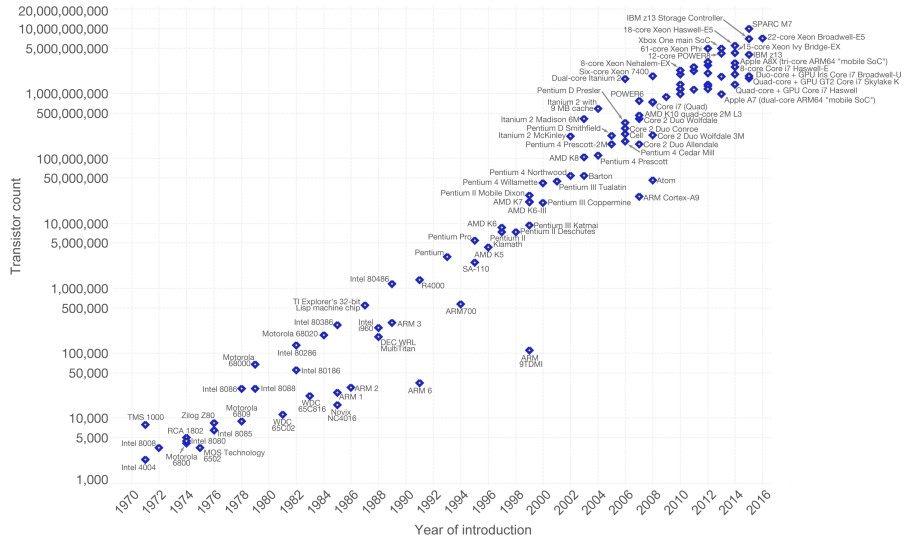

Figure 1: A graph comparing the number of transistors on a given integrated circuit over time. Note how Moore's Law holds strong since its conception: the relationship is almost perfectly linear. It is thought that this relationship could soon come to an end. [1]

Quantum computers are computers that use quantum mechanical phenomena to create, store and modify data instead of classical transistors. Instead of a classical bit that can exist in one of two states (0 or 1), a quantum computer uses qubits. A qubit may exist in either a $|0\rangle$ state or a $|1\rangle$ state, or it may exist in a superposition of the two i.e. it may be in both a $|0\rangle$ and $|1\rangle$ state at the same time. When two or more qubits are working in a system, this collection may also exist in superposition. This vastly increases the number of unique states available for a given number of qubits and thus, theoretically, quantum computers should have vastly more computational power compared to a classical computer with the same number of bits.

It is thought that a practical quantum computer would be able to process large datasets in a fraction of the time it would take an equivalent classical computer. In the age of data this is of utmost importance: the human race is steadily producing more and more data every year. In 2016 alone more than a zettabyte (10^{21} bytes) in internet traffic was sent across the globe. Cisco estimates that this figure will reach 2.3ZB per year by the end of the decade [2]. And this is only internet traffic; many companies and (especially) scientific organisations or facilities produce huge amounts of data. CERN, for example, does not process all the data its detectors output because it is simply too much to process. Instead their systems must sift through the data quickly to find the most promising collision candidates to thoroughly process and analyse. The Square Kilometre Array, a massive array of radio telescope dishes currently under construction, promises to generate an exabyte (10^{18} bytes) of data every day. This is around 100 times the data the LHC produces [3]. If we want any kind of chance of processing all this data, we need a more efficient computing system.

Quantum computers are, however, by no means a perfect replacement for classical computers, neither do they function perfectly: like any other quantum system, a qubit's wave function collapses upon measurement and thus returns to either a $|0\rangle$ state or a $|1\rangle$ state once a measurement is done. Qubits also suffer from an effect known as quantum decoherence. This happens when a quantum system interacts with its surroundings in some way; this can happen if the system is perturbed in some way, for example by a slight change in temperature. For this reason, modern qubits must be cooled to extremely low temperatures ($\sim mK$) to avoid decoherence for as long as possible.

This report seeks to explore the strengths, as well as the potential pitfalls and problems, of quantum computing, as well as proposing possible solutions to some of these problems; especially the problem of higher quantum languages, which may be solved by the implementation of a system that may command and communicate with a quantum computer directly, similar to the way MS-DOS was used as a bridge from classical assembly code to higher level classical computing languages.

2 Historical Background

The theoretical field of Quantum Computing started in the 1980s when scientists began to figure out that, if Moore's Law continued to hold, they would soon reach a size limit in silicon chips. It was understood that at these small scales, their systems would act according to quantum mechanics, and that this could mean the end of Moore's Law. This brought about another question: was it possible to create a new kind of computer based on quantum mechanics; one that took advantage of quantum effects rather than being encumbered by them?

In 1981, Richard Feynman produced an abstract model showing that quantum systems could be used to perform computations [4]. Feynman announced this in a talk at the Conference on Physics of Computation, and was joined at the same event by Paul Benioff, who described Hamiltonian models for quantum mechanical computers, as well as Tommaso Toffoli who produced one of the reversible logic gates, the Toffoli gate (the importance of reversibility of computation will be covered later in this report). These were some of the first attempts to create a model

of a quantum computer.

In the following two decades, quantum computing became a subject, rather than an idea. It was found that quantum entanglement might also be used for quantum computation. The controlled NOT gate (CNOT) was invented in December of the same year. The no-cloning theorem was proven, the first universal quantum computer described, and quantum algorithms began to pop into existence as research and development in quantum computing became more and more interesting. The foundations for a functional quantum computer was quickly being established.

2.1 Shor's Algorithm

In 1994 Peter Shor formulated a quantum algorithm capable of factorising integers in a substantially shorter time than the general number field sieve (the most efficient classical factoring algorithm).

Many modern encryption methods (for example RSA) are based on using large prime integers to generate the public and private keys. Factorising these large primes would take many years on even the most powerful classical computers available to us today; this is why such encryption methods are deemed safe, as it would take a classical computer far too long to solve the prime factorisation problem for it to be a feasible method of breaking the encryption.

Shor's algorithm changed this, as a quantum computer with sufficient qubits running Shor's algorithm would be able to solve this problem in mere seconds, rendering any factorisation-based encryption methods insecure. The problem, however, is that we do not yet have a quantum computer with sufficient qubits, or one that can run for long enough without its state becoming decoherent. For this reason, Shor's algorithm has been a huge motivator in the field of quantum computing, both in terms of theoretical research and practical attempts to construct a universal quantum computer, as well as sparking new fields of research in post-quantum cryptography to find new methods of encryption that would remain secure under the scrutiny of Shor's algorithm.

3 Background

The qubits of a quantum computer may exist in either a $|0\rangle$ state, a $|1\rangle$ state, or a superposition of the two. These states may be represented by 1x2 matrices:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{3.0.1}$$

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{3.0.2}$$

These two states form what is known as the standard basis and are the equivalent of the classical 0 and 1 states.

Similarly, quantum operators or gates are also represented in matrix form. Due to this, figuring out what the resultant state is after applying an operator is done

by multiplying the operator by the state. Take, for example, a bit flip on a $|0\rangle$ state:

$$X \cdot |0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \tag{3.0.3}$$

This process is the same for any operator and any state.

3.1 Quantum Gates & Operators

3.1.1 The Pauli Operators

The Pauli operators are some of the most basic building blocks of quantum computing:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.1.1}$$

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{3.1.2}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{3.1.3}$$

The X-gate is a bit flip; it flips the state of a bit from $|0\rangle$ to $|1\rangle$ and vice versa. Z is a phase flip. It will change the phase of the state (+/-), but will not change the state itself. Y is a combined bit and phase flip.

3.1.2 The Clifford Gates

Another set of basic quantum gates are the Clifford gates:

$$H = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.1.4}$$

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \tag{3.1.5}$$

$$S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix} \tag{3.1.6}$$

The Clifford gates are often used to set up superposition states. The Hadamard (H) gate is one of the most frequently used operators in quantum computing, and may be used to set up the most basic superposition: a superposition state is a combination of $|0\rangle$ and $|1\rangle$. Applying a H gate to $|0\rangle$ results in the $|+\rangle$ state; we can use a Z gate to flip the phase, resulting in the $|-\rangle$ state. These superpositions form a new basis known as the diagonal basis:

$$|+\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} \tag{3.1.7}$$

$$|-\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{3.1.8}$$

The S and $S^\dagger$ gates allow for further superpositions. Once a H gate is applied to a bit, an S gate may be applied to create a $|\circlearrowright\rangle$ state; applying an $S^\dagger$ gate instead will create a $|\circlearrowleft\rangle$ state:

$$|\circlearrowright\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle + i\,|1\rangle) = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ i \end{pmatrix} \tag{3.1.9}$$

$$|\circlearrowleft\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle - i\,|1\rangle) = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 \\ -i \end{pmatrix} \tag{3.1.10}$$

These superpositions form the circular basis.

3.1.3 The CNOT Gate

Another fundamental component is the controlled NOT (CNOT) gate; a CNOT gate is like a classical NOT gate (bit flip) with the ability to control whether or not the gate is triggered/applied: if the control bit is in a $|1\rangle$ state, the target bit will be acted on by the CNOT gate - its state will be flipped. However, if the control is in a $|0\rangle$ state, the CNOT gate will not act on the target, and the target's state will be unaffected. A CNOT is represented by a 4x4 matrix:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.1.11}$$

It is worth noting that a CNOT gate has 2 inputs and 2 outputs, making it a reversible gate. This an important property when attempting to build a universal quantum computer.

The fact that a CNOT allows one qubit to affect the state of another allows qubits to become entangled; the state of the target qubit depends on the state of the control qubit, so their states are entangled.

3.1.4 Non-Clifford Gates

Every gate discussed up until this point (the Clifford group), while reversible and usable in quantum computing, can still be simulated with relative efficiency on a classical computer. Speaking in the most general terms, a quantum computer performs unitary matrix calculations in 2^n dimensions. A quantum computer with the ability to compute any unitary matrix (and thus perform any operation possible on a quantum computer) is known as a universal computer and requires what is known as a universal gate set; this is a finite set of gates that can be used to create or approximate any unitary matrix (this is similar to the concept of functional completeness in classical computing which states that any Boolean function may be created from a functionally complete set of logic gates - such as { AND, NOT }).

As the Clifford group can be simulated using a classical computer, it is not universal, and thus cannot take full advantage of a quantum system. It was found in 1995 by Adriano Barenco that a combination of one- and two-qubit gates can

be used to write any unitary matrix, and that the Clifford group may be made universal by adding almost any non-Clifford gate [5].

While there are several possible choices for these additional non-Clifford gates, the IBM system uses the T and $T^\dagger$ gates:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \tag{3.1.12}$$

$$T^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & e^{-\frac{i\pi}{4}} \end{pmatrix} \tag{3.1.13}$$

The T and $T^\dagger$ gates are examples of phase gates; they add or remove a phase to a state and allow us to reach any point on the Bloch sphere. This can be done by increasing the 'T-depth': the more T gates are used, the more points we can get to. This makes it clearer why these two gates are important for the construction of a universal quantum computer: they allow the user to get to any point on the Bloch sphere, thus theoretically allowing any unitary matrix to be approximated.

3.1.5 Recently Implemented Gates

With the recent update of the IBM QASM Language to QASM 2.0, three additional phase gates have been implemented; denoted by $U1$, $U2$ and $U3$, these gates have one, two and three parameters respectively, all of which may be chosen manually. This provides the ability to create bespoke gates for individual use cases, and allows for more precise control over the quantum states of the five available qubits.

In addition to these three new gates, a 'barrier gate' has been added. This barrier may be stretched across one or more qubits, and prevents transformations across this line, again allowing for more flexibility in algorithm development.

Several more features have been added, such as custom subroutines allowing for custom code to be entered and run. However, these are not available on the real device (yet); they may only be simulated on the IBM Quantum Experience.

3.1.6 The Toffoli Gate

The Toffoli gate is the reversible equivalent of an AND gate; it has two inputs and one output, and only outputs 1 if both inputs are also 1. In quantum terms, if both input qubits are in a $|1\rangle$ state, the output qubit will be acted upon and the qubit will be flipped.

3.2 Reversibility of Quantum Computations

The reversibility of a computation is key to beating the size limit of classical computing: Irreversible computations suffer from information loss: For example, one cannot determine both inputs of a classical AND gate from one output. Hence information has been lost in the use of the AND gate. The same goes for the XOR gate; it has 2 inputs and only 1 output. The Toffoli gate is the reversible implementation of the AND gate which uses three wires, or in the case of quantum computers, three qubits; the CNOT gate is the reversible equivalent of the XOR

gate, and it has 2 inputs and 2 outputs. This is a key part in building a universal quantum computer.

3.3 Measuring superpositions

The standard measurement used in quantum computing measures along the standard basis. This basically means it can tell $|0\rangle$ apart from $|1\rangle$. However, if we measure a superposition state using the standard measurement, we won't get the same result every time; the general qubit state may be expressed in the form $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where the probability of measuring a $|0\rangle$ is $|\alpha|^2$ and the probability of measuring a $|1\rangle$ is $|\beta|^2$ where $|\alpha|^2 + |\beta|^2 = 1$. So for the diagonal basis, each state has a 50% chance. This also works for the circular basis. In fact, there is no measurement that can tell $|0\rangle$, $|1\rangle$, $|+\rangle$, $|-\rangle$, $|\circlearrowright\rangle$ and $|\circlearrowleft\rangle$ apart at the same time; this is a fundamental consequence of the uncertainty principle, and gives rise to the possibility of quantum currency and quantum cryptography.

(Note: this is the case for all quantum states: for $|\psi\rangle = \alpha|0\rangle$ or $|\psi\rangle = \alpha|1\rangle$, $|\alpha|^2 = 1$. For superpositions, α and β will have values less than 1.)

In order to measure in the diagonal and circular bases, the standard measurement must be rotated to the basis of choice. To measure in the diagonal basis, the standard basis (z) is rotated to the diagonal basis (x) using a H gate before measurement. To measure in the circular basis, an $S^\dagger$ gate followed by a H gate is used before measurement. This rotates the standard basis into the circular basis (y).

3.4 Quantum Entanglement

Entanglement is one of those quantum phenomena that seem impossible, but are simply true. Quantum mechanics allows for two or more particles to interact in such a way that their individual quantum states are dependent on the states of the other particles, even if separated by a large distance. This means one particle can instantly affect another particle's state. And it really is instant; in fact, this phenomenon resulted in the formulation of the EPR (Einstein Podolsky Rosen) Paradox as entanglement suggested that information between entangled quantum particles could be transmitted at speeds faster than light. Einstein famously referred to this as "spooky action at a distance".

This is indeed a mind-bending concept, and one can picture it in a classical scenario in the following way: picture a bag with two marbles inside. We know the marbles can both be either red or blue. If we pick one marble from the bag, we can learn about its state i.e. it is red. But this tells us nothing about the other marble: the other marble could still be either red or blue, and we won't know until we take it out of the bag and look. The states of the two marbles are **independent** of each other.

If these two marbles where entangled quantum particles, their states would be **dependent** on each other, and we would know what colour the second marble will be by observing the state of the first i.e. if the first marble was red we would know for certain that the second marble is also red. This seems impossible!

However, experiments have been done to show that quantum entanglement is

indeed possible; quantum systems have been created and observed where particles are inexplicably linked, affecting each other over long distances in times shorter than that needed for light to travel the distance.

Entangled states may also be created using a quantum computer. In fact, entangled states are some of the most useful states in quantum computing.

3.4.1 Bell States

The simplest form of entangled quantum states is a Bell state. Named after John Stewart Bell for his work on quantum mechanics, and more specifically on entanglement, Bell states are states of two entangled qubits. These states were used to prove Bell's Theorem which states that no local hidden variable theory could explain the statistics of these entangled states, and as such their behaviour must be governed by non-classical laws.

The Bell experiment uses states such as the following:

$$\psi_+ = \frac{1}{\sqrt{2}} \cdot (|00\rangle + |11\rangle) \tag{3.4.1}$$

$$\psi_- = \frac{1}{\sqrt{2}} \cdot (|00\rangle - |11\rangle) \tag{3.4.2}$$

As can be seen from the wavefunctions of these states, there are only two possible measurements for a two qubit system: $|00\rangle$ or $|11\rangle$. For two independent qubits, this would not be the case; for any system with n qubits, there are 2^n unique classical states. For this system $n = 2$, so we should have $2^2 = 4$ possible measurement outcomes. However because the qubits are entangled, the state of one qubit can affect the other, and thus they both end up in the same state upon measurement.

A Bell state can be set up on the IBM quantum with ease: first, a superposition is generated on the first qubit using a H gate. The first qubit is then linked with the second using a CNOT gate; the first qubit is used as the control, while the second is the target. Finally, both qubits may be measured:

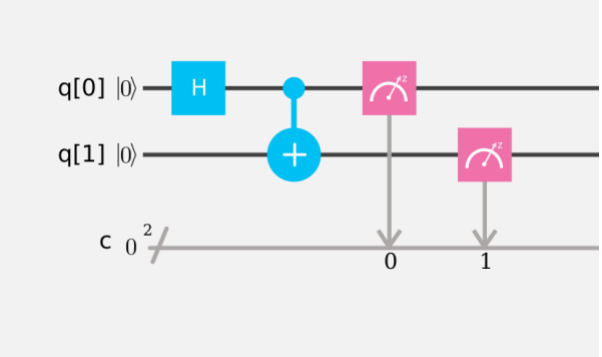

Figure 2: The construction of a Bell state using the IBM quantum composer.

It can be seen in the results that the two qubits always end up in the same state, and that the probability of the $|00\rangle$ and $|11\rangle$ is roughly equal:

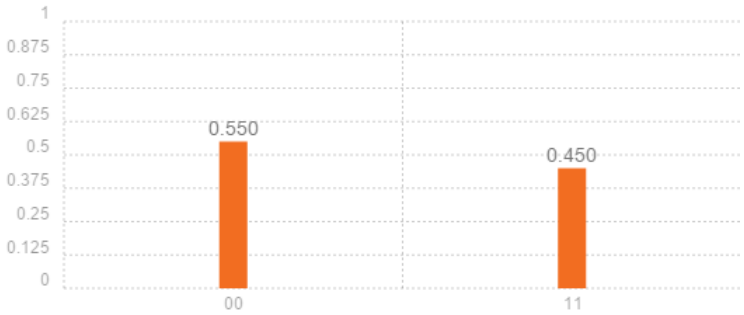

Figure 3: The measurement of a Bell state. It can be seen that only two measurement outcomes are observed, both with roughly 50% probability.

3.4.2 GHZ States

GHZ states generalise Bell states for the 3-qubit case. Named after Greenberger, Horne and Zeilinger who discovered and first studied these states, they are also called "cat states" in reference to Schroedinger's cat. In this case, the wavefunctions would be represented as follows:

$$\psi_+ = \frac{1}{\sqrt{2}} \cdot (|000\rangle + |111\rangle) \tag{3.4.3}$$

$$\psi_- = \frac{1}{\sqrt{2}} \cdot (|000\rangle - |111\rangle) \tag{3.4.4}$$

It can be seen that, again, there are only two possible measurement outcomes: this time $|000\rangle$ or $|111\rangle$. This is an even stranger result: for a 3-qubit system there are $2^3 = 8$ unique classical states, yet in this entangled state, all three qubits always end up in the same state. Again, building a GHZ state can be done with relative ease: in this case the third qubit is the target of two CNOT gates, one from the first qubit and another from the second. This entangles all three qubits together:

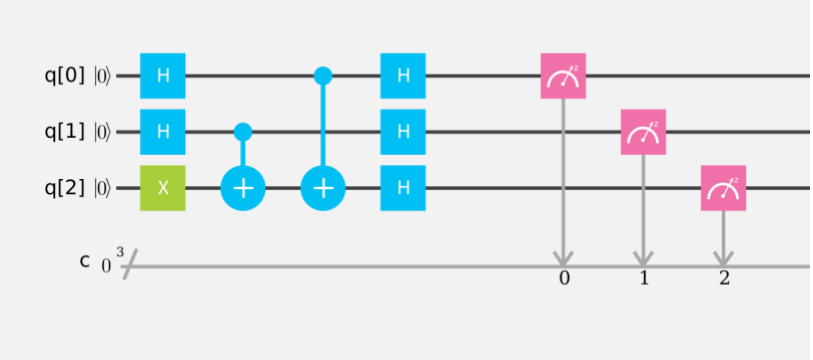

Figure 4: The construction of a GHZ state using the IBM quantum composer.

And again, the results confirm that all three qubits end up in the same state:

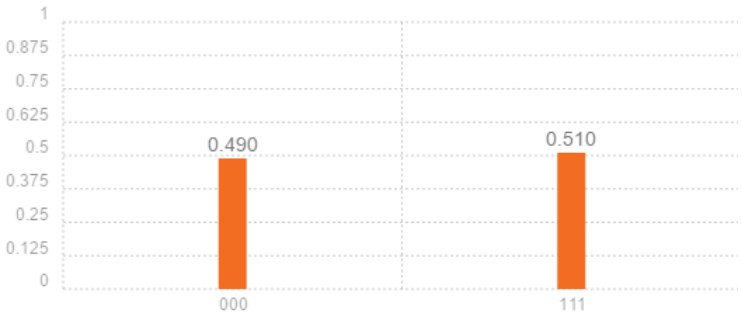

Figure 5: The measurement of a GHZ state.

3.5 Decoherence

While quantum computers use the subtle effects of quantum mechanics to their advantage, they also suffer from the fragility of such systems. If a quantum system is perturbed by a knock or a rise in temperature, it can decohere. This can be seen as a loss of information from the system into the surrounding environment, and it happens because no real world system can be completely isolated from its surroundings; any real quantum computer will be somehow coupled with its environment, and thus the quantum system may 'leak' information and the system decoheres. For example, if a qubit is in a $|0\rangle$ state and experiences a rise in temperature it may get excited and change state, thus losing information from the system. This could result in an error or even the complete failure of a computation.

3.5.1 The Bloch Sphere

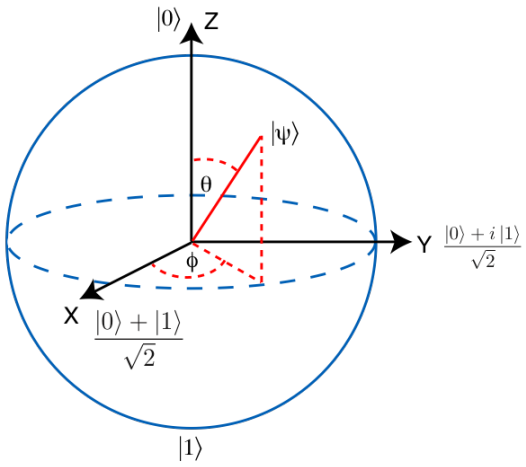

Figure 6: The Bloch Sphere: the standard, diagonal and circular bases can all be seen. [6]

The Bloch sphere is a graphical representation of a qubit; it can show 'where' the state is 'pointing'. For example, if the qubit is in a $|1\rangle$ state, the quantum state points straight down to the bottom of the sphere, as the standard basis points

along the z axis. The rotational basis points along the x axis of the sphere, while the circular basis points along the y axis.

When our system is coherent, the vector inside the Bloch sphere represents what is known as a pure state; the vector's magnitude is 1 and the vector will touch the surface of the Bloch Sphere. However, if our system begins to decohere, our pure quantum states turn into mixed states and the Bloch vector's magnitude is less than 1.

3.5.2 Energy relaxation & Dephasing

Energy relaxation is the process in which a $|1\rangle$ state decays down to a $|0\rangle$. The time taken for this process to occur is represented by T_1.

Dephasing is another process by which a quantum system may decohere, this time by slightly changing the phase (+/-) of a state. Due to this, dephasing only affects superposition states. The time taken for a state to dephase is represented by T_2

Both of these times are of utmost importance, as longer times mean the system is coherent and stable for longer, allowing for more reliable computation as well as longer possible computations. In fact, this is one of the most prominent problems in quantum computing; if we were able to keep a number of qubits from becoming decoherent we would be able to run programs such as Shor's algorithm.

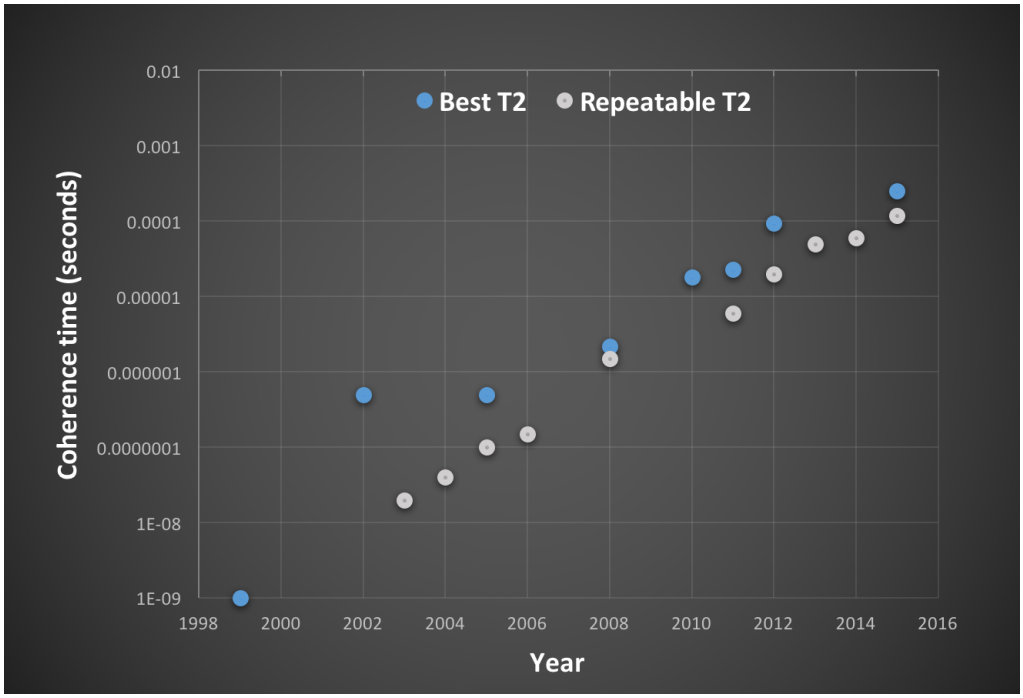

Figure 7: A graph showing the improvement in T_2 over the years. It can be seen that coherence times have been steadily improving. [6]

For these reasons, modern quantum computers must be kept at extremely low temperatures. IBM's qubits are kept at a mere 15mK [6]. This is a task in itself, and is integral to the proper function of the computer.

3.6 Quantum Errors & the Collapse of the Wave Function

As has been explained, quantum systems can be incredibly fragile; in a system of qubits in superposition, if even one qubit is disturbed (for example by interactions with its surroundings - as quantum computers can be built close to the atomic scale, this could even be interactions with surrounding molecules), the entire superposed system may crash. Entangled qubits may detangle, any computation is corrupted and each qubit takes on one of the two classical values: the wave function collapses.

Classical error correction algorithms are heavily based on redundancy measures: for any given computation, several outputs are calculated. These are compared and the most frequent solution is used; the rest is regarded as noisy or corrupted data and is removed. In a quantum system, direct comparison of qubits' states can only be done after measurement; comparing two qubits' states requires taking a measurement of their state, and taking a measurement causes the qubits' wavefunctions to collapse. If this is done in the middle of a computation any superposition states would be destroyed and replaced by classical states, interrupting the computation and rendering the quantum effects of the computation useless.

In fact, the no-cloning theorem (formulated by Wootters, Zurek and Dieks in 1982) [7] shows that it is not possible to create a perfect copy of an unknown quantum state. This means we can't simply copy one qubit's state onto redundant qubits like we would in a classical system; making a copy of the qubit would cause its wavefunction to collapse, destroying any quantum information and disrupting computation.

The challenge, then, is to create error correction algorithms that are able to handle errors during the computation **without** direct measurement of qubit states; the algorithm would have to measure the errors without measuring the data in order to preserve the quantum nature of the system until the computation is complete. The goal is to achieve fault-tolerant quantum computation, meaning the quantum computer would be able to handle errors and tolerate some level of noise while still continuing its computation reliably. This would mean the quantum computer has to be able to handle not only random external noise and quantum effects, but also faulty preparation, imperfect quantum gates and incorrect or inaccurate measurements.

One overarching idea used to correct quantum errors is redundant encoding; this expands the size of the qubit's Hilbert space past what is necessary to store a qubit of information, allowing for errors to be mapped to a set of orthogonal subspaces. This allows for quantum errors to be determined without determining anything about the qubit's quantum state; doing so would collapse the wavefunction [8].

Another frequently used technique is the 3-qubit bit-flip code. This encodes a single qubit into three qubits, and is able to correct for a single bit-flip error; to illustrate how, consider a computation running without bit-flip errors. One might expect a result of $|111\rangle$ as each of the three qubits has gone through an identical process. If a bit-flip error occurs, one would observe a $|110\rangle$ state; it is clear that the last qubit has flipped, and should be ignored as an error.

In both these cases it may appear as though we have copied the quantum states of our logical qubits into our redundant qubits, which is forbidden by the

no-cloning theorem. However, we are not copying the quantum state, but rather encoding a single logical qubit into several physical qubits. These physical qubits go through the same computation and thus should give the same result.

However this bit-flip code cannot account for any more than one bit-flip, and also doesn't correct for phase errors, making it a fairly limited technique. Entangled qubit states are used in a somewhat similar fashion: by using GHZ states (or 'cat states') one logical qubit is encoded in three entangled physical qubits. We can say that the logical qubit is encoded **non-locally**; it is defined by the entanglement of the three physical qubits. This helps to protect against noise; assuming the noise is local, it should act differently on qubits in different positions.

One problem with these techniques is that they all use several qubits to encode only one qubit of information; we do not yet have the technical capability to keep such a large group of qubits coherent for long enough to perform quantum computations with them. However, once we are able to do this, these error correction techniques will become practically useful.

3.7 Density of Information

For a system of n classical bits, the system may be in any one of 2^n unique states at any one time. Take, for example, a simple 2-bit system: $2^2 = 4$, so there are 4 unique states the system may be in: 00, 01, 10 and 11. This rises exponentially, i.e. a 3-bit system has $2^3 = 8$ unique states and a byte (4 bits) has $2^4 = 16$ unique states. Another way of seeing this is that for each bit, the number of unique states available doubles.

Qubits have an advantage here: they have the same 2^n states available to them, and will also end up in one of these 2^n states, but the 'intermediate' state of qubits may also exist in superposition both individually and as a group. Additionally qubits can be entangled, creating even more available states. This increases the density of states drastically, giving rise to the theoretical increase in computing power that quantum computers are so popular for. As quantum computers can exist in superpositions, there is also great opportunity for parallel computations. This ability is almost 'inbuilt' in quantum systems; superposition and entanglement is simply the way quantum systems work, so parallel processing is part of the package. This is another feature that makes quantum computers so desirable.

4 The State of the Art

While quantum computing has been advancing since its conception, huge leaps in the field have been made in the last decade: including the production of the first functioning quantum computers. Several large private companies have become heavily involved. Some, like D-Wave are purely based around the field of quantum computing while others, like IBM and Google, have an extensive background of classical computing to build on, as well as large amounts of money which allows for faster and more efficient research.

4.1 D-Wave

In 2007 a private company called D-Wave released a 16-qubit quantum annealer. By the end of the year they had a 28-qubit processor and have improved their system in the years that have followed; The latest D-Wave system, known as the D-Wave 2000Q has up to 2048 qubits. At first, this all sounds like an incredible achievement, and seems to blow the rest of the competition out of the water. Surely the problem is solved if we can already produce quantum computers with over 2000 qubits?

Sadly, the D-Wave is only a quantum annealer, arguably the weakest form of a quantum computer (the most powerful is the universal quantum computer). Quantum Annealers are good at very specific problems, specifically at optimisation problems like finding the lowest energy state in an energy landscape (the lowest valley). For example, a traveller placed somewhere in the landscape will want to move downhill as often as possible, but they may become trapped in a valley that is not the global minimum [9].

A quantum annealer has an advantage here: superposition allows for the traveller starting from many coordinates simultaneously, and thus has more chance of finding the global minimum. Quantum tunnelling allows the traveller to tunnel through small hills, reducing the chance of getting stuck in a local minimum. This can be applied to a handful of different types of optimisation problems such as protein folding, machine learning and network optimisation, and is known to be drastically faster than classical algorithms: Google have reported that the D-Wave can solve annealing problems at around 10^8 times faster than classically simulated annealing.

While the D-Wave may excel in these problems, it can only solve a very limited number of problems - those centreing around optimisation - and as such is not seen as a big competitor in quantum computing.

There is also controversy over how 'quantum' the D-Wave really is: the company itself obviously boasts about its 2000 qubits, and champions itself as the creator of the first functioning quantum computer. While the D-Wave does show evidence of phenomena such as superposition states and tunnelling, to be named a quantum computer its qubits must be capable of entanglement. This cannot be directly measured while the D-Wave is operating and as such, there is controversy about whether or not this is truly quantum.

4.2 The IBM Quantum Experience

IBM is one of the first companies to successfully construct a working quantum computer. They have also allowed for access to the computer from the cloud, allowing the general public to experience and experiment with quantum computing. This also opens many doors for research, as many untested theories may now be tested on a working quantum computer from a laptop or smartphone.

The IBM quantum processor consists of 5 superconducting transmon qubits, designed to have reduced sensitivity to noise and unlike the D-Wave, the IBM quantum computer is designed to be a universal quantum computer. The qubits are made of niobium and aluminium patterned onto a silicon substrate, and are kept at 15mK in a dilution refrigerator to reduce noise or heat to an absolute minimum [6]; these efforts are done to keep the qubits in as stable a state as possible. This reduces decoherence for as long as possible, allowing for more reliable and accurate quantum computations to be performed.

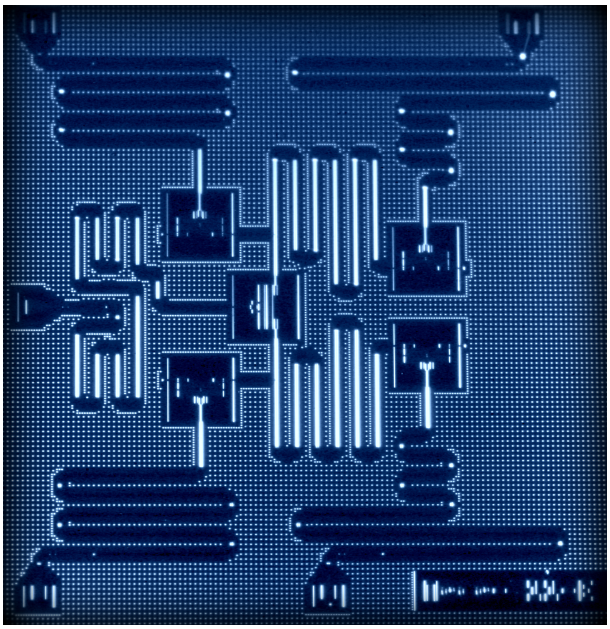

Figure 8: The IBM Quantum Processor. The five qubits can be clearly seen as the dark squares in the centre. [10]

4.2.1 The Quantum Score & Quantum Assembly (QASM)

The IBM quantum computer may be accessed from a web interface which provides the user with everything they need to start building and testing quantum algorithms; tutorials and background information are available, as well as the so-called Quantum Score. The Quantum Score is described to be similar to a musical score: notes and operators are played or applied from left to right. The different lines in the score represent different qubits instead of notes. A quantum algorithm may be created by simply dragging a quantum operator into the score. More of these operators may be placed on one or several qubits, and measurement operator are

placed at the end of an algorithm to determine the output of the algorithm. Then, the score may be run or simulated, and the results are presented. An example of a simple algorithm written using the quantum score is shown below:

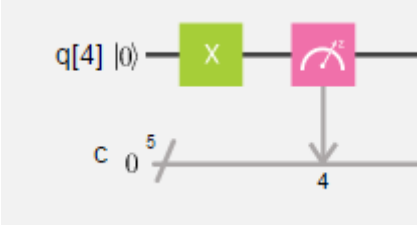

Figure 9: This quantum score simply flips the state of the qubit from $|0\rangle$ to $|1\rangle$, and then takes a measurement of the qubit.

Another way of writing quantum algorithms is by using Quantum Assembly (QASM). This is a simple programming language that represents the quantum operators, and the qubits they operate on, in code form. See, for example, the previous algorithm presented in QASM form:

```
x q[4];
measure q[4] -> c[4];
```

In fact, any algorithm created using the quantum score may be converted to QASM. The quantum score and colourful gates are graphical representations of QASM commands which are sent to the quantum computer. QASM is essentially the binary level of quantum computing.

4.2.2 QASM V.2

A recent update to the QASM language has added some functionality. For instance, it is now possible to create arbitrary rotation operators and include these in algorithms. This adds a huge amount of flexibility and usability to the current system, allowing for highly bespoke algorithms to be run.

4.3 Google

Google have also announced they plan to move into the quantum computing field, although they have been rather secretive about their plans. It is thought that to prove how much better quantum computers are at certain problems would require a quantum computer with thousands of qubits. Google has claimed that they will solve the problem with only 50 qubits [11].

In order to prove this, Google has taken on a notoriously difficult problem: simulating quantum circuits. This is so difficult for classical computers because any slight change to any input will dramatically change the outputs of the system. This means classical computers can't use approximations to help them out. Google have used the Edison supercomputer to simulate quantum circuits with more and more qubits and the difficulty is evident: while a 6x4 grid of qubits requires around 250mB to store, a 6x7 grid (42 qubits) already requires 70TB. 48 qubits would

require more than 2 petabytes, showing that it is near impossible for a classical computer to store and predict the behaviour of quantum circuits.

If Google solved this problem with only 50 qubits, they will have successfully shown 'quantum supremacy' - at least in this category of problems.

Another interesting problem that Google have tackled is molecular simulation: in Summer 2016, Google engineers managed to simulate a Hydrogen (H2) molecule [12]. The problem is a very similar one; classical computers have a lot of trouble working out the tiny, subtle effects found at this small scale; quantum computers, however, seem to excel at this.

5 Problems in Quantum Computing

While quantum computing is theoretically promising there are still numerous obstacles to overcome before the construction of a universal quantum computer is realised.

Some of these problems are technological limits: there's only so much we can reliably cool qubits, and there is still much to learn about ideal construction materials for qubits. The challenge is to create a qubit that allows for us to interact with it in a **controlled** manner. This is difficult however, because if the qubit interacts with its environment in ways we **can't** control, it will decohere very quickly, making it unsuitable for use. On the other hand, if the qubit doesn't interact with its environment much, we will find it very difficult to control its quantum state. The trick is to find a happy medium between these two. Many candidate materials have been tried out; during the years since the conception of quantum computing, scientists have used solids, liquids, and even single atoms as qubits. One such candidate is found as a defect in diamonds [13]: known as the nitrogen vacancy centre, it may sustain a highly entangled quantum state for several milliseconds even at room temperature, and it is easily controlled using lasers or microwaves. Because the defect is contained inside an otherwise almost perfect lattice of atoms, it is very unlikely to interact with any of its neighbours. Such a material would be an ideal candidate; it doesn't interact with its surrounding environment, and works even at relatively high temperatures, while also being easily controllable. This would allow for more stable and reliable qubits to be created, which would in turn allow for longer algorithms and less error-prone computations. Quantum system architectures are still limited in their infancy and until now we only have access to a handful of qubits. But research is ongoing, interest in the quantum computing field is growing as our needs for large data processing grow, and both Google and IBM have made bold technological claims, so perhaps time will solve many of these problems.

Another problem in quantum computing is figuring out what quantum computers are useful for; in which problems do quantum computers excel compared to their classical counterparts? And how does one set up these problems in a way that quantum computers can solve them efficiently? Shor's algorithm has shown that quantum computers can factorise integers substantially faster than the best classical factorising algorithm. This is largely due to the efficiency of the quantum Fourier Transform, another task we know quantum computers to be good at. But these are only two tasks, and are limited in their application. We also think quan-

tum computers should be great at working with large databases due to the fact that they compute using matrices. The real problem comes when attempting to set up computations such that a quantum computer can efficiently perform them. If it takes longer to set up the computation on a quantum computer than it takes a classical computer to perform the computation, it doesn't matter how quickly the quantum computer solves the problem.

A third problem is that of development; while the subject of quantum computing has been growing since its conception, it is still a small and fragmented area of research. IBM's QASM is the first attempt at creating a language for quantum computers. However, QASM is a very low-level language. Even if we had access to quantum computers with hundreds of qubits, developing practical programs in QASM would be incredibly tedious and time consuming.

6 Approaching Higher Languages

This is where higher level languages come in. Higher level languages would allow for more practical development, attracting a wider audience and accelerating the progress of quantum computing.

Classical computing can be used as an example of this: since the assembly stage of classical computing, many steps have been taken to develop high level languages, and with each step developing on classical computers has become more accessible. Writing code in assembly was difficult and complicated. MS-DOS was one of the first attempts to reach for higher classical languages; it essentially allowed the user to execute a variety of functions allowing for file creation, modification and deletion. Each of these functions represented a series of assembly commands. When Microsoft Windows was released, a similar thing happened: now you could double click a file to open it, instead of having to open the file from the MS-DOS (MicroSoft Disk Operating System) command line. While this appears to have been a new feature, double clicking a file to open it would simply trigger the command line code to open the file, which would in turn trigger a series of assembly commands to navigate to and read a certain section of the computer's memory. Since then, many higher level languages have been developed, which; nowadays we are able to create websites through services such as Wix or Squarespace with no knowledge of coding at all. But these commands all break down into simple logical operations; everything can still be broken down to assembly code.

The field of quantum computing is advancing at faster and faster speeds nowadays, and if we want to take full advantage of these advances, new higher level languages must be created; we need a quantum DOS which would provide the first level of higher language: basic boolean and mathematical functions, as well as basic memory operations. Quantum error correction could also be built into this language, providing error preventing and error correcting structures without the need to build these from assembly level every time.

This would allow for faster development, as well as making quantum development more accessible, and allowing users to harness the full power of quantum computing. Once some of the technological problems (such as available qubit numbers and coherence times) have been solved or improved upon, quantum DOS could be modified to include new features, updated to ensure compatibility with

larger qubit numbers, as well as updating error correction algorithms for longer coherence times.

I have created a series of scripts that attempt to reach this first level of higher language development:

6.1 Version 1

Version 1 consists of two Python scripts - one defining all functions and components, and another to compile called functions and produce the output data - and includes all basic functionality of the IBM quantum score. Any of the gates from the quantum composer may be created by calling a function; e.g. to apply a H gate followed by a Z gate on qubit 0, one would use the following code:

$$h(0)$$
$$z(0)$$

This can allow for faster development times than drag and dropping in the quantum composer. The script also provides functions for basic structures such as bit-swaps, Toffoli gates and Bell & GHZ states. This allows for drastic improvements in development time as it is no longer necessary to construct these frequently used tools from the ground up; they can now be created with a single function call.

Once an algorithm has been built, the compiler script is run and outputs a .txt file containing QASM which can be directly imported into the quantum composer and run. Both of these scripts can be found in the appendix.

These features, while basic, can already provide an improvement on development time, as well as making development less repetitive.

This version is designed to be a proof of concept: with a relatively small amount of added functionality, development of quantum algorithms can be significantly streamlined with relative ease.

Version 2 aims to improve on this by adding some higher functionality as opposed to the basic components included in Version 1, in order to take the next small step towards higher language.

6.2 Version 2

In order to create a higher quantum language, higher level functions must be built from their basic building blocks: quantum gates. Version 2 would function as a framework between the quantum computer (which works with quantum gates and operators) and the human user (which prefers to work with readable code) by providing a range of functions such as, for example, mathematical functions (arithmetic operators, square root, trigonometric functions, etc.) as well as larger functions with the possiblity of providing the foundations for quantum algorithms such as Shor's Algorithm or Grover's Search Algorithm. Many of these algorithms must be prepared manually for a given problem; the functions provided by Version 2 would allow for these problems to be set up rapidly rather than having to go through each step of the algorithm manually for each problem.

As well as providing a larger and more complex variety of functions, Version 2 may also be written in a more suitable language: Version 1 was written in Python due to Python's simple syntax, which was ideal for the basic functionality of Version 1. However, Version 1 is a classical script running on a classical computer that eventually outputs a QASM file which may then be input into a quantum computer. Version 2 should reach for higher functionality, and would therefore be designed to work directly on the quantum computer. IBM solves this through the quantum composer: all QASM is built through a graphical interface inside a web app, which is then processed and sent directly to the quantum computer. Version 2 could do a similar thing by using a web API to send the resultant QASM directly to IBM's quantum device. This brings Version 2 closer to a quantum version of MS-DOS. DOS ran on the computer itself, and this is the eventual aim of quantum DOS.

7 Conclusion

Quantum computing is quickly becoming more and more practical; nowadays anyone with access to the internet is able to program and experiment with a functional quantum computer through the IBM Quantum Experience. The advent of large scale quantum computing seems almost inevitable as more and more researchers and companies delve into the complex world of quantum computing. Many problems still remain, but many solutions have been proposed and more are proposed every year.

The idea of Quantum computing has not only spawned its own field of research, but several others too: quantum information theory and quantum cryptography are two examples of this, and with each new field of research, quantum computing has grown, progressed and reached a little further. With the rapid progression of quantum computing, it is time to pay some attention to the field of quantum computational languages. As qubit counts increase and quantum computers become more reliable and more coherent, higher level languages will become a necessity. QASM must evolve and improve with the quantum computers it was designed for in order to be as effective and efficient as possible.

It is clear from the proof-of-concept scripts written in Python that improvements in efficiency and functionality can be achieved with relative simplicity and ease. This concept may also be extended as quantum computers improve. By doing this, a sort of quantum operating system may be created, which would be able to communicate and manage the computations done on quantum computers, making development for quantum computers a practical exercise, rather than one marred by repetition and overcomplication.

References

[1] Our World In Data:
Moore's Law over the years
https://ourworldindata.org/technological-progress/

[2] Cisco VNI Forecast and Methodology, 2015-2020
http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html

[3] What to Do With 1,000,000,000,000,000,000 Bytes of Astronomical Data per Day
https://www.wired.com/2012/04/what-to-do-with-1000000000000000000-bytes-of-ast

[4] Simulating Physics with Computers
https://people.eecs.berkeley.edu/~christos/classics/Feynman.pdf

[5] Elementary gates for quantum computation
https://arxiv.org/pdf/quant-ph/9503016.pdf

[6] The IBM Quantum Experience;
The Weird and Wonderful World of the Qubit
https://quantumexperience.ng.bluemix.net

[7] The No-Cloning Theorem
W. K. Wootters & W. H. Zurek
http://www.physics.umd.edu/studinfo/courses/Phys402/AnlageSpring09/TheNoCloningTheoremWoottersPhysicsTodayFeb2009p76.pdf

[8] Quantum Error Correction for Beginners
S. J Devitt, W. J. Munro & K. Nemoto
https://arxiv.org/pdf/0905.2794.pdf

[9] How D-Wave Quantum Systems Work
http://www.dwavesys.com/quantum-computing

[10] IBM Is Now Letting Anyone Play With Its Quantum Computer
https://www.wired.com/2016/05/ibm-letting-anyone-play-quantum-computer/

[11] Google's plan for Quantum Computer Supremacy
https://www.newscientist.com/article/mg23130894-000-revealed-googles-plan-for-

[12] Googles Quantum Computer Just Accurately Simulated a Molecule for the First Time
http://www.sciencealert.com/google-s-quantum-computer-is-helping-us-understand

[13] A Qubit Candidate Shines Brighter
https://publishing.aip.org/publishing/journal-highlights/qubit-candidate-shines-brighter

8 Appendix

8.1 QASM Definitions Code

```
#Code variable definition
def original():
        return(0)

original.code = ['include "qelib1.inc";','qreg q[5];','creg c[5];']

# Basic operators
def x(bit):
        original.code += ['x q[' + str(bit) + '];']
        return(original.code)

def y(bit):
        original.code += ['y q[' + str(bit) + '];']
        return(original.code)

def z(bit):
        original.code += ['z q[' + str(bit) + '];']
        return(original.code)

def h(bit):
        original.code += ['h q[' + str(bit) + '];']
        return(original.code)

def s(bit):
        original.code += ['s q[' + str(bit) + '];']
        return(original.code)

def sdg(bit):
        original.code += ['sdg q[' + str(bit) + '];']
        return(original.code)

def t(bit):
        original.code += ['t q[' + str(bit) + '];']
        return(original.code)

def tdg(bit):
        original.code += ['tdg q[' + str(bit) + '];']
        return(original.code)

def id(bit):
        original.code += ['id q[' + str(bit) + '];']
        return(original.code)
```

```
def cnot(control, target):
        original.code += ['cx q[' + str(control) + '], q[' + str(target)
        return(original.code)

def measure(bit):
        original.code += ['measure q[' + str(bit) + '] -> c[' + str(bit)
        return(original.code)

def u1(bit, phase):
        original.code += ['u1(' + str(phase) + ') q[' + str(bit) + '];']
        return(original.code)

def u2(bit, phase1, phase2):
        original.code += ['u1(' + str(phase1) + ',' + str(phase2) + ') q
        return(original.code)

def u3(bit, phase1, phase2, phase3):
        original.code += ['u1(' + str(phase1) + ',' + str(phase2) + ',' +
        return(original.code)

def barrier(*arg):
        code = 'barrier '
        for item in sorted(arg, key=int):
                code += 'q[' + str(item) + '],'
        code = code[:-1]
        code += ';'
        original.code += [code]
        return(original.code)

# Basic functions
def controlledPauli(bit, gateType):
        if gateType.upper() == 'Y':
                gate = [
                'sdg q[2];',
                'cx q[' + str(bit) + '], q[2];',
                's q[2];']
        elif gateType.upper() == 'Z':
                gate = [
                'h q[2];',
                'cx q[' + str(bit) + '], q[2];',
                'h q[2];']
        else:
                gate = ['cx q[' + str(bit) + '], q[2];']

        original.code += gate
        return(original.code)
```

```
def controlledHadamard(bit):
        gate = [
        'h q[' + str(bit) + '];',
        's q[' + str(bit) + '];',
        'h q[2];',
        'sdg q[2];',
        'cx q[' + str(bit) + '], q[2];',
        'h q[2];',
        't q[2];',
        'cx q[' + str(bit) + '], q[2];',
        't q[2];',
        'h q[2];',
        's q[2];',
        'x q[2];']

        original.code += gate
        return(original.code)

def GHZ(bitA, bitB):
        state = [
        'h q[' + str(bitA) + '];',
        'h q[' + str(bitB) + '];',
        'x q[2];',
        'cx q[' + str(bitA) + '], q[2];',
        'cx q[' + str(bitB) + '], q[2];',
        'h q[' + str(bitA) + '];',
        'h q[' + str(bitB) + '];',
        'h q[2];']

        original.code += state
        return(original.code)

#THIS FUNCTION SHOULD NOT BE CALLED: IT IS ONLY HERE TO BE USED BY bitSw
def simpleSwap(bit):
        swap = [
        'cx q[' + str(bit) + '], q[2];',
        'h q[' + str(bit) + '];',
        'h q[2];',
        'cx q[' + str(bit) + '], q[2];',
        'h q[' + str(bit) + '];',
        'h q[2];',
        'cx q[' + str(bit) + '], q[2];']

        return(swap)

def bitSwap(bitA, bitB):
        if bitA == 2:
```

```
            swap = simpleSwap(bitB)

        elif bitB == 2:
            swap = simpleSwap(bitA)

        else:
            swap = simpleSwap(bitA) + simpleSwap(bitB) + simpleSwap(

        original.code += swap
        return(original.code)

def toffoli(controlA, controlB):
        gate1 = [
        'h q[2];',
        'cx q[' + str(controlB) + '], q[2];',
        'tdg q[2];',
        'cx q[' + str(controlA) + '], q[2];',
        't q[2];',
        'cx q[' + str(controlB) + '], q[2];',
        'tdg q[2];',
        'cx q[' + str(controlA) + '], q[2];',
        't q[' + str(controlB) + '];',
        't q[2];',
        'h q[2];']

        gate2 = ['cx q[' + str(controlA) + '], q[2];',
        't q[' + str(controlA) + '];',
        'tdg q[2];',
        'cx q[' + str(controlA) + '], q[2];']

        gate = gate1 + bitSwap(controlB, 2) + gate2 + bitSwap(controlB,

        original.code += gate
        return(original.code)
```

8.2 QASM Compiler Code

```
from QASM import *

#CALL QUANTUM FUNCTIONS HERE
h(0)
cnot(0,1)
x(2)
barrier(0,1,2)
h(2)
measure(0)
measure(1)
measure(2)

#END OF QUANTUM FUNCTIONS

print('\n'.join(original.code))

with open('QASMout.txt', 'wt') as outputFile:
        outputFile.write('\n'.join(original.code))

print('Done')
```

8.3 Example Output

Below is a sample file created by Version 1 of the script. This QASM can be directly imported to the IBM Quantum Experience. The first three lines define simple properties such as the library being used, the size of the qubit register, as well as the classical bit register (which is used to store measurement results):

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