SCHOOL OF SYSTEMS ENGINEERING

UNIVERSITY OF READING

Project Report

BSc. Computer Science

SQASM: Simple Quantum Assembly

Ryan Watkins

SCHOOL OF SYSTEMS ENGINEERING

UNIVERSITY OF READING

Project Report

SQASM: Simple Quantum Assembly

Author:	Ryan Watkins
Supervisor:	Dr. James Anderson
Submission Date:	22nd April 2016

I confirm that this project report is my own work and I have documented all sources and material used.

Reading, United Kingdom, 22nd April 2016 Ryan Watkins

Acknowledgments

Foremost, I would like to express sincere gratitude to my advisor Dr James Anderson for his continuous support and efforts to push me further in my search for understanding Quantum Computing. The pressure given to research more made my search worthwhile and gratifying.

I would also like to thank my family for all the support and care they have given me over the years.

Abstract

New quantum programming languages need to be derived for when full-scale quantum systems become available. This is a point that seems to be not too far away, given quantum computers of 1000 qubit size [TK15] as of August 2015. SQASM is a simple quantum assembly presented as a language other higher order Quantum Programming Language (QPL) implementations can compile down towards. The language is designed for the ease of understanding and for pragmatic usage close to hardware. SQASM ties into a complimentary quantum simulator with the help of parsing and lexical analysis from classical compilation tools YACC and LEX respectively. The quantum simulator is written in Python. The idealized architectural model chosen is quantum random access memory (QRAM). One may perform local quantum algorithms and arithmetic with compilation down to gate descriptions. Full adherence of quantum mechanics is given to provide true quantum simulation. The simulator contains implementations of a quantum full-adder [Gos98], Quantum Cost efficient quantum multiplier [KG15] and also Deutsch's algorithm [Deu85]. Both the simulator and programming language are seperable components. Wrapper functions within the simulator allow the language to utilize quantum computation.

Contents

1. Introduction

The main theme of this paper is to present a quantum programming language SQASM (Simple Quantum Assembly) and it's quantum simulator counterpart. SQASM makes calls to a quantum simulator via the Python-C API [VD02]. Included are implementations of quantum arithmetic, bell state configurations, simulated quantum random access machine architecture (QRAM), various quantum gates and also Deutsch's algorithm. The quantum machine that is being simulated can be thought of as a coprocessor similar to GPU's today. The language presented is easy to understand but practical in finding or performing quantum algorithms. Figure 1.1 presents SQASM syntax. The exploration of the language is given further in Section 6.0.1.

```
INITIALIZE R 2
U TENSOR H I2
APPLY U R
SELECT S1 R 0 1
MEASURE S1 RES
APPLY CNOT R
MEASURE R RES
```

Figure 1.1.: SQASM Syntax

The quantum circuit computational model is used as opposed to the Quantum Turing Machine [Deu85] model, which uses quantum automata and quantum finite state machines. As a quantum circuit computational model is more pragmatic and easier to reason about, it is elected instead.

Peter Shor's algorithm [Sho99] demonstrated the power of quantum machines. Shor demonstrated a factoring and discrete logarithmic algorithm which could break modern cryptography (RSA) in polynomial time. Naturally, there has been discussion of moving away from such measures. Instead, one can utilize quantum cryptography due to interference effects found in quantum objects. That is to say, one can leverage the fact that measurement upon a quantum state stops the quantum bit from being quantum mechanical.

It's important to realise that finding quantum algorithms is a hard problem and requires different thinking. Typically, quantum algorithms lie within NP intermediate (NPI) complexity. Shor's algorithm lies within BQP (bounded error quantum polynomial time). This category of algorithms have an error probability of $\frac{1}{3}$. It holds relations to the NP problem space found within computer science. NPI problems are likely to hold yield for non-trivial speedups when applying quantum computational models.

The idea of exploring quantum algorithms using quantum programming languages on classical machines has been explored within the field and is seen as a valid way of finding quantum algorithm speedups. The transition from theory to practice often shows unprecedented ways of gaining speedup. Further exploration into non-trivial algorithm implementations are discussed in Section 5.

Quantum computation has various subsections of research. Quantum artificial intelligence has formulated due to the ability to search a large problem space more efficient than a classical machine. In fact, the current wave of thinking is that there will be the capability to stop making approximations in AI algorithms such that one can form a full solution using quantum means with just 100 to 1000 qubits (quantum bits). Further reading in Quantum AI can found in [Ben+15], [Rie+15], [VMR15].

Quantum error correction deals with decoherence between quantum machines and their environment. Error detection lies within information theory. Errors occur during measurement due to decoherence. This is known as the phenonema that collapses quantum probabilistic states into classical ones. An assumption is made in this report that quantum error correction is handled for us, such that the underlying system simulated already contains quantum error correction. Advancements have been made in the field in regards to quantum error correction such that precise and accurate measurement is possible.

Those new to quantum computation or needing a refresher can find such in Section 2. Further analysis is given on what to expect from the implementation in Section 3. Those already familiar with quantum computing may wish to skip to Section 3 or Section 4. Section 4 outlines the approach taken to build a simple quantum programming language (QPL) and quantum simulator.

For an interested reader, the literature can be found and analysed in Section 5. Section 6 outlines the implementation of the simulator and programming language. Various code snippets are given to help show the simplicity of the language. A Cost-Efficient quantum multiplier and quantum full-adder is outlined and explained also. A pragmatic coder may wish to skip to the Appendix to see the full implementation of the quantum simulator.

One hope is that others may take the language and produce compilers which compile down towards SQASM for means of physical usage or that one can learn from it's simplicity to formulate better high-level quantum programming languages.

2. Quantum Computing Prerequisites

2.1. Nomenclature and Notation

2.1.1. Dirac Notation

Dirac notation or 'Ket' notation is the way in which one may represent quantum states. A 'Ket' is the brother to a 'Bra' (hence the name for bracket). A Bra takes the form $\langle 0|$ and a Ket takes the form $|0\rangle$. Both are used to represent a linear composition of quantum states.

The quantum state zero may be represented as $|0\rangle$ and the quantum state one may be represented as: $|1\rangle$. The physics reader may wish to think of this as the down and up state configuration for a given electron spin or rather an excited and ground state respectively.

The notation can be extended to describe further aspects. A given quantum register or quantum computer can be denoted also as being in the configuration as,

$$X = |001\rangle$$

where q_1, q_2 and q_3 are the states $[0, 0, 1]$ respectively

One can also specify operations on quantum states:

$$c\,|a\rangle = |a'\rangle$$

Addition can be shown between two state vectors such that,

$$|a\rangle + |b\rangle = |c\rangle$$

An inner product can also be shown such that,

$$\text{Let X } = \langle a|b\rangle$$

Where a is the column vector: $\begin{pmatrix} \overline{a_1} \\ \overline{a_2} \end{pmatrix}$ and b is the row vector: $\begin{pmatrix} b_1 & b_2 \end{pmatrix}$

Then,

$$X = \overline{a_1}b_1 + \overline{a_2}b_2$$

One can also note that,

$$\langle a|b\rangle = \overline{\langle b|a\rangle}$$

2.1.2. Magnitudes

Magnitudes represent quantum state probabilities. Magnitudes $\in \mathbb{C}$. This means that each magnitude consists of a number that is $a + bi$. For the computational basis of states $[0, 1]$, the two coefficients α and β represent the coefficients for each state $|0\rangle$ and $|1\rangle$ respectively. Magnitudes represent the possibility that upon observation, a given qubit will be a given state.

Let X be a single qubit quantum system denoted as,

$$X = \alpha\,|0\rangle + \beta\,|1\rangle$$

Then,

$$\overline{\alpha}\alpha = P(|0\rangle) = |0|$$

And,

$$\overline{\beta}\beta = P(|1\rangle) = |1|$$

We also obey the law that probabilities **must** add to one.

$$\text{Therefore, } \overline{\alpha}\alpha + \overline{\beta}\beta = 1$$

One can **normalize** a given set of states by taking the inner product of one state with itself. This allows the probabilities to add to 1.

$$\langle a|a\rangle = |a| \tag{2.1}$$

A key issue in the simulation of quantum computers is exponential scaling of amplitudes per qubit. For n qubits, there exists 2^n amplitudes. Thus, we run into a storage issue during classical simulation and as such, it is advised those wishing to use the simulator do not do so with qubit amounts higher than 30. It should also be noted that this is the reason it is not possible to efficiently simulate a quantum computer using classical machines!

2.1.3. Superposition

In classical systems, the states are such that a given bit is either $|0\rangle$ or $|1\rangle$. However, superposition allows a quantum bit (qubit) to be in the state $|0\rangle$, $|1\rangle$ or $|0\rangle + |1\rangle$ which is known as a superposition. This is where quantum systems bear fruit. If one can harness superpositions, one can perform very powerful computation. Both superposition and entanglement give quantum computers more capability than that of classical machines and as such open new ways to perform computation.

A general n-qubit system can be an arbitrary superposition over all 2^n computational *basis* states,

$$\sum_{q_1 q_2 \dots q_n \in \{0,1\}^n} c_{q_1 \dots q_n |q_1 \dots q_n\rangle} = \sum_{i=0}^{2^n-1} c_i \, |i\rangle \tag{2.2}$$

Where c_i represents complex amplitudes and the integer i represents the basis state $q_1 \dots q_n$ in binary form. This means that one can take this as one superposition state and apply operations to it's entirety in one step. This is known as *quantum parallelism*. The computation however, only results in another superposition state and a measurement only will pick a given qubit at random. However, using inteference effects one can cancel states during measurement and work around this to some success. Usage of this can found in Shor's algorithm and numerous other quantum algorithms.

Likewise, we have the issue of probing the system to get a given quantum register into useful states which we will take a look at in Section 2.2.5. One must perform unitary operations to get a quantum computer to behave desirably.

2.1.4. Entanglement

Entanglement can be found in Bell states. Entanglement is a concept such that n qubits can be found in entanglement such that the measurement upon one qubit affects that which it is tied to non-locally. In fact, were it not for the classical communicating means of transferring the information, this would break the speed of light barrier.

The bell states are said to be in maximum entanglement for all possible states. The effect of measurement is that in a non-local and definite way. The four Bell states are formed from the quantum circuit in Figure 2.1 and are found to be,

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}; \tag{2.3}$$

$$|\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}; \tag{2.4}$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}; \tag{2.5}$$

$$|\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}; \tag{2.6}$$

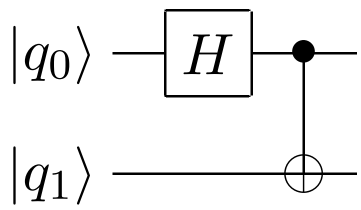


Figure 2.1.: EPR pair formed of *Hadamard* and *CNOT* gates

As mentioned previously, entanglement is a very powerful tool in a quantum computist's arsenal. It should also be noted that it can be extended to n qubits.

2.1.5. Observables

The *observables* of a system are the things that you can measure and get answers for. These are the states that can be extracted from a normal configuration or from a given superposition or entanglement state. The end result is of $[0, 1]$ and as such represents a classical bit after *collapse* from a quantum state.

2.1.6. Hermitian Matrices

Hermitian matrices hold the following two properties:

$$M_{ij} = M_{ji}^{*} \tag{2.7}$$
$$\therefore M_{ii} \in \mathbb{R} \tag{2.8}$$

By definition, this means diagonals must be real numbers. Hermitian matrices are used to measure the quantum observables.

2.1.7. Hermitian Conjugate

Notation is defined as $M^\dagger$

Definition 1. *The Hermitian conjugate is the complex conjugate of the transpose of Hermitian Matrix M.*

$$M^\dagger_{ij} = \overline{M_{ji}}$$

2.1.8. No Cloning Theorem

This theorem states that one can not **copy** quantum states in superposition. In classical computation, one can clone states. The classical example is the cloning of digital information.

The incapability to clone a quantum state is illustrated by the **uncertainty principle**. This is the phenomena that occurs when one tries to observe a quantum state. When one tries to observe a quantum state, the state changes into a non-quantum state, classical state of zero or one. The uncertainty principle was formulated from the observation that one can not clone something that can not be measured precisely. The no cloning theorem is found as,

Definition 1. *There is no valid quantum process that takes as input an unknown quantum state $|\Psi\rangle$ and an ancillary system in a known state, and outputs two copies of $|\Psi\rangle$*

This can also be proven to be broken if we configure a system that has any non-trivial superposition states.

Proof. Consider two orthogonal states $|\Psi\rangle, |\Phi\rangle$ by the definition of cloning:
$|\Psi\rangle \otimes |0\rangle = |\Psi\rangle \otimes |\Psi\rangle$
$|\Phi\rangle \otimes |0\rangle = |\Phi\rangle \otimes |\Phi\rangle$
By linearity, given a state of superposition:
$U[(\langle\alpha|\Psi\rangle + \langle\beta\Phi\rangle) \otimes |0\rangle] = \alpha U(|\Psi\rangle \otimes |0\rangle) + \beta U(|\Phi\rangle \otimes |0\rangle)$
$= \langle\alpha|\Psi\rangle \otimes \Psi + \langle\beta|\Phi\rangle \otimes \Phi$ *(Because we know what U does from above)*
But, of course, it should have produced this:
$(\langle\alpha|\Psi\rangle + \langle\beta|\Phi\rangle) \otimes (\langle\alpha|\Psi\rangle + \langle\beta|\Phi\rangle)$ □

2.1.9. Tensor Products

Quantum states reside within a *Hilbert space*, that is to say they reside in a complex vector space as orthogonal matrices. Hilbert spaces are complex vector spaces of n dimensional size. To apply unitary operations to their state, one must use the dot product of the amplitude column vector with a given quantum gate matrix of sufficient size which leads one to the notion of building quantum gates.

Suppose we wish to combine one gate with another via the Kronecker product so that we can affect our quantum system in some way. One can formulate this new gate combination as outlined below. Note, using the tensor or Kronecker product allows us to increase our gate dimensionality to a level in which we can apply it to a full quantum register. Let,

$$I^2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, H^2 = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.9}$$

$$I^2 \otimes H^2 = \begin{bmatrix} a_{11} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{12} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \\ \\ a_{21} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} & a_{22} \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.10}$$

2.1.10. Unitary Transformations

Unitary operations are how we apply operations to our quantum states to manipulate them in some way, shape or form. A unitary operation follows the rule that the transformation preserves the inner product. As such, a unitary transformation allows one to keep amplitudes that will always add to one. A unitary transformation is how one applies gates to quantum bits and performs quantum computation. Every quantum gate must be a unitary matrix. The unitary aspect also preserves reversibility. By default, every quantum gate operation can be reversed, however it should be noted that measurement cannot be reversed.

2.2. Quantum Concepts

The model with which we model our quantum computer is called the Quantum Circuit model. One can construct quantum circuitry by applying quantum gates to our quantum states. The Quantum Circuit model is formed by the application of quantum gates to quantum bits (qubits). Further explanation can be found in Section 2.2.5. An alternate approach is that of the Quantum Turing Machine. However, intuition allows us to delve easier into the Quantum Circuit model.

2.2.1. Quantum States

Quantum bits hold states just as classical bits do. However, while classical bits deal in binary states, quantum bits can be a continuum between $|0\rangle$ and $|1\rangle$ known as superposition or they can even be in entanglement. The quantum states reside in a complex vector space.

Observation collapses the quantum bit's current state down a classical state in the basis $[0, 1]$. This means one can not observe a quantum bit and rightfully know exactly what the state was, this is known as the **uncertainty principle**.

2.2.2. Quantum Architecture

For the purposes of this report, the architecture is formed of quantum registers, quantum bits and quantum gates. A register here is unbound to containing various superposition states in combination with single definite states. A register is also given a qubit size at compile-time. One can also apply operations to individual quantum bits by selection from registers. It is also clear that there is a co-processor approach given here such that the classical machine 'probes' the quantum machine and gets it to perform computation via the quantum circuitry model.

Error-correction is abstracted away and assumed to be non-problematic for the purpose of this report. The architecture adheres to the QRAM architecture [Kni96]. The nature of physicality and usage of quantum memory access lies outside the scope of the report. For the reader who wishes to study this further, various architectures such as [Mar+11], [Tay+05] and [KMW02] have been proposed.

Quantum software architecture has been researched into by teams at Microsoft Research, hence their Liquid project [Han+16]. However, for the sake of this report, there is no need to worry about the underlying software architecture as there is no general need to build a quantum compiler that adheres to different hardware architectures as SQASM is presented for simplicity.

2.2.3. Reversible Computing

Quantum machines can only be effected by unitary operations and as such are constrained to only perform reversible computation. For both simplicity and also to adhere to this principle our system will take input from a classical machine and give it's output back to this classical machine. Classical machines are not de facto reversible due to the large usage of the AND gate. If you are given an output of $|0\rangle$ for a given bit, one can not determine if the input was $|10\rangle$ or $|01\rangle$ or in-fact $|00\rangle$.

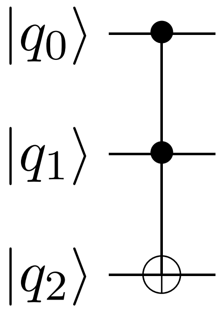
While it is possible to build modern classical machines that perform reversible computation, it is not something widely deployed because they require huge amounts of temporary storage due in turn for the need of garbage ancillary bits. If we take the Church-Turing Machine as an example, such a machine requires that all actions are recorded on an extra tape. Eventually, one must erase the information stored which dissipates energy.

Rolf Landauer argued that logical irreversibility is rather unavoidable in classical machines because to hold state information for reversibility means that the machine would disippate energy for each bit of information it erases or throws away. This is known as **Landauer's principle**.

Charles H. Bennett stated the act of erasing is what dissipates energy and this is irreversible. Therefore one may assume that if a computer is reversible and does not erase information, then there would be less energy wastage [KL70].

2.2.4. Classical Computation

It is well known in computer science that a NAND gate is a *Universal* gate. One can use a NAND gate to form any other gate. As such, the question is asked; can quantum computers perform classical computation in the same way? As quantum computers must perform unitary operations, one must formulate a NAND gate differently. The gate that does such is called the Toffoli gate. Figure 2.2 illustrates the Toffoli gate with a truth table and circuit diagram.



Inputs			Outputs		
a	b	c	a′	b′	c′
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Figure 2.2.: Circuit representation of Toffoli gate with truth table

The Toffoli gate flips the third qubit if the first two are set such that the output for q_3 is $q_3 \oplus q_1 q_2$ where $\oplus$ is modulo addition 2. This should help formalise the idea a Quantum Turing Machine can perform the same computation as a Classical Turing Machine.

2.2.5. Quantum Gates

As mentioned in Section 2.2.3 and 2.1.10, one can compute on a quantum machine using unitary operations. By using the tensor product to formulate any quantum gate, one can then use matrix multiplication to apply a given gate to a set of qubits. This changes the given amplitudes and thus as a result, the quantum system state. That is to say, we can begin to formulate entanglement, superposition or definite states. A quantum gate is a unitary matrix. It can be extended in dimensionality and it's given values can be changed to create a universality of gates.

Consider the application of a quantum gate to a set of quantum bits. Let H be the hadamard gate and ψ be the the state of two qubits,

$$H^2 = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, \psi = \begin{pmatrix} \alpha_1\beta_1 \\ \alpha_2\beta_1 \\ \alpha_1\beta_2 \\ \alpha_2\beta_2 \end{pmatrix} \tag{2.11}$$

$$H \times \psi = \psi' = \begin{pmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \tag{2.12}$$

This can also be thought of in a similar way to classical applications of gates via the usage of circuits. Figure 2.3 presents the hadamard transform explicitly. Each qubit is represented by a wire and each box represents a quantum gate.

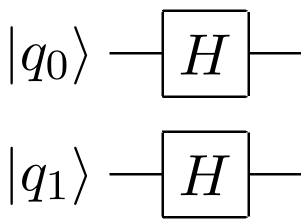


Figure 2.3.: Hadamard transform applied to q_1 and q_2

Also note that *universality* of gates can be found such that combining the Controlled-NOT gate with any single qubit gate forms any multiple qubit logic gate.

2.2.6. Quantum Algorithms

As previously mentioned, Quantum Computing opens up many possibilities. Quantum computers exploit problem areas differently to classical machines and as such, one can formulate quantum algorithms which exploit different ways of computation by using the concepts of superposition and entanglement.

Key quantum algorithms can be split into the sections of Quantum search and Quantum Fourier Transform. The Quantum Fourier Transform is one that helps to break RSA among other things.

We can quantify a way of finding a *good* quantum algorithm as the following:

1. A problem that utilizes superposition and/or entanglement
2. A problem that is easily checkable classically
3. A problem that can be handled in probabilistic fashion
4. Measurement is such that cancellations happen due to exploitable interference effects
5. The time order is non-trivially better than that of classical machines

The progress within the field is rapidly progressing. For instance we can already solve systems of linear equations with an exponential speedup, the progress into simulating quantum systems as given by Feynman in 1982 has had considerable progress such that the molecule Ferredoxin can be simulated in just 1 hour as opposed to the initial classical simulation of $24 * 10^9$.

As previously mentioned, the invention of the Quantum Fourier Transform (QFT) is also huge. So huge in-fact that it's usage means that 1GB data takes 27 ops as opposed to 10^9 ops. The QFT is used as a subroutine in a lot of quantum algorithms due to it's nice speedup. The Quantum Fourier Transform can be used to also form Quantum Phase Estimation which is the key building block within Shor's algorithm.

We encounter Deutsch's Algorithm in the quantum simulator outlined in Section 6. The algorithm is one of the first proposed quantum algorithm's. It is a simple algorithm but still one that is potent in teaching the main thinking process behind quantum algorithms. It's usage of a quantum oracle is one that is also encountered in various other algorithms. A quantum oracle is illustrated later in the analysis of Deutsch's algorithm however for now we shall take it to be a blackbox function that does something useful.

3. Problem Articulation and Technical Specification

A need for a toy quantum programming language is necessary for both pragmatic and learning purposes. SQASM presents a language that is easy to understand and extend such that realised quantum machines could potentially use the language or perhaps a similar form. The project satisfies two major needs, the need for a simple quantum programming language and also a need for a feature-rich simple quantum simulator.

Python is chosen as the language of choice for implementing the simulator due to it's large uptake in the software community, ease to write math and ease to read and digest. Abstractions are made such that one does not need to worry about quantum architecture other than that we take a hybrid approach that combines both classical means and quantum means. The QRAM architecture model is followed and operations are performed according to the quantum circuitry model. The selection of this model is for simplicity and pragmatic concerns also.

A specification is outlined below which helps meet the aforementioned criteria. Key stakeholders include students in Science, Technology, Engineering and Medicine fields (STEM) or those who wish to better understand quantum computing at an introductory level. The project is developed in an environment whereby the field is relatively new and there exists only a select handful of quantum programming languages. One hope is that others may study the language and simulator and use it to help conduct further research into quantum computing.

3.0.1. Technical Specification

The project should satisfy the following technical specification:

1. Perform Binary Arithmetic via constant or quantum register arguments
2. Disable copying of states to obey No Cloning Theorem outlined in Section 2.1.8
3. Ability to store literals or objects in registers

4. Ability to measure quantum states and simulate *collapse* from a quantum state to a classical state
5. Quantum mechanical postulates and laws should be adhered to in full
6. Ability to select a subset of quantum bits from a quantum register
7. Local quantum algorithms should be implementable and fully expressible, including the ability to call quantum oracles
8. One should be able to apply quantum gates to quantum registers
9. Quantum error-correction is assumed to be provided in hardware ad infinitum
10. Execution should be efficient and where possible, optimized
11. The simulator should be easily extensible and follow best-practice object orientation principles
12. Allow one to test simulated quantum machine. Includes ability to see amplitudes after operations
13. Easily readable code such that others can contribute in an open source repository more easier
14. Interfacing between simulator and compiler should be deterministic and reliable
15. SWAP gates can be applied to swap adjacent quantum bits

Testing of these aspects and implementations of these aspects can be found in Section 7 and Section 6 respectively. The general approach to solving the issues is detailed in Section 4.

4. The Solution Approach

To create a simple but functional programming language and quantum simulator, one has to begin to model the interactions within the QRAM architecture. The QRAM architecture allows one to abstract away from having to worry about other memory storage paradigms. Thus, the QRAM architecture can be seen to be just something that exists to hold the quantum registers and quantum bits. As the assumption is made that error-correction is not an issue, one does not have to worry about dealing with error correcting methods in getting quantum information out of the QRAM. As a result, the interactions in the system are simple and easy to understand. As we are simulating a quantum machine, one has to assume that data needs to be stored locally and thus a symbol table is also created for variables.

Figure 4.1 illustrates the key components of the solution as interactions. First the SQASM input file is input to the compiler by means of LEX/YACC[1]. The compiler then makes calls to the Quantum Simulator and the simulator communicates results back to the compiler by means of QRAM. The results can be quantum states, quantum gate descriptions or quantum registers. These results are then stored in a hashtable so that one can recall previously stored variables in further instructions[2]. The result communicated back can be thought of to be an accepting state as the quantum machine will essentially *halt* after doing computation and wait for the next input.

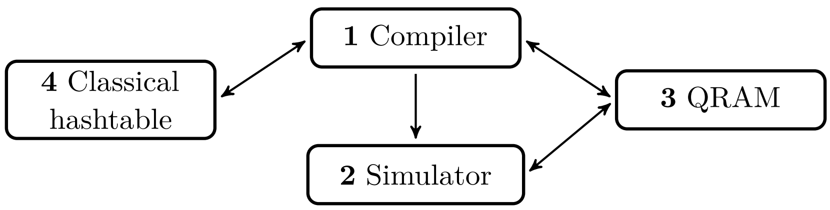


Figure 4.1.: Overview of component interactions during SQASM compilation

[1]Both LEX and YACC provide an easy way to build a quick toy quantum programming language.
[2]This hashtable is hitherto known as a symbol table in compiler literature.

4.1. Designing The Language

4.1.1. Classical Languages

From a classical standpoint, we think of programming languages as being classified into three categories; static-typed abstract languages such as C#, functional languages such as the Lisp[McC60] or Haskell and also logic languages such as OCaml[Ler14] that are useful for pattern matching.

It should be noted that one can take the notion of classical language theory and use it to formulate a *good* quantum programming language. This is a unique way to look at a new computing paradigm. One can look upon all computer science present knowledge and other fields such as information theory, cryptography and physics and right away formulate all the essential means to be ready for when the paradigm is realised. This process is a magnitude quicker than that of the silicon revolution.

Typically languages get compiled down to assembly and then down to machine code by means of an *assembler*. Of course, not all languages are compiled, some are interpreted, such as Python. There also typically exists an intermediary step whereby a intermediary language such as Three Address Code (TAC) is built. This is done by means of the tree data structure with terminals and non terminal formulation. As SQASM is to be interpreted easily, this intermediary interpretation is left for others to implement should they wish to compile down to SQASM. Optimization steps can be made at TAC creation.

Assembly is one of the most fundamental, basic programming languages in existence which is used to program close to hardware. It is a low-level human readable language. Assembly is often used for optimization after compilation downwards as abstract high-level languages often contain bloat.

Existing software architecture can also be utilized from classical languages as the solution contains a hybrid approach. That is to say that we are writing code on a classical machine, not a quantum machine and as such, the existing compiler toolchains and optimization steps can be adapted or utilized.

4.1.2. Quantum Programming Languages

Despite the field being relatively new, innovations in quantum programming languages have been consistent and significant. QPL's such as Quipper[Gre+13], QuIDDPro[VGH11], Scaffold[Abh12] and QuTiP[JNN12] comprise either quantum libraries or entirely new quantum programming languages. Some are more substantial than others. For instance, Quipper is a fully fledged quantum programming language with control statements and

C-like syntax.

As mentioned in Section 5.0.1, Microsoft Research has been leading development on an entire software architecture such that they use F# as a host domain language that gets compiled down to some form of quantum assembly. An eager reader should peruse the literature analysed in Section 5 if they are interested in producing their own quantum programming language or indeed, using a more advanced one. As others have taken care of building high-level QPL's, SQASM positions itself as an easy to understand simple QPL that talks close to hardware. There is a need for clear standards and purpose on the QPL's but as for now, it is exciting to see the speed at which they are being built.

4.1.3. Simulator Solution Approach

In order for our quantum programming language to store variables, one must construct the model as illustrated in Figure 4.1. At a lower-level of abstraction, one must look at API's to communicate from the compiler to the simulator. Thus, the Python-C API is utilized and more notably, the type *PyObject* is used to store variables. To use the Python-C API, one must create an instance of the Python interpreter. Further detail about how this is done can be found in Section 6.

Simulating a quantum collapse during measurement can be done via using a random number generator to collapse a state probabilistically, weighted according to it's amplitudes. This is done inside the quantum simulator.

One must also be able to apply quantum gates to quantum registers. As such, a substantial math library from a classical language must be used. One of the key reasons behind using Python for simulation is easy imports of math libraries that are lightweight. As a result, the library *numpy* is chosen to perform advanced mathematical functions required of the simulator. Efficiency is not sacrificed. Of course, Python does not suffer bad code readability due to it's PEP8 standards.

Other languages would have had a hard time performing operations such as the tensor product or could compromise readability and ease of understanding. For instance, despite that C++ may give a speedup in quantum operations, the lack of code readability would make it a poor fit for the simulator implementation.

In order to construct a simulator one must think about the type of computational model. As mentioned previously, there exists the quantum circuit model or the Quantum Turing Machine model. Both are of equivalence. Alternatively to these two approaches one may also use Lambda calculus similar to Haskell, Scheme and Lisp. Lambda calculus is highly similar to the Quantum Turing Machine. This is looked at in [Ton04]. The quantum circuit model is chosen as this can be used in accordance with quantum gates.

The simulator requires ancillary functions also to help make the code more readable. For instance, the need for an easy to access tensor function is useful as it is referenced a lot during creation of quantum gates. The quantum architecture should be architected also in a way that makes sense and as such, it should be comprised of classes. To address the need for quantum arithmetic, one must look towards papers within the field that illustrate quantum circuitry which perform the required functionality. Two papers as a result were found to give the desired circuitry and are explored further in Section 6. The two research papers provide circuit descriptions for a quantum ripple carry adder and quantum cost efficient multiplier.

Classes and object orientation should also help comprise several aspects of the quantum simulator. For instance, abstractly defining registers and arithmetic instances helps code readability and understanding.

Various quantum gates are given inside the simulator such that one can readily apply them. For instance, using the aforementioned tensor product with swap gates allows one to swap arbitrary qubits.

Due to the complexity of the quantum simulator and topic, one should utilize testing in such a way as to promote determinance. Like-wise the notion of amplitudes should be easily printable and readable at any given stage of compilation. Thus, a variable should be kept for given quantum registers that allows one to view the amplitudes.

To illustrate the ability to perform local algorithms, an easy to understand but powerful quantum algorithm should be selected. Thus, Deutsch's algorithm is elected to be included within the quantum simulator implementation.

The ability to select and manipulate quantum bits is again given by manner of keeping the amplitudes as a variable which can be utilized for a given quantum register instance.

4.1.4. Language Solution Approach

As SQASM is presented as a form of quantum assembly, it should be kept as simple as possible in order to provide easy optimization upon reading. As mentioned previously, in order to keep registers and quantum quantities within variables, one must keep a local hashtable.

The Python-C API must be initialized upon compilation also as compilation is the first time step within Figure 4.1. If SQASM was to be used with a real quantum machine, the need for the Python-C API would be redundant as the simulator would no longer be needed.

Key operations within the language are the application of quantum gates to registers, selection of quantum bits, initialization of quantum registers, measurement, arithmetic and storage of variables. This can all be found within Figure 1.1. SQASM can be thought of to be similar in that to typical specification of QASM, which can be found in [I05].

5. Literature Review

Many items of literature have helped to architect decisions during the project lifecycle and have also served as inspiration during. A breakdown of key pieces of literature are given below.

5.0.1. Liquid, Microsoft Research

Liquid is a quantum software architecture that aims to be the 'Visual Studio of Quantum Computing'. It's goal is similar of that to SQASM in that it is created with the idea of being immediately usable and with the hybrid approach of combining classical machines and quantum machines.

The underlying language is F# and the architecture contains a low level quantum compiler. It allows programmers to program in both classical and quantum code with compilations down to quantum intermediate representations.

Microsoft has managed to make breakthroughs in quantum chemistry by simulating the molecule Forredoxin exponentially quicker than classical means and also quantum theoretical attempts. It's reduction is from 24×10^9 years to one hour and is thus an exponential speedup using **simulation only**. This shows that simulation is helping make breakthroughs in quantum algorithms and will speedup the progress dramatically in realising useful quantum computation.

Microsoft Research are also looking into topological quantum computers. Thus, Microsoft Research is tackling both the hardware and software problems within quantum computation. An interesting aspect of the paper [Han+16] is the specification of quantum libraries. It is well known that libraries improve developer productivity and it is clear that libraries are needed upon realization of quantum programming languages. The paper illustrates the need for the compartmentalization of certain aspects of the quantum programming language and could prove to be interesting in revisions of SQASM.

The paper breaks down the 'quantum library' into components of quantum types, quantum gates, quantum control, quantum arithmetic, quantum math and quantum algorithms. The quantum types are especially interesting also as the reduction of types into bits is highly important at a point where we do not currently have quantum machines with a high amount of bits. It allows one to make optimizations for quantum computers that exist as of now.

The paper also illustrates a pitfall within SQASM; the ability to uncompute or to reverse computation is not implemented explicitly. That is to say that there is no function that can revert back to a specified stage of computation. Another interesting aspect is that of manipulating gate choices at compilation. For instance, the reduction of control statements can be used to further reduce gate usage which speeds up computation. However, SQASM is positioned as a representation followed from an quantum intermediate representation (QIR). As such, the QIR would likely handle this feature.

Another key element is the inclusion of quantum mathematical libraries. Especially that of high-level mathematical functions. This provides quantum programmers with the tools to perform necessary math within their quantum algorithms and to do so efficiently. The usage of the library would also increase code reuse. For instance, the usage of a fused multiple-add allows one to perform multiply and addition. This allows one to use only 6 cycles as opposed to 8 in a normal multiplication followed by addition use-case. The fused multiple-add can found be in SIMD (single instruction, multiple data) approaches.

The paper outlined and SQASM's inclusive quantum simulator both share the usage of quantum algorithm subroutines and as such, share a similar property of both handling quantum algorithm library implementations. Though it should be noted that Microsoft's implementation is far more extensive and extends to Quantum Fourier Transforms, Quantum Phase Estimations, Linear system solvers and various others. For instance, the quantum phase estimation is used in Shor's algorithm [Sho99].

5.0.2. Conventions for quantum pseudocode

The paper outlined in [Kni96] describes the QRAM model in full. The nature of the pseudocode outlined is very similar to that of SQASM syntax. The paper could be used as future inspiration for incorporating some functional language aspects. Other interesting aspects include the ability to initialize classical and quantum registers. This would have been a nice language feature, yet as SQASM is purely for simplicity, it lies outside the scope. It also nicely demonstrates quantum conditionals and the way in which a quantum algorithm may be succinctly specified which is important for the sharing and demonstration of a quantum algorithm.

5.0.3. Quantum Computation and Quantum Information

This is the go-to book for anybody wishing to study quantum computation or quantum information. Nielsen and Chuang give a detailed overview of a whole range of fields in relation to the topic and break down the basics needed. There have been countless times when the book has been used as a reference point and was definitely essential to the construction of the project. Key sections from the book that have been read several times are Quantum Architecture, Linear Algebra and Quantum Models of Computation. Nielsen and Chuang address the topic from various angles including that as a crytographer, computer scientist, mathematician and information scientist.

5.0.4. MIT 8.03 Physics III Material

Various items of video lectures and physical lecture material served as educational resources for the better understanding of quantum mechanics and quantum computing as a whole. The select lecture on Bell Inequality and Deutsch's algorithm helped create the simulators implementation of Deutsch's algorithm. [MG03].

6. Implementation

SQASM is comprised of a quantum simulator and quantum programming language which can perform quantum arithmetic, select arbitrary quantum bits from quantum registers, store quantum objects in variables, perform quantum algorithms, perform quantum gate computation and initialize quantum registers of arbitrary size.

6.0.1. SQASM

For simplicity, the amount of functions is kept to to a minimum. SQASM is a language that only consists of six functions. This improves readability and code reuse. It also helps to see the forest for the trees. The demonstration of quantum computation is clear in any SQASM program. The language is summarised in Figure 6.1.

Operation	**Description**
INITIALIZE $[r, n, pos]$	Initializes a quantum register of n qubit size with definite configuration
$[v_1]$ TENSOR $[g_1, g_2]$	Applies tensor product to unitary matrices
APPLY $[g, r]$	Applies matrix multiplication between quantum state column vector and unitary quantum gate
SELECT $[v, r, n_1, n_2]$	Selects quantum bits from a range inside a quantum register
MEASURE $[r, v]$	Measures the state of a given qubit or register
ADD $[v_1, v_2, r]$	Performs addition or subtraction between constants or variables
PEEK $[r]$	Allows one to peek into a given registers amplitudes for testing purposes
HAD, ID, CNOT, ...	Shorthand references to constant quantum gates Hadamard, Identity and Controlled-NOT respectively

Where r = register, n = number, g = gate and v = variable

Figure 6.1.: SQASM Syntax Table

6.0.2. Compiler Implementation

As SQASM is purely for demonstration purposes and a quantum machine is not easily accessible as of yet, there is a need for a bridge between the compiler and quantum simulator. The Python-C API is a natural fit as YACC (Yet Another Compiler Compiler) takes the form of C code and handles the parsing for SQASM. Likewise, the quantum simulator in SQASM is written in Python.

To bridge the gap, during compilation, a Python interpreter is started by calling `Py_Initialize ();`. After doing so, one can make calls to any Python file and as such, upon different lexical tokens, the compiler will call different quantum simulator wrapper functions. The cycle from compilation to simulator and back to compiler for storage is shown explicitly in Figure 6.3.

In order to pass arguments from compiler to Python interpreter one must formulate a tuple with `PyTuple_SetItem (tup,pos,item)`, where the arguments are the tuple data structure to keep the tuple in, position and item to store respectively. Upon doing so, the data structure has to be sent across to the correct simulator wrapper function, `PyObject_CallFunction (pFunc,tup)`.

The need for a hashtable to store and retrieve different variables during compilation is essential. The general concept of how this works is demonstrated in Figure 6.2. It should be noted that all items within the hashtable are *PyObject* type.

0 $\rightarrow PyObject_0\{QReg_1\}$

1 $\rightarrow PyObject_1\{QGate\} \rightarrow$ Compiler

2 $\rightarrow PyObject_2\{Measurement\} \leftarrow$ From Compiler

4 $\rightarrow PyObject_3\{QReg_2\}$

Figure 6.2.: Hash-table storing of quantum simulator objects within SQASM compiler

```
1 ht_set (hashtable, $2, callpy ("INITIALIZE", tup));
2 return QReg (int (n))
3 next = hashtable->table[ht_hash(hashtable,key)]; // Place in hashtable
```

Figure 6.3.: From compiler to wrapper function to placement in hash-table. The cycle is traced explictly to show the outline in Figure 4.1

6.0.3. Quantum Simulator

The quantum simulator is used to perform all the necessary computation needed of the quantum programming language. It is also an independent component which can perform computation separately. An overview and description on a per-class basis is given in Figure 6.4.

Class	Description
QReg	Initialize quantum registers with amplitudes and a set definite state. One can also obtain the current state of the qubit with the *getState ()* function.
QSimulator	Measurement of quantum registers and selection of specific qubits from a quantum register. Application of quantum gates to registers. NAND gate implementation. Also included are quantum gate matrices such as the MTSG and Peres gate
QAdder	Complete implementation of the quantum ripple-carry adder in [Gos98] can be found here with the Quantum Majority Gate (QMG) and Quantum Full-Adder (QFA) split into two different functions for code reusability. The adder class also permits subtraction by using *Two's Complement*
QMultiplier	Contains a complete implementation of a quantum cost efficient multiplier circuit taken from a research paper [KG15].

Figure 6.4.: Class breakdown for quantum simulator in SQASM

In Figure 6.5, we outline the quantum register class. First, the number of qubits are set along with complex amplitudes in an array. If the user has set a definite value for the state of the quantum register, the state is set explicitly.

```
class QReg:
        def __init__ (self, n_qubits, setVal=-1):
            self.n_qubits = n_qubits
            self.qubits = [0] * n_qubits
            self.amps = [0] * (1 << n_qubits) # 2^n_qubits complex numbers
            self.amps[len (self.amps)-1] = 1
            if (setVal!= -1):
               self.amps[setVal] = 1
                if (setVal!=len (self.amps)-1):
                    self.amps[len (self.amps)-1] = 0
            self.amps = np.matrix (self.amps).T
```

Figure 6.5.: The simple *QReg* quantum register class inside SQASM's simulator

Creation of such a class allows for arbitrary quantum registers to be set. An example of initializing a quantum register can be found in Figure 6.6.

```
q = QReg (3, 6)
print ('QReg Amplitudes are:')
print (q.amps.T)
```

Figure 6.6.: Initialization of quantum register with three qubits set to $|110\rangle$

Quantum gate computation can be easily performed according to Figure 6.7. We start by initializing a quantum register with four qubits. Then Hadamard is applied to each qubit which is described in the circuit diagram outlined in Figure 6.8. Applying Hadamard to all qubits achieves superposition within each qubit. Note that the *t* function is the tensor function. Also note that displaying of amplitudes is strictly for testing purposes and does not portray true quantum computation due to violation of the uncertainty principle. However, as this is a simulator, testing can be very helpful while performing large amounts of quantum gate computation.

```
r = QReg(4)  # Get quantum system with 4 qubits
qs.applyGate (t (HAD, ID, ID, ID), r)  # Had bit1
qs.applyGate (t (ID, HAD, ID, ID), r)  # Had bit2
qs.applyGate (t (ID, ID, HAD, ID), r) # Had bit3
qs.applyGate (t (ID, ID, ID, HAD), r) # Had bit4
print (r.amps.T)
```

Figure 6.7.: Applying Hadamard gates within the quantum simulator

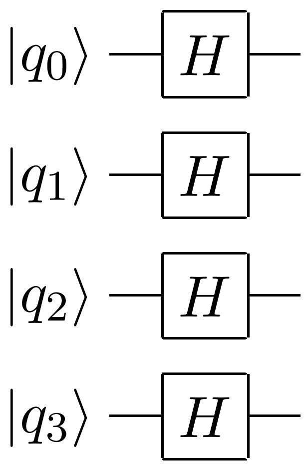


Figure 6.8.: Quantum circuit diagram compliment to Figure 6.7

Quantum arithmetic is also performed within the quantum simulator. The issue of adding and subtracting numbers can be addressed by a quantum ripple-carry adder. The circuitry outlined in Figures 6.9, 6.10 and 6.11 outline the quantum full adder, quantum majority gate and quantum ripple-carry adder respectively in accordance to [Gos98]. The full implementation of the circuitry can be seen in the Appendix under Quantum Simulator.

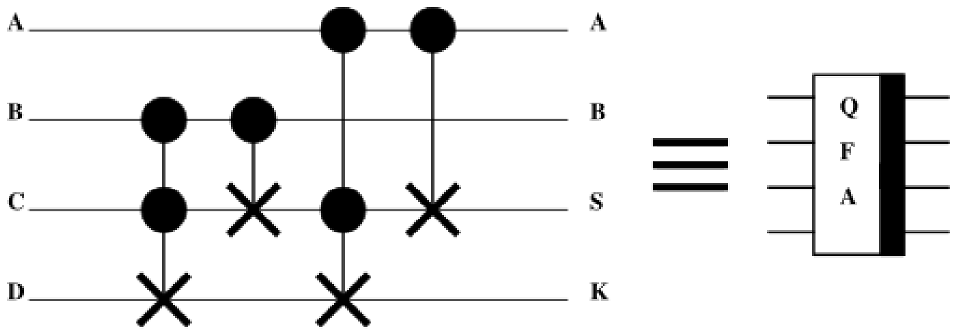


Figure 6.9.: Quantum Full Adder Circuitry

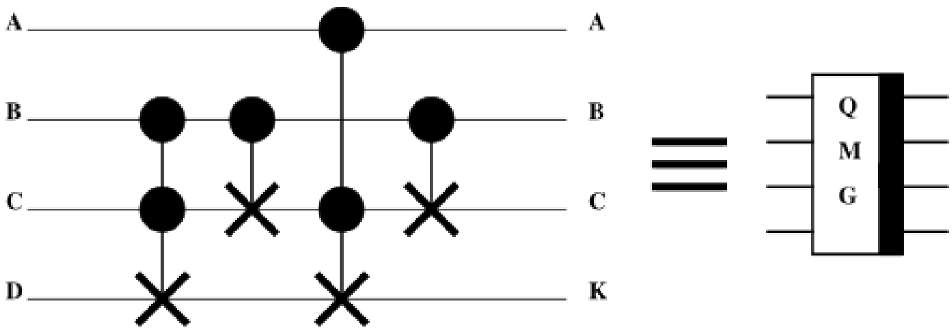


Figure 6.10.: Quantum Majority Gate Circuitry

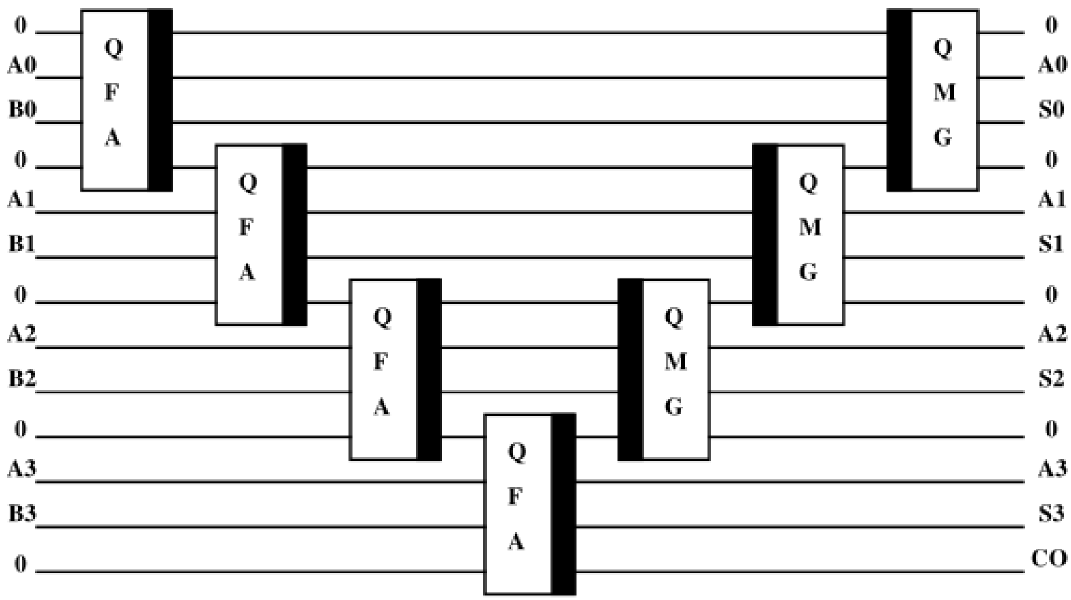


Figure 6.11.: Quantum Ripple Carry Adder

Multiplication can also be performed for an arbitrary bit size. The quantum cost efficient quantum multiplier utilizes PERES gates which are just AND gates with ancilla bits. These are used to form partial products. These partial products are summated to form the answer required. It's implementation thus can utilize the aforementioned ripple-carry adder and indeed it does. Again, the full implementation can be found within the Appendix. Testing of the multiplier and ripple carry-save is performed and analysed in Section 7.

6.0.4. Quantum Algorithms

The Deutsch-Jozsa algorithm is an algorithm which exploits entanglement principles. It's speedup is unprecedented and reduces $\Theta(2^n/2+1)$ to $\Theta(1)$ to evaluate $f(0) =? \ f(1)$. A formal outline is given,

For n qubits, evaluate the amount of qubits evenly distributed in states such that half fall on $|0\rangle$ and half on $|1\rangle$. The two possibilities are referred to as *balanced* or *constant*.

Let U_f be a quantum oracle function such that,

$$U_f : |x\rangle |y\rangle \rightarrow |x\rangle |f(x) \oplus y\rangle$$
$$f(0) \oplus f(1) = \begin{cases} 0 & \text{if same} \\ 1 & \text{if not same} \end{cases}$$

Using this quantum oracle function we can determine in one operation whether the function is balanced or constant. We start by initializing a quantum register or system with two qubits into $|0\rangle |1\rangle$ and apply Hadamard to both qubits.

$$\begin{aligned} \psi_{in} &= |0\rangle |1\rangle \\ H \times \psi &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= \frac{1}{2}(|0\rangle (|0\rangle - 1)) + |1\rangle (|0\rangle - |1\rangle) \end{aligned}$$

After doing so, we apply our quantum oracle to the equation above.

$$\begin{aligned} &= \frac{1}{2}[|0\rangle (|f(0) \oplus 0\rangle - |f(0) \oplus 1\rangle) + |1\rangle (|f(1) \oplus 0\rangle - |f(1) \oplus 1\rangle)] \\ &= \frac{1}{2}[-1^{f(0)} |0\rangle (|0\rangle - |1\rangle) + -1^{f(1)} |1\rangle (|0\rangle - |1\rangle)] \\ &= \frac{1}{2} -1^{f(0)}[|0\rangle + -1^{f(0) \oplus f(1)} |1\rangle](|0\rangle - 1) \end{aligned}$$

Then, we apply Hadamard again to get a deterministic result that tells us if the function is constant or balanced.

$$\psi_{out} = \frac{1}{2}(1 + -1^{f(0)\oplus f(1)}) |0\rangle + \frac{1}{2}(1 - 1^{f(0)\oplus f(1)}) |1\rangle$$
$$\text{Measure first qubit} \rightarrow \begin{cases} 0 & \text{same} \\ 1 & \text{not} \end{cases}$$

The following can be understood easier in implementation in Figure 6.12. First, we initialize a system of n qubits. Then, we apply the quantum oracle function. After this, we Hadamard each qubit again. Lastly, we perform measurements to get the deterministic result.

```
r = QReg (4, 0)  # Initialise system w/ 4 qubits
qs.applyGate (t (HAD, ID, ID, ID), r)  # Had 1st qubit
qs.applyGate (t (ID, HAD, ID, ID), r)  # Had 2nd qubit
qs.applyGate (t (ID, ID, HAD, ID), r)  # Had 3rd qubit
qs.applyGate (t (ID, ID, ID, HAD), r)  # Had 4th qubit
qs.quantumOracle (function,r)
qs.applyGate (t (HAD, ID, ID, ID), r)  # Had 1st qubit
qs.applyGate (t (ID, HAD, ID, ID), r)  # Had 2nd qubit
qs.applyGate (t (ID, ID, HAD, ID), r)  # Had 3rd qubit
qs.applyGate (t (ID, ID, ID, HAD), r)  # Had 4th qubit
for qubit in range (4):
        functionChanges |= (qs.measure (r,qubit)==1)

     if functionChanges:
         print ('Function is balanced')
     else:
         print ('Function is constant')
```

Figure 6.12.: Deutsch-Josza algorithm implementation in SQASM simulator

7. Testing: Verification and Validation

This section contains tests for quantum arithmetic and determinancy between simulator and compiler. Verification can also be made on the quantum mechanical principles entanglement and superposition. The ability to compute classically using a quantum simulator is proven. Quantum gate testing is considered in Section 7.0.8. The Deutsch-Josza algorithm is verified in full in Section 7.0.3 to prove that quantum algorithms can be deterministic and easily verified.

7.0.1. Quantum Arithmetic

Quantum addition and multiplication are proven to be deterministically accurate. The test for addition is illustrated in Figure 7.1.

The two numbers 10 and 12 are used in the test. Line 5 sets binary values to variables bin_1 and bin_2 dependent on the integer values. Log statements are output to the console to verify the binary values are indeed correct though a given unit test could have proved such. Line 8 performs the actual computation and the results are stored within the register and result variable respectively. Line 9 asserts whether the addition is correct. Upon success, a print statement lets the user know that the assertion passed. Note, *BIT_ARITHMETIC_AMOUNT* is set to 16 to perform 16-bit addition.

```
def adderTest (adder, qs, k=0):
    print ('BEGINNING ADDER TEST DONE')
    n1 = 10
    n2 = 12
    adder.setAdderBinaryValues (n1, n2)
    log (adder.bin_1)
    log (adder.bin_2)
    r, res = adder.rippleCarryAdder (BIT_ARITHMETIC_AMOUNT, qs)
    assert res == n1 + n2
    k = k + 1
    print ('TEST PASSED 16-BIT QUANTUM RIPPLE CARRY ADDER')
    return res
```

```
BEGINNING TESTS DONE
BEGINNING ADDER TEST DONE
ADD RESULT: 10 + 12 = 22
TEST PASSED 16-BIT QUANTUM RIPPLE CARRY ADDER
END OF TESTS DONE
```

Figure 7.1.: Quantum addition unit test with output

Quantum multiplication is tested with the function outlined in Figure 7.2. This unit test runs three times with 9, 10 and 11 for both a_1 and b_1 respectively. Again, binary versions of the numbers are created so that partial product generation can be performed with PERES gates. An assertion is run again to check the multiplication is successful. Upon success, the user gets a test passed message sent back. It should be noted also that the test extends further and evaluates how many gate operations were used for each section as given by [KG15]. The partial product generation takes 20 gate operations for standard 4-bit multiplication. Note that in Line 6, the qubit amount is set to 4 but can easily be extended to 8. Both 16 and 32-bit multiplication will stress RAM heavily due to large amounts of quantum register instantiation with high dimensional amplitude representations.

```
def multiplierTest (qs, k=0):
    print ('BEGINNING MULTIPLIER TEST DONE')
    for i in range (3):
        a1 = 9 + i
        b1 = 9 + i
        n1, n2 = m.prepMultiplier (a1, b1, 4)  # Creates binary versions of a1,b1
        res = m.applyMultiplier (qs, n1, n2)
        print ('ANSWER')
        print (bArrToDec (n1) * bArrToDec (n2))
        assert res == a1 * b1
        k = k + 1
        print ('TEST PASSED 4-BIT MULTIPLIER')
```

```
BEGIN MULTIPLIER[4 bits]
REGISTERS IN P1: 20
Time elapsed for cycle[0] in P2: 0.031205000000000001s
Time elapsed for cycle[1] in P2: 0.031936999999999999s
Time elapsed for cycle[2] in P2: 0.032575999999999994s
MUL R: 81 = 9 * 9
TEST PASSED 4-BIT MULTIPLIER

BEGIN MULTIPLIER[4 bits]
REGISTERS IN P1: 20
Time elapsed for cycle[0] in P2: 0.032203000000000001s
Time elapsed for cycle[1] in P2: 0.032034000000000001s
Time elapsed for cycle[2] in P2: 0.032387s
MUL R: 100 = 10 * 10
TEST PASSED 4-BIT MULTIPLIER

BEGIN MULTIPLIER[4 bits]
REGISTERS IN P1: 20
Time elapsed for cycle[0] in P2: 0.03236s
Time elapsed for cycle[1] in P2: 0.031893999999999998s
Time elapsed for cycle[2] in P2: 0.032219s
MUL R: 121 = 11 * 11
TEST PASSED 4-BIT MULTIPLIER
```

Figure 7.2.: Quantum multiplier unit test with output

7.0.2. Entanglement

Entanglement can be verified by performing the Hadamard and then the CNOT gate to a two qubit system as in Figure 2.1. This is accomplished in SQASM in Figure 7.3. The same is possible in the simulator as performed in Figure 7.4. The outputs demonstrates that the $|00\rangle$ amplitude is found to be probability $\frac{1}{2}$ and the $|11\rangle$ amplitude is found to be probability $\frac{1}{2}$. It should be noted again that probability is normalised.

```
INITIALIZE R 2
U TENSOR HAD ID
APPLY U R
APPLY CNOT R
PEEK R
```

```
<INPUT PEEK R RES>
GET Hash[82] -> R -> <sim.QReg instance at 0x7f139fed3e18>
SUCCESS Python Simulator Function Call
SET Hash[83]
GET Hash[83] -> RES -> matrix ([[ 0.70710678+0.j],
        [ 0.00000000+0.j],
        [ 0.00000000+0.j],
        [ 0.70710678+0.j]])
```

Figure 7.3.: Forming Bell States in SQASM with output

```
# Bell states demonstration- Entanglement of states
reg = QReg (2, 2)  # Init state |00>

# Remember- computation basis for 2 qubit system~|00>, |01>, |10>, |11>
qs.applyGate (t (HAD, ID), reg)  # Hadamard the first bit
qs.applyGate (CNOT, reg)  # Apply CNOT

print (reg.amps.T)  # readable form
```

```
[[0.70710678, 0, 0, -0.70710678]]
```

Figure 7.4.: EPR state β_{00} given in SQASM simulator

7.0.3. Deutsch-Jozsa Algorithm

The quantum algorithm is verified to be correct by applying four different functions. Namely the functions whereby the output is always one or always zero and is odd or is even. These functions are applied by the quantum oracle. The Deutsch-Jozsa algorithm can be found in Figure 6.12 and the output generated from calling the test is found in Figure 7.5 in simplified mathematical notation.

$U_f = \lambda = 0$
$\psi = |0000\rangle$
$H^4 \times \psi = \psi' = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]^T$
$U_f \times \psi' = \psi'' = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]^T$
$H^4 \times \psi'' = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$
$\therefore$ *constant*

$U_f = \lambda = 1$
$\psi = |0000\rangle$
$H^4 \times \psi = \psi' = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]^T$
$U_f \times \psi' = \psi'' = [\text{-}\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}]^T$
$H^4 \times \psi'' = [-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$
$\therefore$ *constant*

$U_f = \lambda = odd$
$\psi = |0000\rangle$
$H^4 \times \psi = \psi' = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]^T$
$U_f \times \psi' = \psi'' = [\frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}-, \frac{1}{4}]^T$
$H^4 \times \psi'' = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$
$\therefore$ *balanced*

$U_f = \lambda = even$
$\psi = |0000\rangle$
$H^4 \times \psi = \psi' = [\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}]^T$
$U_f \times \psi' = \psi'' = [\text{-}\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, -\frac{1}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{1}{4}]^T$
$H^4 \times \psi''' = [0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]^T$
$\therefore$ *balanced*

Figure 7.5.: Deutsch-Jozsa algorithm in SQASM Simulator output

7.0.4. Swap gate computation

Swap gates can be used to swap two qubits. The matrix representation is given in Figure 7.6. Verification of correct SWAP gate usage can be found in Figure 7.7. The output verifies the swap from $|10\rangle$ to $|01\rangle$.

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

Figure 7.6.: Quantum SWAP Gate Matrix

```
# SWAP gate demonstration on two qubit system
reg = QReg(2, 2)  # Init state |00>
print(reg.amps.T)  # Before swap
qs.applyGate(SWAP, reg)  # Apply SWAP gate
print(reg.amps.T)  # After swap
```

```
[[0 0 1 0 ]]
[[0 1 0 0 ]]
```

Figure 7.7.: SWAP on two qubit register with output for SQASM simulator

7.0.5. NAND gate computation

Using a Toffoli gate, one can form NAND gates and perform classical computation using a quantum simulator. The NAND gate formulation function is given in Figure 7.8 and called with the basis states for two qubits such that $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ are given as input using the arguments a and b in the function call `qs.NAND(a, b, QReg(3))`. A NAND gate is formed with an ancilla bit that is set to $|1\rangle$ and flipped according to two inputs. The ancilla bit is the output. Results and testing is given in Figure 7.9.

```
def NAND (self, a, b, r):
    binNumStr = str (a) + str (b) + str (1)
    binNum = int (binNumStr, 2)
    states = [0] * len (r.amps)
    states[binNum] = 1
    return self.applyGate (self.T, np.matrix (states).T)
```

Figure 7.8.: NAND gate formulation in SQASM simulator

```
# Classical computation NAND gate demonstration
reg = QReg (3)
print ('NAND |001> -> %s' % pretty(qs.NAND (0, 0, QReg (3)).T))
print ('NAND |011> -> %s' % pretty(qs.NAND (0, 1, QReg (3)).T))
print ('NAND |101> -> %s' % pretty(qs.NAND (1, 0, QReg (3)).T))
print ('NAND |111> -> %s' % pretty(qs.NAND (1, 1, QReg (3)).T))
```

```
NAND |001> -> |001>
NAND |011> -> |011>
NAND |101> -> |101>
NAND |111> -> |110>
```

Figure 7.9.: Quantum NAND gate test and output in SQASM simulator

7.0.6. Logging Control

Extensive testing can be enabled from within the quantum simulator by switching the `DEBUG = false` to `DEBUG = true`. This is just a simple way of controlling the amount of debug statements during simulation runs. This can be useful when extending the simulator and performing tests on complicated gate computation. The log function is outlined in Figure 7.10.

```
def log(s):
    if DEBUG:  # For debugging purposes
        print(s)
```

Figure 7.10.: Logging within SQASM quantum simulator

7.0.7. Python-C API Tests

Making sure the Python-C API is operating correctly is essential to determinacy within SQASM. A test for calling a simulator wrapper function is illustrated in Figure 7.11. `callpy()` is the function that makes calls to Python functions in the quantum simulator. Given that the compiler hasn't crashed at Line 4, the user receives a prompt to let them know a result was successfully obtained from the function call.

```
PyObject* callpy (char* f_name, PyObject *tup) {
   pFunc = PyDict_GetItemString (pDict, (char*) f_name);
   presult = call_pyfunc ();
   printf ("SUCCESS Python Simulator Function Call\n");
   return presult;
}
```

Figure 7.11.: Python-C API tests in SQASM compiler

7.0.8. Quantum Gate Tests with Truth Tables

Classical verification of gate computation is often given by truth tables. As such, one can extend this to quantum gate computation. Figure 7.12 illustrates verification of the PERES gate, an essential test as the PERES gate is used within quantum multiplication. Assertions are made against the truth table during computation.

```
def peresGateTest(qs):
    ''' PERES: TOF (C, B, A), flips A bit if C & B are set
               CNOT(C, B), flips B bit if C is set
        Image of circuit:
        http://www.informatik.uni-bremen.de/rev_lib/doc/real/peres_9.jpg '''

    # PERES gate truth table
    tt = np.matrix([[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 1, 0],
        [1, 1, 1], [1, 0, 1], [1, 0, 0]])

    for i in range(pow(2, 3)):
        r = QReg(3, i)  # iterating over all possibilities, i.e. 000, 001, etc
        qs.doPeresGate(r)  # apply the peres gate to those possibilities
        inp = getBinNum(i, 3)  # Our binary representation input we began with
        out = qs.measureMQubits(r, 3)  # Our binary result after PERES gate
        log('PERES[%s] R: %s' % (inp, out))  # log the result
        assert out == tt[i].tolist()[0]  # Check against the truth table
    print('PERES TEST SUCCESSFUL!')
```

Figure 7.12.: PERES gate unit test in SQASM simulator with truth table verification

8. Discussion

Included in this section are details of further work in lieu with literature and comments regarding results found in Section 7.

8.0.1. Quantum Arithmetic

Quantum arithmetic was verified to be correct for 16-bit addition and 4-bit multiplication as outlined in Section 7.0.1. This satisfies the criteria that the implementation must be able to perform arithmetic with constants and variable values.

The addition test could have contained further test cases, similar to that of multiplication. This would have provided a more solid foundation. Likewise, edge cases could have been identified and tested such as subtraction or perhaps numbers that were erroneous could have been attempted such as a number that is far too large.

Despite memory concerns, multiplication is likely to need at-least 16-bit numbers and as such optimizations will need to be made such as deleting ancillary bits instantly upon usage or perhaps dynamic switching of bit sizes on the fly. These types of optimizations can be found in [Han+16].

Providing gate operation count within quantum arithmetic is important and can be found within the multiplication test. This can be used to evaluate optimization and quickly see how efficient the arithmetic is. Even more useful would be attaching quantum cost to each gate and considering this as a form of metadata. Data structure manipulation could also provide optimization based on quantum architecture used.

Extra additions to addition and multiplication could include divide and fused multiply add, as seen in SIMD architecture.

8.0.2. Deutsch-Jozsa Algorithm

The verification of Deutsch-Jozsa algorithm shows that quantum algorithms can be easily implemented from within SQASM's simulator. This shows that other quantum algorithms can be added arbitrarily. The simple implementation educates the user considerably in the thinking behind quantum algorithms.

8.0.3. SWAP gate computation

The swap between the two qubits is successful. SWAP gates can be utilized for getting qubits into the correct ordering to apply certain gates such as CNOT. The gate is one of the most important gates in quantum computation as a result. The Quantum Ripple-Carry Adder and Quantum Cost-Efficient Multiplier both include SWAP gates in implementation.

8.0.4. NAND gate computation

All cases of a NAND gate input and output are shown to be correct. As a result, classical computation is proven to be possible as one can reflect NAND's truth table using the Toffoli gate.

8.0.5. Quantum Gate Testing Results

The usage of truth tables in quantum gate testing is highly useful for educative purpose and also for very complex quantum gates. Use cases could also include when one optimizes quantum gates and wishes to time how long they take to compute. This is certainly useful when one is testing quantum algorithms or compiling a quantum programming language for specific hardware architectures which is also done in [Han+16].

8.0.6. General Comments

It should be noted that a full testing suite would be needed upon scale of SQASM. This would help consolidate the open source effort and provide those who could be using it for their hardware architectures with the knowledge that SQASM is deterministic. Compilation to gate diagrams would also help educate the users and provide extra material in understanding quantum algorithms or quantum programs written with SQASM. This could be achieved by tieing into QASM [I05]. Linking of quantum libraries would promote code reuse also and going further it would be key to implement a Quantum Fast-Fourier Transform (QFFT) for implementation of Shor's algorithm.

9. Conclusion

SQASM has been presented as an easy to understand low-level quantum programming language. It's usability has been demonstrated in Section 6. The QPL can easily perform quantum algorithms, quantum arithmetic, generic quantum gate computation and adheres to QRAM architecture. Quantum mechanical concepts of entanglement and superposition are verified in Section 7 with other important quantum computational concepts such as classical computation proof.

Further work includes that of quantum oracle programming within SQASM such that the user can more easily write their own quantum algorithms. The Quantum-Fourier Transform should also be added to SQASM. The operation of fused multiply-add would be a very good operation to add to quantum arithmetic as well to save on gate operations. Classical means of programming can be used in tandem also to exploit the wide range of infrastructure already in place. Consideration of a graphical circuit per quantum program is also worthwhile and would provide educative and pragmatic benefit.

Lastly, as more hardware architectures come to light, the language could be changed to be more of an instruction set approach on a per situation basis. Gates could also be optimized on a hardware dependent basis. The nature of reversibility should also be addressed in further revisions of SQASM to allow a user to reverse their computation to a specified state back in time as demonstrated in [Han+16].

10. Project Commentary

Initial outlining of the PID had only included that of a quantum simulator. However, further review of the field indicated that quantum programming language design could be included and could provide more usefulness given that quantum machines are close to realisation, perhaps being as near as ten years away.

Therefore, the PID was modified heavily to include the QPL and as such, extra reviews into other QPL's and compiler literature became highly necessary. The review of the field showed that quantum arithmetic was an important aspect missing from the quantum simulator and initial QPL outline and thus quantum addition and multiplication were decidedly added at a later stage. Shor's algorithm was not implemented and instead the Deutsch-Josza algorithm was chosen as the algorithm illustrates quantum mechanical concepts far easier and the formalism behind Deutsch-Josza fits the educative purpose and simplicity theme far better.

Aside from these big additions, the PID was followed through in it's entirety and a simple construction of quantum computation with high functionality has been illustrated.

11. Social, Legal, Health, Safety and Ethical Issues

No software licencing has been broken in the creation of the QPL or Quantum Simulator. As stated, all work has been referenced if it has been of influence to the project.

The social impact of quantum computing is such that a change is needed in regards to cryptography and RSA. The large reshifting towards new quantum cryptography will reshape the way that ecommerce is performed online.

It should be stated also that SQASM will be found on GitHub [Wat16] and deployed with MIT licencing.

12. Reflection

The journey through quantum mechanics and quantum computing is an experience that takes one through various scientific fields. The amount of mathematical, physics, cryptographic, computer science and general scientist knowledge to be garnered from trying to build a quantum programming language and quantum simulator is unprecedented.

Typically issues upon study were mathematical or conceptual. For instance, linear superpositions are hard to fathom. How can one have a state that is able to perform computation on all states at once? Slowly as the building blocks of linear algebra were refined and learned, this became more and more apparent and clear.

Quantum mechanical thinking is trained and not environmentally given. Naturally, upon completion of the report, one would like to think that the training has reached a stage where the field can be analysed much more easier.

The undertaking of full-scale research has taught that one should persevere no matter what the circumstance and that the research can be very highly rewarding when results start to occur. It has also taught that the field that one wishes to study becomes more and more interesting the deeper one becomes immersed.

Most importantly of all, the project has taught that quantum computation is something highly valuable. The field is so wonderfully complex and new that all areas require research. I believe there are exciting times ahead for the fields Quantum AI, Quantum Cryptography, Quantum Error-Correction and Quantum Algorithms as the hardware progresses.

Appendices

A. Quantum Simulator

```python
#!/usr/bin/env python
# encoding: utf-8

from cmath import sqrt
import random
import numpy as np
import math
import time

# Quantum Simulator - Ryan Watkins
# MIT LICENCE

ONE_LOWER_TOLERANCE = 0.999
ONE_UPPER_TOLERANCE = 1.001
BIT_ARITHMETIC_AMOUNT = 16
DEBUG = False

# TODO: Fix multiplier - produce graphs of registers used and time spent in
# processing tests..
# TODO: ln 283: Should be a quantum operation


"""
Quantum Gate Matrices
"""

ID = np.matrix([[1, 0], [0, 1]])

CNOT = np.matrix([[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]])
```

```python

T = np.matrix([[1, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0],
               [0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 0],
               [0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 0, 0, 1, 0]])

SWAP = np.matrix([[1, 0, 0, 0], [0, 0, 1, 0],
                  [0, 1, 0, 0], [0, 0, 0, 1]])

HAD = np.matrix([[1 / sqrt(2), 1 / sqrt(2)], [1 / sqrt(2), -1 / sqrt(2)]])

NOT = np.matrix([[0, 1], [1, 0]])

COEF = (1 + 1j) / 2

# TODO: Fix CV/CVPLUS - ask question on stackexchange or find out elsewhere
CV = np.matrix([[1, 0, 0, 0], [0, 1, 0, 0],
                [0, 0, COEF * 1, COEF * -1j], [0, 0, COEF * - 1j, COEF * 1]])

COEF2 = (1 - 1j) / 2

CVPLUS = np.matrix([[1, 0, 0, 0], [0, 1, 0, 0],
                    [0, 0, COEF2 * 1, COEF2 * 1j],
                    [0, 0, COEF2 * 1j, COEF2 * 1]])


class QReg:
    def __init__(self, n_qubits, setVal=-1):
        self.n_qubits = n_qubits
        self.qubits = [0] * n_qubits
        # in this classical simulation, we use 2^n_qubits complex numbers
        self.amps = [0] * (1 << n_qubits)
        self.amps[len(self.amps) - 1] = 1
        if (setVal != -1):
            self.amps[setVal] = 1
            if (setVal != len(self.amps) - 1):
```

```
                self.amps[len(self.amps) - 1] = 0
        self.amps = np.matrix(self.amps).T

    def getState(self, qs):
        return qs.measureMQubits(self.amps)


class QSimulator:
    def NAND(self, a, b, r):
        binNumStr = str(a) + str(b) + str(1)
        binNum = int(binNumStr, 2)
        states = [0] * len(r.amps)
        states[binNum] = 1
        return np.dot(T, np.matrix(states).T)

    def doPeresGate(self, r):
        self.applyGate(T, r)
        self.applyGate(t(CNOT, ID), r)
        return r

    def doMTSGGate(self, r):
        """
        Implementation of quantum circuitry from:
        http://ijarcet.org/wp-content/uploads/IJARCET-VOL-4-ISSUE-4-1382-1386.pdf
        """
        # 1st op: v(b, d)
        self.applyGate(t(ID, SWAP, ID), r)
        self.applyGate(t(ID, ID, CV), r)
        self.applyGate(t(ID, SWAP, ID), r)
        # 2nd op: v(a, d)
        self.applyGate(t(SWAP, ID, ID), r)
        self.applyGate(t(ID, SWAP, ID), r)
        self.applyGate(t(ID, ID, CV), r)
        self.applyGate(t(ID, SWAP, ID), r)
        self.applyGate(t(SWAP, ID, ID), r)
        # 3rd op: cnot(a, b)
```

```
        self.applyGate(t(CNOT, ID, ID), r)
        # 4th op:
        self.applyGate(t(ID, ID, CV), r)
        log('After CV[3]: %s' % r.amps.T)
        # 5th op:
        self.applyGate(t(ID, CNOT, ID), r)
        # 6th op:
        self.applyGate(t(ID, ID, CVPLUS), r)
        log('After CPLUS: %s' % r.amps.T)

    def measure(self, r, q):
        oneProb, zeroProb = self.getProbsForQubit(r, q, r.amps[:])
        oneProb = abs(oneProb)
        zeroProb = abs(zeroProb)

        if(oneProb > 0.999):
            return 1
        elif(zeroProb > 0.999):
            return 0
        else:
            print('Undefinite state detected: probabilistic collapse needed')
            zeroProb = math.ceil(zeroProb * 100)
            oneProb = math.ceil(oneProb * 100)
            probs = [0] * int(zeroProb) + [1] * int(oneProb)  # int stops floats
            choice = random.choice(probs)
            return choice

    def measureMQubits(self, r, d_length=0):
        # st = time.clock()
        # for i in range(len(r.qubits)):
        # self.measure(r, r.qubits[i])
        # end = time.clock()
        # print('Time spent measuring all qubits of register: %s' % (end-st))
        for i in range(len(r.amps)):
            if (r.amps[i].real.item(0) > ONE_LOWER_TOLERANCE):
                return getBinNum(i, d_length)
```

```

# Apply a gate to a register
    def applyGate(self, u, r):
        r.amps = np.dot(u, r.amps)

    def select(self, r, off, n_qubits):
        selection = []
        for i in range(off, n_qubits + 1):
            selection.append(i)
        return selection

    def getProbsForQubit(self, r, q, amps, oneProb=0.0, zeroProb=0.0):
        log('Current qubit: %d' % q)
        log(r.amps)
        for index in range(0, len(r.amps)):
            bin_n = getBinNum(index, r.n_qubits)
            bin_n = list(reversed(bin_n))
            log('Bin number for index: %s' % bin_n)
            log('bin_n[q]: %d' % bin_n[q])
            if bin_n[q] == 1:   # If there is a 1 in column of index & mask
                oneProb += amps[index] * amps[index]
            else:
                zeroProb += amps[index] * amps[index]
        log('oneProb: %s, zeroProb: %s, qbit(%d)' % (oneProb, zeroProb, q))
        return (oneProb, zeroProb)

    def getAvgToAddFromOldStates(self, r, q, np_state, pStates=0, carry=0):
        for index in range(0, len(r.amps)):
            bin_n = getBinNum(index)
            if (bin_n[q] == np_state):
                carry += r.amps[index]
                r.amps[index] = 0
            elif (bin_n[q] != np_state and r.amps[index] != 0):
                pStates = pStates + 1
        avgToAdd = carry / pStates * carry / pStates
        return (avgToAdd, carry, pStates)
```

```
    def alterStates(self, r, q, np_state, pStates=0, carry=0):
        avgToAdd, carry, pStates = self.getAvgToAddFromOldStates(r, q, np_state)
        for index in range(0, len(r.amps)):
            bin_n = getBinNum(index)
            if (bin_n[q] != np_state and r.amps[index] != 0):
                r.amps[index] = r.amps[index] * r.amps[index] + avgToAdd

    def quantumOracle(self, function, r):
        "This is constant time on a quantum computer if f(x) is constant time"
        # We go in steps of 2 as the first qubit is not an input to our function
        for index in range(0, len(r.amps), 2):
            result = function(index // 2)  # Check if f(x) = balanced/constant
            #  print('result for ' + str(index) + ' // 2: ' + str(index // 2))
            if result == 1:
                r.amps[index] = - r.amps[index]
                r.amps[index + 1] = - r.amps[index + 1]

        print('After Uf (quantum oracle) applied: %s ' % r.amps.T)

    def isOne(self, number):  # evaluates if our number is 1.0
        #  isOne = number > self.ONE_LOWER_TOLERANCE and \
        #     number < self.ONE_UPPER_TOLERANCE
        #  print('eval in isOne(): %s, for number: ' % isOne, number)
        return number > ONE_LOWER_TOLERANCE and \
            number < ONE_UPPER_TOLERANCE


class Adder:
    def __init__(self):
        self.aOuts = [None] * BIT_ARITHMETIC_AMOUNT  # AOut in QAdder Paper
        self.bOuts = [None] * BIT_ARITHMETIC_AMOUNT  # bOuts in QA Paper
        self.sOuts = []  # output sums
        self.tZeros = [None] * BIT_ARITHMETIC_AMOUNT  # TODO
        self.bZero = 0   # TODO
        self.bin_1 = None
```

```
        self.bin_2 = None
        self.regs = []

    def clearVars(self):
        self.aOuts = []  # AOut in QAdder Paper
        self.bOuts = []  # bOuts in QA Paper
        self.sOuts = []  # output sums
        self.tZeros = []
        self.bZero = 0
        self.regs = []

    def rippleCarryAdderPreProcess(self, b1, b2, isSubtract):
        """
        Preprocess values so that continual adders can be applied
        and also utilize twos compliment in event of subtraction
        on the adder
        """
        self.clearVars()
        minusStr = self.minusStr(isSubtract)
        log('Beginning addition for: %s + (%s)%s' % (b1, minusStr, b2))

        if(isSubtract):
            b2 = self.twosCompliment(b2)
        return b2

    def rippleCarryAdder(self, nbits, qs, subtract=False, j=0):
        """
        Entirety of quantum adder processing is here,
        qs = Quantum Simulator, bin_1 = binary number 1,
        bin_2 = binary number 2, number of bits to perform addition on
        {tZero, aOuts, bOuts, bZero} are all outputs of implementation
        sOuts means outputs used for summation at the end...
        """
        #  TODO: make tZeros, aOuts & bOuts their own regs.

        log("bin_1: %s" % self.bin_1)
```

```
        log("bin_2: %s" % self.bin_2)

        # Preprocess to deal with subtraction edge case
        self.rippleCarryAdderPreProcess(self.bin_1, self.bin_2, subtract)
        log("BEGIN QFA PART OF QUANTUM RIPPLE CARRY ADDER")
        self.doQRCFullAdderPart(BIT_ARITHMETIC_AMOUNT, qs)
        self.sOuts.append(self.bOuts[0])  # Stores bOuts[0] for summation later
        self.logQFAOuts()

        log("BEGIN QMAJORITY PART OF QUANTUM RIPPLE ADDER")
        for i in range(1, nbits):
            tZero = self.tZeros[i]
            # Prepare a register for Quantum Majority Gate
            r = self.getQMAReg(tZero, self.aOuts[i], self.bOuts[i], self.bZero)

            self.applyQuantumMajorityGate(r, qs)

            # Measure and check results..
            m = qs.measureMQubits(r, nbits)
            log('After QMAJ: %s\n' % m)
            self.bZero = m[nbits - 4]

            self.sOuts.append(m[nbits - 2])

        log('Sums: %s' % self.sOuts)

        # Begin summation part..
        negBit = [self.sOuts[0]]

        for i in range(nbits - 1):
            negBit.append(0)

        negBit = int(''.join(map(str, negBit)), 2)
        self.sOuts[0] = 0

        # Joins all the sums
```

```
        result = int(''.join(map(str, self.sOuts)), 2)

        r = QReg(BIT_ARITHMETIC_AMOUNT, result)
        if (subtract):
            return (r, result + -(negBit))
        else:
            return (r, result + negBit)

    def doQRCFullAdderPart(self, nbits, qs):
        """
        Do Quantum Ripple Carry Full Adder processing
        """
        j = 0
        for i in range(nbits - 1, -1, -1):
            # Get the Quantum Full Adder Register by giving the ith element
            # of the binary numbers, bZero is the first element
            # We end up with a register like so: [bZero, bin1, bin2, 0]
            r = self.getQFAReg(self.bin_1[i], self.bin_2[i], self.bZero, j, 4)

            log('QFA Reg Begin: %s' % r.amps.T)

            self.applyQuantumFullAdder(r, qs)  # Do actual gate operations

            # Check if our quantum full adder worked.
            m = qs.measureMQubits(r, 4)
            log('After QFA: %s\n' % m)

            # Store vals for Quantum Majority Gate portion of implementation
            self.storeQFAValues(m)
            j += 1  # for iterative purposes

    def getQFAReg(self, a, b, bZero, j, nbits):
        bState = [str(bZero), str(a), str(b), str(0)]
        bState = ''.join(bState)
        state = int(bState, 2)
        log('QReg[%s]: %s' % (j, bState))
```

```
        r = QReg(nbits, state)
        self.regs.append(r)
        return r

    def logQFAOuts(self):
        log('tZeros: %s' % self.tZeros)
        log('aOuts: %s' % self.aOuts)
        log('bOuts: %s' % self.bOuts)
        log('bZero: %s' % self.bZero)

    def storeQFAValues(self, m):
        ""
         Store vals to plug back into our Quantum Majority Gate portion of
        implementation
        ""
        self.tZeros.insert(0, m[0])
        self.aOuts.append(m[1])
        self.bOuts.insert(0, m[2])
        self.bZero = m[3]

    def twosCompliment(self, b):
        bState = ''.join(map(str, b))
        state = int(bState, 2)
        print('Initial binary state before inversion: %s' % bState)
        print('Initial state before inversion: %s' % state)
        r = QReg(4, state)
        print("Prior to one's compliment, amps: %s" % r.amps.T)
        qs.applyGate(t(NOT, ID, ID, ID), r)
        qs.applyGate(t(ID, NOT, ID, ID), r)
        qs.applyGate(t(ID, ID, NOT, ID), r)
        qs.applyGate(t(ID, ID, ID, NOT), r)
        print("One's compliment amps: %s" % r.amps.T)
        m = qs.measureMQubits(r, 4)
        print("One's compliment: %s" % m)
        oc = int(''.join(map(str, m)), 2)
        ocPlusOne = oc + 1
```

```
        print("One's compliment integer value: %s" % oc)
        print("One's compliment plus one: %s" % ocPlusOne)
        res = getBinNum(ocPlusOne, BIT_ARITHMETIC_AMOUNT)
        log('After INVERT: %s' % res)
        return res

    def minusStr(self, isSubtract):
        if(isSubtract):
            return '-'

    def testQFA(self, qs):
        """
        Test against Table 2 - Quantum Full Adder - paper ref (below)
        """
        tt = np.matrix([[0, 0, 0, 0], [1, 0, 0, 0], [0, 1, 0, 0], [1, 1, 0, 0],
                        [0, 1, 1, 0], [1, 1, 1, 0], [1, 0, 1, 0], [0, 0, 1, 0],
                        [0, 1, 0, 1], [1, 1, 0, 1], [1, 0, 0, 1], [0, 0, 0, 1],
                        [1, 0, 1, 1], [0, 0, 1, 1], [1, 1, 1, 1], [0, 1, 1, 1]])
        for i in range(16):
            r = QReg(4, i)
            self.qfadder(r)
            inp = getBinNum(i, 4)[::-1]
            out = qs.measureMQubits(r, 4)[::-1]
            print('QFA[%s] R: %s' % (inp, out))
            assert out == tt[i].tolist()[0]

    def setAdderBinaryValues(self, n1, n2):
        bitsAvailable = pow(2, BIT_ARITHMETIC_AMOUNT - 1)
        if (n1 >= bitsAvailable or n2 >= bitsAvailable):
            raise Exception('Value error: Integer too big for addition')
        self.bin_1 = getBinNum(n1, BIT_ARITHMETIC_AMOUNT)
        self.bin_2 = getBinNum(n2, BIT_ARITHMETIC_AMOUNT)

    def testAdder(res, n1, n2):
        print('ADD RESULT: %s' % res)
        assert res == n1 + n2
```

```
        print('SUCCESSFUL RESULT \n\n')

    def applyQuantumMajorityGate(self, r, qs):
        qs.applyGate(t(ID, T), r)  # 1st op - Toffoli (b, c, d)
        qs.applyGate(t(ID, CNOT, ID), r)  # 2nd op - CNOT (b, c)
        qs.applyGate(t(ID, SWAP, ID), r)  # sw(b, c) -> (a, c, b, d)
        qs.applyGate(t(ID, ID, SWAP), r)  # sw(b, d) -> (a, c, d, b)
        qs.applyGate(t(T, ID), r)  # tof(a, c, d)
        qs.applyGate(t(ID, ID, SWAP), r)  # sw(b, d) -> (a, c, b, d)
        qs.applyGate(t(ID, SWAP, ID), r)  # sw(c, b) -> (a, b, c, d)
        qs.applyGate(t(ID, CNOT, ID), r)  # 4th op - CNOT (b, c)

    def getQMAReg(self, tZero, a, b, bZero):
        bState = [str(tZero), str(a), str(b), str(bZero)]
        bState = ''.join(bState)
        state = int(bState, 2)
        r = QReg(4, state)
        return r

    def applyQuantumFullAdder(self, r, qs):
        """
        Quantum full adder implementation ref:
        http://arxiv.org/pdf/quant-ph/9808061.pdf
        """
        qs.applyGate(t(ID, T), r)  # 1st op - TOF(b, c, d)
        qs.applyGate(t(ID, CNOT, ID), r)  # 2nd op - CNOT(b, c)

        # 3rd op: TOF(A, C, D), => SWAP B & C, SWAP B & D, SWAP BACK
        qs.applyGate(t(ID, SWAP, ID), r)
        qs.applyGate(t(ID, ID, SWAP), r)
        qs.applyGate(t(T, ID), r)

        # Now we need to swap d & b, then c and b and we're back to normal
        qs.applyGate(t(ID, ID, SWAP), r)
        qs.applyGate(t(ID, SWAP, ID), r)

```

```
        # 4th op - need to swap b & c and back again after cnot(a, c)
        qs.applyGate(t(ID, SWAP, ID), r)
        qs.applyGate(t(CNOT, ID, ID), r)
        qs.applyGate(t(ID, SWAP, ID), r)


class Multiplier():
    def __init__(self):
        self.regs = []
        self.sumRegs = []

    def applyMultiplier(self, qs, b1, b2, l=0, m=0, prevRes=0):
        '' Fig 9. Fig. 10 from paper: http://ijarcet.org/?page_id=3143
            for part1 and part2 respectively                                  ''

        log('Applying multiplication to: %s * %s' % (b1, b2))
        bLength = len(b1)  # Get length of binary values passed in
        log('bLength: %s' % bLength)

        # Partial Product Generation (Using PERES Gate to generate AND gate)
        # Does PERES gate, line by line (x[j], y[i])
        # x0 is b1[0], y0 is b2[0]
        for i in range(bLength - 1, -1, -1):
            m = 0  # Used to keep track of variables in logging
            for j in range(bLength - 1, -1, -1):
                r = self.prepAND(b1[j], b2[i], i)
                log('Begin state x[%s]y[%s]: %s ' % (m, l, r.amps.T))
                r = qs.doPeresGate(r)
                log('After PERES x[%s]y[%s]: %s\n' % (m, l, r.amps.T))
                m += 1
                if (j == 0):
                    s = []  # s is the sum array of the and operations

                    for k in range(bLength - 1, -1, -1):
                        log("Reg accessed for sum: %s" % (k + (l * bLength)))
                        m = qs.measureMQubits(self.regs[k+(l * bLength)],
```

```
                                                 bLength - 1)
                        log("Measurement on reg %s" % m)
                        log("Sum is appending val: %s" % (m[bLength - 4 + 2]))
                        s.append(m[bLength - 4 + 2])
                    for p in range(l):  # amount to start of binary
                        s.append(0)
                    for p in range(bLength - l - 1):
                        s.insert(0, 0)

                    l = l + 1

                    # Creates a new quantum register to store sums
                    self.sumRegs.append(QReg(bLength + 1, bArrToDec(s)))
                    log('Summation of (%s)th line: %s\n' % (l, s))

        print('REGISTERS IN P1: %s' % (len(self.regs) + len(self.sumRegs)))
        self.regs = []
        for i in range(len(self.sumRegs) - 1):
            st = time.clock()
            if(prevRes == 0):
                # st2 = time.clock()
                a = qs.measureMQubits(self.sumRegs[i], bLength)
                # end2 = time.clock()
                # print('Time elapsed measuring a: %s' % (end2-st2))
            else:
                a = getBinNum(prevRes, bLength)
            # st2 = time.clock()
            b = qs.measureMQubits(self.sumRegs[i + 1], bLength)
            # end2 = time.clock()
            # print('Time elapsed measuring b: %s' % (end2-st2))
            log('a: %s' % a)
            log('b: %s' % b)
            adder.setAdderBinaryValues(bArrToDec(a), bArrToDec(b))
            r, prevRes = adder.rippleCarryAdder(BIT_ARITHMETIC_AMOUNT, qs)
            end = time.clock()
            print('Time elapsed for cycle[%s] in P2: %ss' % (i, end-st))
```

```
        res = prevRes
        self.sumRegs = []
        return res

    def getState(self, i1, i2, i3=0, i4=0):
        if (i3 == 0):
            return int(''.join([str(i1), str(i2)]), 2)
        elif (i4 == 0):
            return int(''.join([str(i1), str(i2), str(i3)]), 2)
        else:
            return int(''.join([str(i1), str(i2), str(i3), str(i4)]), 2)

    def prepMultiplier(self, n1, n2, b_amount=4):
            print('\nBEGIN MULTIPLIER[%s bits]' % b_amount)
            b1 = getBinNum(n1, b_amount)
            b2 = getBinNum(n2, b_amount)
            return b1, b2

    def prepAND(self, a, b, j):
        bState = [str(a), str(b), str(0)]
        bState = '.join(bState)
        state = int(bState, 2)
        log('QReg[%s]: %s' % (j, bState))
        r = QReg(3, state)  # Note: [1 0 0 0 0 0 0 0] - means 000
        self.regs.append(r)  # Saving reg for later use
        return r


# Various ancillary functions
def t(f1, f2, f3=0, f4=0):
    """
    Tensor product for up to three functions
    """

    if(type(f3) is int):
        return np.kron(f1, f2)
```

```
    elif(type(f4) is int):
        u = np.kron(f1, f2)
        return np.kron(u, f3)
    else:
        u = np.kron(f1, f2)
        u = np.kron(u, f3)
        return np.kron(u, f4)


def bArrToDec(ba):
    return int(''.join(map(str, ba)), 2)


def dec_to_bin(x):
    return int(bin(x)[2:])


def fixBinToDec(x, d_length):
    if (len(x) < d_length):
        amountToPad = d_length - len(x)
        for i in range(amountToPad):
            x.insert(0, 0)
        return x
    else:
        return x


def getBinNum(x, d_length=0):
    if (d_length == 0):
        d_length = len(bin(x))
    bin_n = [int(i) for i in str(dec_to_bin(x))]
    return fixBinToDec(bin_n, d_length)


def log(s):
    if DEBUG:  # For debugging purposes
```

```
        print(s)


def checkProbs(l):
    ""
    Checks if probabilities add to one in sufficient manner
    ""
    probs = sum(abs(i)*abs(i) for i in l)
    probs = probs.item(0)
    assert probs < ONE_UPPER_TOLERANCE and probs > ONE_LOWER_TOLERANCE


# Wrapper functions for quantum programming language SQASM
def MEASURE(r):
    '' Measurement on a given register in a given range
        r[0] = selection, r[1] = reg - Error handling for
        other situations included ''
    qs = QSimulator()

    # Error handling for passing various value types from compiler
    try:
        selection = r[0]
    except AttributeError:
        selection = [0, r.n_qubits - 1]
    try:
        reg = r[1]
    except AttributeError:
        reg = r

    res = []

    print("Amount of amplitudes in register %s" % len(reg.amps))
    print('selection range: %s' % selection)
    begin = selection[0]
    end = selection[1] + 1

```

```
    for i in range(begin, end):
        res.append(qs.measure(reg, i))

    print('RES: %s' % res)  # Reads left to right in order of qubits
    return res


def SELECT(r, begin, end):
    qs = QSimulator()
    return (qs.select(r, begin, end), r)


def INITIALIZE(n, pos):
    return QReg(int(n), pos)


def APPLY(gate, qreg):
    qs = QSimulator()
    qs.applyGate(gate, qreg)
    return qreg


def ADD(a, b):
    r = QReg(BIT_ARITHMETIC_AMOUNT)  # 16 bit addition
    t0 = time.clock()
    adder = Adder()
    adder.setAdderBinaryValues(a, b)
    log('a: %s, b: %s' % (adder.bin_1, adder.bin_2))
    qs = QSimulator()
    r, res = adder.rippleCarryAdder(BIT_ARITHMETIC_AMOUNT, qs)
    print('Elapsed time for add: %ss' % (time.clock() - t0))
    log('ADD RESULT: %s + %s = %s' % (a, b, res))
    assert res == a + b
    print('SUCCESS: ADD')
    return r

```

```
def PEEK(r):
    return r.amps


# Deutsch's algorithm functions
def alwaysZero(value):
    return 0


def alwaysOne(value):
    return 1


def isOdd(value):
    #  print('value & 1 inside isOdd is: ' + str(value & 1))
    return (value & 1)


def isEven(value):
    #  print('(value ^ 1) & 1 is: ' + str((value ^ 1) & 1))
    return (value ^ 1) & 1


functionList = [
    (alwaysZero, "AlwaysZero"),
    (alwaysOne, "AlwaysOne"),
    (isOdd, "isOdd"),
    (isEven, "isEven")
]


# Testing functions
def adderTest(adder, qs, k=0):
    print('BEGINNING ADDER TEST... DONE')
    n1 = 10
```

```
    n2 = 12
    adder.setAdderBinaryValues(n1, n2)
    log("Trying to do %s + %s" % (n1, n2))
    log('a: %s' % adder.bin_1)
    log('b: %s' % adder.bin_2)
    r, res = adder.rippleCarryAdder(BIT_ARITHMETIC_AMOUNT, qs)
    print('ADD RESULT: %s + %s = %s' % (n1, n2, res))
    assert res == n1 + n2
    k = k + 1
    print('TEST PASSED 16-BIT QUANTUM RIPPLE CARRY ADDER')
    return res


def multiplierTest(qs, k=0):
    print('\nBEGINNING MULTIPLIER TEST... DONE')
    for i in range(3):
        a1 = 9 + i
        b1 = 9 + i
        n1, n2 = m.prepMultiplier(a1, b1, 4)  # a1,b1 -> binary
        res = m.applyMultiplier(qs, n1, n2)
        print('MUL R: %s = %s * %s' % (res, bArrToDec(n1), bArrToDec(n2)))
        assert res == a1 * b1
        k = k + 1
        log('Successful results: %s' % k)
        print('TEST PASSED 4-BIT MULTIPLIER')


def MTSGGateTest(qs):
    for i in range(pow(2, 4)):
        r = QReg(4, i)
        qs.doMTSGGate(r)
        inp = getBinNum(i, 4)
        out = qs.measureMQubits(r, 4)
        log('MTSG[%s] R: %s' % (inp, out))
    print('MTSG Gate Test SUCCESSFUL')
```

```python

def peresGateTest(qs):
    '' PERES: TOF (C, B, A), flips A bit if C & B are set
              CNOT(C, B), flips B bit if C is set
        Image of circuit:
        http://www.informatik.uni-bremen.de/rev_lib/doc/real/peres_9.jpg ''

    # PERES gate truth table
    tt = np.matrix([[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 1, 0],
                    [1, 1, 1], [1, 0, 1], [1, 0, 0]])

    for i in range(pow(2, 3)):
        r = QReg(3, i)  # iterating over all possibilities, i.e. 000, 001, ...
        qs.doPeresGate(r)  # apply the peres gate to those possibilities
        inp = getBinNum(i, 3)  # Our binary representation input we began with
        out = qs.measureMQubits(r, 3)  # Our binary result after PERES gate
        log('PERES[%s] R: %s' % (inp, out))  # log the result
        assert out == tt[i].tolist()[0]  # Check against the truth table
    print('PERES TEST SUCCESSFUL!')


def qsimDeutschTest(qs):
    ""
    David Deutsch's Algorithm (1992)
    ""
    for function, name in functionList:
        r = QReg(4, 0)
        print("Beginning amps: %s" % r.amps.T)
        qs.applyGate(t(HAD, ID, ID, ID), r)
        qs.applyGate(t(ID, HAD, ID, ID), r)
        qs.applyGate(t(ID, ID, HAD, ID), r)
        qs.applyGate(t(ID, ID, ID, HAD), r)
        print("After hadding all bits: %s" % r.amps.T)

        qs.quantumOracle(function, r)

```

```
        qs.applyGate(t(HAD, ID, ID, ID), r)
        qs.applyGate(t(ID, HAD, ID, ID), r)
        qs.applyGate(t(ID, ID, HAD, ID), r)
        qs.applyGate(t(ID, ID, ID, HAD), r)
        print("After hadding all bits again: %s" % r.amps.T)
        functionChanges = False

        for qubit in range(4):
            functionChanges |= (qs.measure(r, qubit) == 1)

        if functionChanges:
            print("FOUND RESULT: %s is balanced\n" % name)
        else:
            print("FOUND RESULT: %s is constant\n" % name)


def wrapperTests():
    r = ADD(15, 35)
    MEASURE(([0, 15], r))  # Somehow need to verify this


def runTests(qs, adder, m):
    print('BEGINNING TESTS... DONE')
    # adderTest(adder, qs)
    multiplierTest(qs)
    # peresGateTest(qs)
    # MTSGGateTest(qs)
    # qsimDeutschTest(qs)
    # wrapperTests()
    print('END OF TESTS... DONE')


def pretty(reg, y=0):
    x = "{0:b}".format(reg.argmax())
    zeroes = '
    y = y + 2 if reg.argmax() <= 1 else y + 1 if reg.argmax() <= 3 else 0
```

```python

    for i in range(y):
        zeroes = zeroes + '0'

    return str('|' + zeroes + str(x) + '>')

if __name__ == "__main__":
    adder = Adder()
    m = Multiplier()
    qs = QSimulator()

    # Bell states demonstration - Entanglement of states
    # reg = QReg(2, 2)  # Init state |00>

    # Remember - computation basis for 2 qubit system ~ |00>, |01>, |10>, |11>
    # qs.applyGate(t(HAD, ID), reg)  # Hadamard the first bit
    # qs.applyGate(CNOT, reg)  # Apply CNOT

    # print(reg.amps.T)  # readable form

    # SWAP gate demonstration on two qubit system
    # reg = QReg(2, 2)  # Init state |00>
    # print(reg.amps.T)  # Before swap
    # qs.applyGate(SWAP, reg)  # Apply SWAP gate
    # print(reg.amps.T)  # After swap

    # Classical computation NAND gate demonstration
    # reg = QReg(3)
    # print('NAND |001> -> %s' % pretty(qs.NAND(0, 0, QReg(3)).T))
    # print('NAND |011> -> %s' % pretty(qs.NAND(0, 1, QReg(3)).T))
    # print('NAND |101> -> %s' % pretty(qs.NAND(1, 0, QReg(3)).T))
    # print('NAND |111> -> %s' % pretty(qs.NAND(1, 1, QReg(3)).T))

    # -- TESTING - #
    runTests(qs, adder, m)

```

```
    print('Starting up quantum simulator...   DONE')'))'
]')))'"
```

B. SQASM Parser

```
%{
#define _XOPEN_SOURCE 500 /* Enable certain library functions (strdup) on linux */

#include <stdio.h>     /* C declarations used in actions */
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <Python.h>

int yylex();
void yyerror (const char *s);

/* Purely hashtable */
struct entry_s {
        char *key;
        PyObject *value;
        struct entry_s *next;
};

typedef struct entry_s entry_t;

struct hashtable_s {
        int size;
        struct entry_s **table;
};
typedef struct hashtable_s hashtable_t;

hashtable_t *hashtable;

```

```
hashtable_t *ht_create( int size );
int ht_hash( hashtable_t *hashtable, char *key );
entry_t *ht_newpair( char *key, PyObject *value );
void ht_set( hashtable_t *hashtable, char *key, PyObject *value );
PyObject *ht_get( hashtable_t *hashtable, char *key );

/* Purely Python C-API */
char str[15]; char str2[15];

PyObject *pName, *pModule, *pDict, *pFunc, *pValue, *presult, *tup, *v, *v2;
PyObject* callpy(char* f_name, PyObject* tup);
PyObject* get_pytup(void* a1, void* a2, void* a3, char* t1, char* t2, char* t3, int n_arg
PyObject* call_pyfunc();

int set_tupitem(char* type, void* item, int pos);
%}

%union {int num; char* id;}         /* Yacc definitions */
%start line
%token print
%token exit_command
%token init
%token tensor
%token sel
%token measure
%token add
%token peek
%token <id>gate
%token apply
%token <num> number
%token <id> identifier
%type <num> line exp term
%type <id> assignment

%%

```

```
/* descriptions of expected inputs     corresponding actions (in C) */

line    : assignment                        {;}
                | exit_command                 {exit(EXIT_SUCCESS);}
                | print exp                 {printf("Printing %d\n", $2);}
                | line assignment         {;}
                | line print exp        {printf("Printing %d\n", $3);}
                | line exit_command         {exit(EXIT_SUCCESS);}
                | line exp              {;}
                | exp                        {;}
        ;

assignment : init term term term {
           printf("\n<INPUT: INIT %s %i %i>\n", $2, $3, $4);
           tup = get_pytup($3, $4, " ", "int", "int", NULL, 2);
           ht_set(hashtable, $2, callpy("INITIALIZE", tup));
           ht_get( hashtable, $2 ); }
        ;
exp            : term                 {$$ = $1;}
        | exp ’+’ term          {$$ = $1 + $3;}
               | exp ’-’ term          {$$ = $1 - $3;}
        | add term term term        {
        printf("\n<INPUT: ADD %i %i %s>\n", $2, $3, $4);
        tup=get_pytup($2, $3, " ", "int", "int", " ", 2);
        ht_set(hashtable, $4, callpy("ADD", tup));
        ht_get( hashtable, $4 ); }
        | term tensor gate gate   {
        printf("\n<INPUT: %s TENSOR %s %s>\n", $1, $3, $4);
        tup = get_pytup($3, $4, " ", "str", "str", NULL, 2);
        ht_set(hashtable, $1, callpy("t", tup));
        ht_get( hashtable, $1 ); }
        | term tensor term term {
        printf("\n<INPUT: %s TENSOR %s %s>\n", $1, $3, $4);
        tup = get_pytup(ht_get(hashtable, $3), ht_get(hashtable, $4), " ", "py", "py", NU
        ht_set(hashtable, $1, callpy("t", tup));
        ht_get( hashtable, $3 ); }
```

```
        | apply term term       {
        printf("\n<INPUT: APPLY %s %s>\n", $2, $3);
        tup = get_pytup(ht_get(hashtable, $2), ht_get(hashtable, $3), " ", "py", "py", NU
        ht_set(hashtable, $3, callpy("APPLY", tup));
        ht_get( hashtable, $3 );
        }
        | apply gate term        {
        printf("\n<INPUT: APPLY GATE TERM>\n", $2, $3);
        tup = get_pytup($2, ht_get(hashtable, $3), " ", "str", "py", NULL, 2);
        ht_set(hashtable, $3, callpy("APPLY", tup));
        ht_get( hashtable, $3 );}
        | measure term term       {
        printf("\n<INPUT: MEASURE %s %s", $2, $3);
        printf(">\n");
        tup = get_pytup(ht_get( hashtable, $2), " ", " ", "py", NULL, NULL, 1);
        ht_set(hashtable, $3, callpy("MEASURE", tup));
        ht_get( hashtable, $3 ); }
        | peek term term {
        printf("\n<INPUT: PEEK %s %s", $2, $3);
        printf(">\n");
        tup = get_pytup(ht_get( hashtable, $2), " ", " ", "py", NULL, NULL, 1);
        ht_set(hashtable, $3, callpy("PEEK", tup));
        ht_get( hashtable, $3 ); }
        | sel term term term term {
        printf("\n<INPUT: SELECT %s %s %i %i>\n", $2, $3, $4, $5);
        tup = get_pytup(ht_get(hashtable, $3), $4, $5, "py", "int", "int", 3);
        ht_set(hashtable, $2, callpy("SELECT", tup));
        ht_get( hashtable, $2 ); }
        ;
term            : number                {$$ = $1;}
        | identifier                {$$ = $1;}
        ;

%%                      /* C code */


```

```
/* Create a new hashtable. */
hashtable_t *ht_create( int size ) {

        hashtable_t *hashtable = NULL;
        int i;

        if( size < 1 ) return NULL;

        /* Allocate the table itself. */
        if( ( hashtable = malloc( sizeof( hashtable_t ) ) ) == NULL ) {
                return NULL;
        }

        /* Allocate pointers to the head nodes. */
        if( ( hashtable->table = malloc( sizeof( entry_t * ) * size ) ) == NULL ) {
                return NULL;
        }
        for( i = 0; i < size; i++ ) {
                hashtable->table[i] = NULL;
        }

        hashtable->size = size;

        return hashtable;
}

/* Hash a string for a particular hash table. */
int ht_hash( hashtable_t *hashtable, char *key ) {
        unsigned long int hashval;
        int i = 0;

        while( hashval < ULONG_MAX && i < strlen( key ) ) {
                hashval = hashval << 8;
                hashval += key[ i ];
                i++;
        }
```

```
        return hashval % hashtable->size;
}

/* Create a key-value pair. */
entry_t *ht_newpair( char *key, PyObject *value ) {
        entry_t *newpair;

        if( ( newpair = malloc( sizeof( entry_t ) ) ) == NULL ) {
                return NULL;
        }
        if( ( newpair->key = strdup( key ) ) == NULL ) {
                return NULL;
        }
        if( ( newpair->value = value ) == NULL ) {
                return NULL;
        }
        newpair->next = NULL;

        return newpair;
}

/* Insert a key-value pair into a hash table. */
void ht_set( hashtable_t *hashtable, char *key, PyObject *value ) {

        int bin = 0;
        entry_t *newpair = NULL;
        entry_t *next = NULL;
        entry_t *last = NULL;
        bin = ht_hash( hashtable, key );
        next = hashtable->table[ bin ];

        printf("SET Hash[%i]\n", bin);

        while( next != NULL && next->key != NULL && strcmp( key, next->key ) > 0 ) {
                last = next;
                next = next->next;
```

```
        }
        /* There's already a pair.  Let's replace that string. */
        if( next != NULL && next->key != NULL && strcmp( key, next->key ) == 0 ) {
                printf("Found a pair already on key: %s...\n", key);
                next->value = value;

        /* Nope, could't find it.  Time to grow a pair. */
        } else {
                newpair = ht_newpair( key, value );

                /* We're at the start of the linked list in this bin. */
                if( next == hashtable->table[ bin ] ) {
                        newpair->next = next;
                        hashtable->table[ bin ] = newpair;

                /* We're at the end of the linked list in this bin. */
                } else if ( next == NULL ) {
                        last->next = newpair;

                /* We're in the middle of the list. */
                } else  {
                        newpair->next = next;
                        last->next = newpair;
                }
        }
}

/* Retrieve a key-value pair from a hash table. */
PyObject *ht_get( hashtable_t *hashtable, char *key ) {
        int bin = 0;
        entry_t *pair;
        bin = ht_hash( hashtable, key );
        printf("GET Hash[%i] -> ", bin);
        /* Step through the bin, looking for our value. */
        pair = hashtable->table[ bin ];
        while( pair != NULL && pair->key != NULL && strcmp( key, pair->key ) > 0 ) {
```

```
                pair = pair->next;
        }
        /* Did we actually find anything? */
        if( pair == NULL || pair->key == NULL || strcmp( key, pair->key ) != 0 ) {
                printf("ERROR: Found nothing from hashtable for key: %s\n", key);
        /* So we'll just return the key then... */
                return key;

        } else {
                printf("%s -> ", key);
                PyObject_Print(pair->value, stdout, 0); printf("\n");
                return pair->value;
        }

}

PyObject* callpy(char* f_name, PyObject *tup)
{
           pFunc = PyDict_GetItemString(pDict, (char*)f_name);
        presult = call_pyfunc();
        printf("SUCCESS: Python Simulator Function Call\n");
        return presult;
}

PyObject* get_pytup(void* a1, void* a2, void* a3, char* t1, char* t2, char* t3, int n_arg
{
        tup = PyTuple_New(n_args);
        //printf("initialised tuple with %d args...\n", n_args);
        PyErr_Print();

        if (a1 != " ") {
                set_tupitem(t1, a1, 0);
                // printf("SUCCESS set first tup item..\n");
        }
        if (a2 != " ") {
                set_tupitem(t2, a2, 1);
```

```
                // printf("SUCCESS set 2nd tup item...\n");
        }
        if (a3 != " ") {
                set_tupitem(t3, a3, 2);
                // printf("SUCCESS set third tup item...\n");
        }
        return tup;
}

int set_tupitem(char* type, void* item, int pos) {
        if (type == "py") {
                PyTuple_SetItem(tup, pos, item);
                // printf("set item[%d] as py obj\n", pos);
        }
        else if (type == "str") {
                PyTuple_SetItem(tup, pos, PyDict_GetItemString(pDict, item));
                // printf("set item[%d] as str obj\n", pos);
        }
        else if (type == "int") {
                PyTuple_SetItem(tup, pos, Py_BuildValue("i", item));
                // printf("set item[%d] as int obj\n", pos);
        }
        else {
                printf("ERROR Setting item %s in pos %d..\n", item, pos);
                return 0;
        }
        return 1;
        PyErr_Print();
}

PyObject* call_pyfunc()
{
        if (PyCallable_Check(pFunc))
           {
                PyErr_Print();
                       presult = PyObject_CallObject(pFunc,tup);
```

```
                PyErr_Print();
            } else
            {
                        PyErr_Print();
            }
        return presult;
}



int main (void) {
        /* Set PYTHONPATH TO working directory */
        setenv("PYTHONPATH",".",1);

         /* Initialize the Python Interpreter */
        Py_Initialize();

        /* Prep Python */
        pName = PyString_FromString((char*)"sim");
        pModule = PyImport_Import(pName);
        pDict = PyModule_GetDict(pModule);

        /* Init hashtable for python objects */
        hashtable = ht_create( 128 );

        return yyparse ( );
}

void yyerror (const char *s) {fprintf (stderr, "%s\n", s);}
```

Glossary

Collapse The observation of a quantum state such that the state falls into state zero or one.

Deutsch-Josza The two co-authors behind the first quantum algorithm that demonstrated the power of quantum computing.

Qubit A quantum bit.

Reversible Computation The ability to go back to previous states based on actions already made by a given machine.

Superposition Equal likeliness of a quantum state being either zero or one.

Vector Space A mathematical space in which many vectors reside.

Acronyms

BQP Bounded error quantum polynomial time.

CNOT Controlled-NOT.

LEX A Lexical Analyser Generator.

QMA Quantum Merlin Arthur.

QRAM Quantum Random Access Machine.

RSA Rivest-Shamir-Adleman.

SIMD Single Instruction Multiple Data.

SQASM Simple Quantum Assembly.

TAC Three Address Code.

YACC Yet Another Compiler Compiler.

List of Figures

List of Tables

Bibliography

[Abh12] J. e. a. Abhari. *Scaffold: Quantum Programming Language*. July 2012. URL: `ftp://ftp.cs.princeton.edu/techreports/2012/934.pdf`.

[Ben+15] M. Benedetti, J. Realpe-Gómez, R. Biswas, and A. Perdomo-Ortiz. "Estimation of effective temperatures in quantum annealers for sampling applications: A case study towards deep learning." In: *ArXiv e-prints* (Oct. 2015). arXiv: `1510.07611 [quant-ph]`.

[Deu85] D. Deutsch. "Quantum theory, the Church-Turing principle and the universal quantum computer." In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*. Vol. 400. 1818. The Royal Society. 1985, pp. 97–117.

[Gos98] P. Gossett. *Quantum carry-save arithmetic*. 1998.

[Gre+13] S. Green, P. Lefanu, Lumsdaine, J. Ross, P. Selinger, and B. Valiron. *Quipper: A Scalable Quantum Programming Language*. Apr. 2013. URL: `http://arxiv.org/pdf/1304.3390v1.pdf`.

[Han+16] T. Haner, D. S. Steiger, K. Svore, and M. Troyer. "A Software Methodology for Compiling Quantum Programs." Apr. 2016.

[I05] C. I. *Quantum Architectures: qasm2circ*. Mar. 2005. URL: `http://www.media.mit.edu/quanta/qasm2circ/`.

[JNN12] R. Johansson, D. Nation, and F. Nori. *QuTiP 2: A Python framework for the dynamics of open quantum systems*. Nov. 2012. URL: `http://arxiv.org/abs/1211.6518`.

[KG15] Kaur and Goel. "Quantum Cost efficient Reversible Multiplier." In: *International Journal of ADvanced Research in Computer Engineering & Technology (IJARCET)* 4.4 (Apr. 2015), pp. 1382–1386.

[KL70] R. W. Keyes and R. Landauer. "Minimal Energy Dissipation in Logic." In: *j-IBM-JRD* 14.2 (Mar. 1970), pp. 152–157. ISSN: 0018-8646 (print), 2151-8556 (electronic).

[KMW02] D. Kielpinski, C. Monroe, and D. J. Wineland. "Architecture for a large-scale ion-trap quantum computer." In: *Nature* 417.6890 (2002), pp. 709–711.

[Kni96] E. Knill. *Conventions for quantum pseudocode.* June 1996. DOI: `10.2172/366453`.

[Ler14] e. a. Leroy X. *The OCaml system release 4.02.* Sept. 2014. URL: `http://caml.inria.fr/pub/docs/manual-ocaml/`.

[Mar+11] M. Mariantoni, H. Wang, T. Yamamoto, M. Neeley, R. C. Bialczak, Y. Chen, M. Lenander, E. Lucero, A. O'connell, D. Sank, et al. "Implementing the quantum von Neumann architecture with superconducting circuits." In: *Science* 334.6052 (2011), pp. 61–65.

[McC60] J. McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* Apr. 1960. URL: `http://www-formal.stanford.edu/jmc/recursive.pdf`.

[MG03] Mavalvala and Grektak. *8.03 Physics III.* `http://ocw.mit.edu`. [Online; accessed 16-April-2016]. 2003.

[Rie+15] E. G. Rieffel, D. Venturelli, B. O'Gorman, M. B. Do, E. M. Prystay, and V. N. Smelyanskiy. "A case study in programming a quantum annealer for hard operational planning problems." In: *Quantum Information Processing* 14 (Jan. 2015), pp. 1–36. DOI: `10.1007/s11128-014-0892-x`. arXiv: `1407.2887 [quant-ph]`.

[Sho99] P. W. Shor. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." In: *SIAM review* 41.2 (1999), pp. 303–332.

[Tay+05] J. Taylor, H.-A. Engel, W. Dür, A. Yacoby, C. Marcus, P. Zoller, and M. Lukin. "Fault-tolerant architecture for quantum computation using electrically controlled semiconductor spins." In: *Nature Physics* 1.3 (2005), pp. 177–183.

[TK15] I. Trummer and C. Koch. "Multiple Query Optimization on the D-Wave 2X Adiabatic Quantum Computer." In: *CoRR* abs/1510.06437 (2015).

[Ton04] A. van Tonder. "A Lambda Calculus for Quantum Computation." In: *SIAM Journal on Computing* 33.5 (2004), pp. 1109–1135. DOI: `10.1137/S0097539703432165`. eprint: `http://dx.doi.org/10.1137/S0097539703432165`.

[VD02] G. Van Rossum and F. L. Drake Jr. “Python/C API reference manual.” In: *Python Software Foundation* (2002).

[VGH11] F. Viamontes, L. Garcia J.and Markov, and P. Hayes. *QuIDDPro: User's Guide*. June 2011. URL: `http://vlsicad.eecs.umich.edu/Quantum/qp/qp_manual_3.8.pdf`.

[VMR15] D. Venturelli, D. J. J. Marchand, and G. Rojo. “Quantum Annealing Implementation of Job-Shop Scheduling.” In: *ArXiv e-prints* (June 2015). arXiv: `1506.08479 [quant-ph]`.

[Wat16] R. Watkins. *SQASM: Simple Quantum Assembler*. Apr. 2016. URL: `http://github.com/watkins/sqasm`.