Resources Estimation for Shor's Semiprime Integer Factorization Algorithm in Multiple Physical Platforms

Cesar Zaragoza Cortes

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2022

Reading Committee:
Boris Blinov, Chair
Anna Goussiou

Program Authorized to Offer Degree:
Physics

University of Washington

Abstract

Resources Estimation for Shor's Semiprime Integer Factorization Algorithm in Multiple Physical Platforms

Cesar Zaragoza Cortes

Chair of the Supervisory Committee:
Associate Professor Boris Blinov
Department of Physics

An important task in the development of quantum computing technologies is to determine the resources needed to execute a particular algorithm in a device with certain characteristics. Resources estimation allows not only to determine whether it is possible to execute an algorithm in a current Noisy Intermediate-Scale Quantum (NISQ) device, but also to experiment with the parameters of quantum hardware to find out how much it would have to scale to achieve quantum advantage. For this thesis, we use the Q# quantum programming language to implement Shor's semiprime integer factorization algorithm, leverage the infrastructure built around Q# to estimate the resources required to run the algorithm on trapped-ion and superconducting quantum hardware platforms, and do a comparison between the two physical platforms based on the results.

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

Algorithms designed for quantum computers have the potential to solve some problems that cannot be efficiently solved by algorithms designed for classical computers. However, estimating how much resources are needed to execute a quantum algorithm that outperforms a classical one is a difficult task. There are many quantum programming languages and tools built around them such as Q#[16], Qiskit[17] and Cirq[11] that allow execution of quantum algorithms on simulators but out-of-the-box options to estimate resources are limited to the logical level or not existent.

1.1 The Purpose of This Thesis

This thesis aims to estimate the resources required at the physical level to run Shor's semiprime integer factorization algorithm for trapped-ion and superconducting quantum hardware platforms. To do this, we will extend the simulators infrastructure built around Q# to calculate the maximum number of physical qubits, the total number of physical gates, and the maximum runtime required to execute a particular quantum algorithm.

Furthermore, we will also analyze the effects of errors introduced by the physical gates, analyze the feasibility of obtaining reliable results without implementing fault-tolerance, and estimate the cost of running the algorithm using error-corrected qubits and fault-tolerant gates.

In order to make this thesis more accesible to people from different backgrounds, we dedicate the next few chapters to provide a brief overview of the basics of quantum computing, quantum error correction, and Shor's semiprime integer factorization algorithm.

Chapter 2

BASICS OF QUANTUM COMPUTING

2.1 Dirac Notation

Also known as bra-ket notation, Dirac notation provides a convenient way of expressing the vectors used in quantum mechanics.

Dirac notation defines two elements:

- The *ket* ($|\psi\rangle$), which denotes a column vector in a complex vector space that represents a quantum state.

$$|\psi\rangle = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \cdots \\ \psi_n \end{bmatrix}$$

- The *bra* ($\langle\psi|$), which denotes a row vector that is the conjugate transpose, or adjoint, of a corresponding *ket* ($|v\rangle$).

$$\langle\psi| = \begin{bmatrix} \psi_1^* & \psi_2^* & \cdots & \psi_n^* \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \cdots \\ \psi_n \end{bmatrix}^\dagger$$

An *operator* $\mathbf{A}$, represented by a $n \times n$ matrix, acting on a *ket* $|\psi\rangle$ produces another *ket* $|\psi'\rangle$ such that the produced *ket* can be computed by matrix multiplication:

$$|\psi'\rangle = \mathbf{A}\,|\psi\rangle = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} = \begin{bmatrix} A_{11}\psi_1 + A_{12}\psi_2 \\ A_{21}\psi_1 + A_{22}\psi_2 \end{bmatrix}$$

The *inner product* is denoted as a bra-ket pair $\langle\phi|\psi\rangle$ and represents the probability amplitude that a quantum state ψ would be subsequently found in state ϕ:

$$\langle\phi|\psi\rangle = \begin{bmatrix} \phi_1^* & \phi_2^* \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} = \phi_1^*\psi_1 + \phi_2^*\psi_2$$

This notation also provides a way to describe the state vector of n uncorrelated quantum states, the *tensor product*:

$$|\phi\rangle \otimes |\psi\rangle = |\phi\rangle\,|\psi\rangle = |\phi\psi\rangle = \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} \otimes \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} = \begin{bmatrix} \phi_1\psi_1 \\ \phi_1\psi_2 \\ \phi_2\psi_1 \\ \phi_2\psi_2 \end{bmatrix}$$

Note that $|\psi\rangle^{\otimes n}$ represents the tensor product of n $|\psi\rangle$ quantum states:

$$|\psi\rangle^{\otimes n} = |\psi\rangle \otimes \cdots \otimes |\psi\rangle = |\psi\rangle \cdots |\psi\rangle = |\psi \cdots \psi\rangle = \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix}$$

2.2 Quantum Systems

Quantum mechanics is a mathematical framework for the development of physical theories[10]. On its own quantum mechanics doesn't tell you what laws a physical system must obey, but it does provide a mathematical and conceptual framework for the development of such laws. The postulates described next provide a connection between the physical world and the mathematical formalism of quantum mechanics.

Postulate 1: The state of an isolated physical system is fully described by a unit vector $|\psi\rangle$ in Hilbert space.

Postulate 2: The evolution of a closed quantum system is described by a unitary transformation. This means that the state $|\psi\rangle$ at time t_1 is related to the state $|\psi'\rangle$ at time t_2 by a unitary operator U which depends only on the times t_1 and t_2:

$$|\psi'\rangle = U(t_1, t_2)\,|\psi\rangle$$

Postulate 3: The state space of a composite physical system is the tensor product of the state spaces of the component physical systems.

Postulate 4: Quantum measurements are described by a collection of M_m of measurement operators.

2.3 Qubits

In classical information theory and classical computing, a bit is the fundamental building block. Analogously, in quantum information theory and quantum computing, a quantum bit or qubit is the fundamental building block.

Physically, a quibit is a two-level quantum-mechanical system. The polarization of a single photon and the spin of the electron are examples of such systems.

Mathematically, a qubit is a linear combination of states $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ where α and β are complex numbers known as amplitudes, and $|0\rangle$ and $|1\rangle$ are the computational basis states.

When a qubit is measured, the result is either $|0\rangle$ with probability $|\alpha|^2$ or $|1\rangle$ with probability $|\beta|^2$. Since the probabilities must sum to one, the qubit's state is normalized:

$$|\alpha|^2 + |\beta|^2 = 1$$

The computational basis states, which are analogous to the two values (0 and 1) that a classical bit may take, form an orthonormal basis represented by the following vectors:

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

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

Using the previous definitions, we can see a qubit as a unit vector in two-dimensional complex vector space:

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

2.4 Superposition

The principle of quantum superposition states that the most general state of a quantum-mechanical system is a linear combination of all distinct valid quantum states. A qubit is an example of a quantum superposition of the basis states $|0\rangle$ and $|1\rangle$.

A concrete example of a qubit in superposition that has the same probability of being measured as $|0\rangle$ or $|1\rangle$ is the following:

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

2.5 Quantum Logic Gates

In classical digital circuits, logic gates are the building blocks. Analogously, in the quantum circuit model of computation, quantum logic gates are the building blocks of quantum algorithms. Quantum gates act on qubits, transform them in different ways, and can be applied sequentially to perform complex quantum computations.

Quantum gates are unitary operators represented as $2^n \times 2^n$ unitary matrices where n is the number of qubits the gate operates on. Unitary matrices are complex square matrices $\mathbf{U}$ which have the property that its conjugate transpose or adjoint $\mathbf{U}^\dagger$ is also its inverse $\mathbf{U^{-1}}$:

$$\mathbf{U}^\dagger\mathbf{U} = \mathbf{U}\mathbf{U}^\dagger = \mathbf{U}\mathbf{U^{-1}} = \mathbf{I}$$

A quantum gate is applied to a qubit system by multiplying the gate's matrix representation by the qubits' state vector. This operation transforms the qubit system:

$$|\psi_1\rangle = \mathbf{U}\,|\psi_\mathbf{0}\rangle$$

Applying a sequence of quantum gates is equivalent to performing a series of these multiplications. For example, applying gate $\mathbf{U_a}$ followed by a gate $\mathbf{U_b}$ to a state vector $|\psi\rangle$ is represented by the follwing expression where the gates closest to the state vector are applied first:

$$\mathbf{U_b}\mathbf{U_a}\,|\psi\rangle$$

Since matrix multiplication is associative, multiplying $\mathbf{U_a}$ by $\mathbf{U_b}$ produces a compund gate $\mathbf{U_b}\mathbf{U_a}$ that is equivalent to applying $\mathbf{U_a}$ followed by $\mathbf{U_b}$:

$$\mathbf{U_b}\mathbf{U_a}\left|\psi\right\rangle = \mathbf{U_b}(\mathbf{U_a}\left|\psi\right\rangle) = (\mathbf{U_b}\mathbf{U_a})\left|\psi\right\rangle$$

Note that all quantum gates are reversible since they are represented by unitary matrices. This means that for any gate, another gate exists that reverts the gate's transformation on a state vector:

$$\left|\psi\right\rangle = \mathbf{U}^{\dagger}(\mathbf{U}\left|\psi\right\rangle) = \mathbf{U}^{\dagger}\mathbf{U}\left|\psi\right\rangle = \mathbf{I}\left|\psi\right\rangle$$

2.6 Single-Qubit Gates

Single-qubit gates are represented by 2×2 unitary matrices. Applying a gate to a matrix can be done by multiplying the gate's matrix by the qubit's state vector.

Next, some of the most commonly used gates.

Pauli-X gate:

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

Pauli-Y gate:

$$Y = \begin{bmatrix} 0 & -\imath \\ \imath & 0 \end{bmatrix}$$

Pauli-Z gate:

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

Hadamard gate:

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

RX gate:

$$R_x(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -\imath sin\frac{\theta}{2} \\ -\imath sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$$

RY gate:

$$R_y(\theta) = \begin{bmatrix} cos\frac{\theta}{2} & -sin\frac{\theta}{2} \\ sin\frac{\theta}{2} & cos\frac{\theta}{2} \end{bmatrix}$$

RZ gate:

$$R_z(\theta) = \begin{bmatrix} e^{-\imath\frac{\theta}{2}} & 0 \\ 0 & e^{\imath\frac{\theta}{2}} \end{bmatrix}$$

R1 gate:

$$R_1(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{\imath\theta} \end{bmatrix}$$

2.7 Multi-Qubit Gates

Multi-qubit gates are represented by $2^N \times 2^N$ matrices where N is the number of qubits the gate operates on.

Of special importance is the CNOT gate which is a two-qubit gate where the first qubit is referred to as the control qubit and the second qubit as the target qubit. CNOT acts as a conditional gate, if the control qubit is in state $|1\rangle$, it applies the Pauli X gate to the target qubit, otherwise it does nothing. The main use of this gate is to prepare entangled states.

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

Chapter 3
BASICS OF QUANTUM ERROR CORRECTION

This chapter explains the basic concepts to understand how quantum computations can be performed reliably in the presence of noise. Given the fragility of coherent quantum systems, quantum error correction and fault-tolerant quantum computation are fundamental arbitrarily large quantum algorithms operating on many qubits.

3.1 Noise and Error Correction

Noise can introduce errors in data while it is being processed, sent through a channel and/or stored in a medium. A common error that can be introduced by noise is a bit flip (e.g. $0 \rightarrow 1$, $1 \rightarrow 0$). The task of error correction is to detect when an error has occurred and correct it.

In classical error correction, coding based on data-copying is extensively used. The key idea is to protect data against the effects of noise by adding redundancy. For example, a classical repetition code can encode a logical bit using three physical bits:

$$0 \rightarrow 000$$

$$1 \rightarrow 111$$

The problem is that classical error correction techniques cannot be directly applied to qubits because of the following reasons:

- It is impossible to create an independent and identical copy of a qubit in an arbitrary unknown state. This is known as the no-cloning theorem of quantum mechanics and it has the consequence that qubits cannot be protected from errors by simply making multiple copies[19].

- Measurement destroys quantum information, implying that we cannot measure a qubit to decide which correcting action to perform.
- In addition to bit flip errors ($|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow |\psi\rangle = \alpha|1\rangle + \beta|0\rangle$), qubits are also susceptible to phase flip errors ($|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow |\psi\rangle = \alpha|0\rangle - \beta|1\rangle$) that have no classical analogous. Therefore, quantum error correction must be able to simultaneously correct for both.
- Errors in quantum information are continuous. This means that qubits, in the presence of noise, experience angular shifts rather than full bit or phase flips.

In order to protect qubits against the effects of noise, we can use the following process:

1. Encode: multiple physical qubits are used to encode a two-level quantum state into a logical qubit.
2. Manipulation: operations are performed using the logical qubit (here's where errors can be introduced).
3. Syndrome detection: measurements are performed that indicate what error, if any, ocurred.
4. Recovery: the value of the syndrome is used to select the procedure to use to correct the error.
5. Decode: logical qubit is decoded before final measurements are made.

3.2 Bit Flip Code

To make it possible to correct bit flip errors, we can use three physical qubits to encode each logical qubit:

$$|0_L\rangle = |000\rangle$$

$$|1_L\rangle = |111\rangle$$

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \underrightarrow{encode} |\psi_L\rangle = \alpha|0_L\rangle + \beta|1_L\rangle$$

The notation $|0_L\rangle$ and $|1_L\rangle$ indicates that these are the logical $|0\rangle$ and logical $|1\rangle$ states, and not the physical $|0\rangle$ and physical $|1\rangle$ states.

A circuit that performs this encoding is illustrated in figure 3.1:

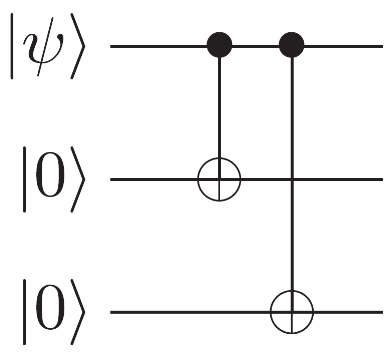

Figure 3.1: Bit flip encoding circuit [10]

After the logical qubit has been manipulated, syndrome detection and recovery is done. There are four bit flip posibilities at this stage:

- No error: $\alpha\,|000\rangle + \beta\,|111\rangle$
- First qubit flipped: $\alpha\,|100\rangle + \beta\,|011\rangle$
- Second qubit flipped: $\alpha\,|010\rangle + \beta\,|101\rangle$
- Third qubit flipped: $\alpha\,|001\rangle + \beta\,|110\rangle$

For each one of these posibilities, the recovery operation is clear. The circuit shown in figure 3.2 detects the error using two auxiliary qubits that are entangled with the logical qubit such that measurements on those qubits yield two classical bits of information whose values map to a specific recovery operation that is then applied. Note that the X gate is used as the recovery operation.

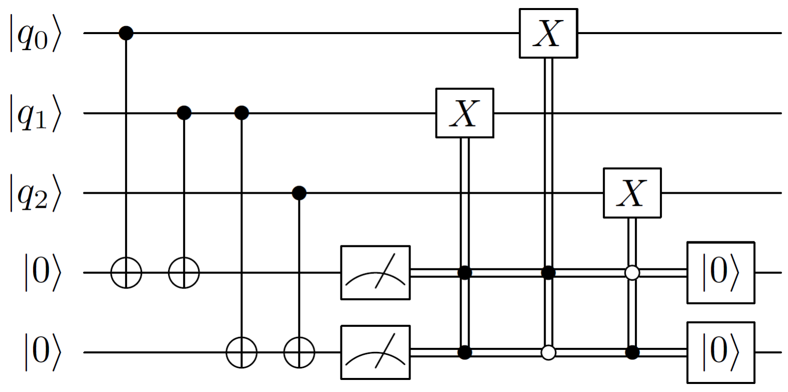

Figure 3.2: Bit flip detection and recovery circuit [18]

3.3 *Phase Flip Code*

Phase flip errors can similarly be corrected by using three physical qubits to encode a logical qubit. Since a complete phase flip error in the $|+\rangle$ and $|-\rangle$ basis switches between these states, logical qubits are the following:

$$|0_L\rangle = |+++\rangle$$

$$|1_L\rangle = |---\rangle$$

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \xrightarrow{encode} |\psi_L\rangle = \alpha\,|0_L\rangle + \beta\,|1_L\rangle$$

A circuit that performs this encoding is illustrated in figure 3.3:

After the logical qubit has been manipulated, syndrome detection and recovery is done. Like in the bit flip case, there are four posibilities at this stage:

- No error: $\alpha\,|+++\rangle + \beta\,|---\rangle$
- First qubit flipped: $\alpha\,|-++\rangle + \beta\,|+--\rangle$
- Second qubit flipped: $\alpha\,|+-+\rangle + \beta\,|-+-\rangle$
- Third qubit flipped: $\alpha\,|++-\rangle + \beta\,|--+\rangle$

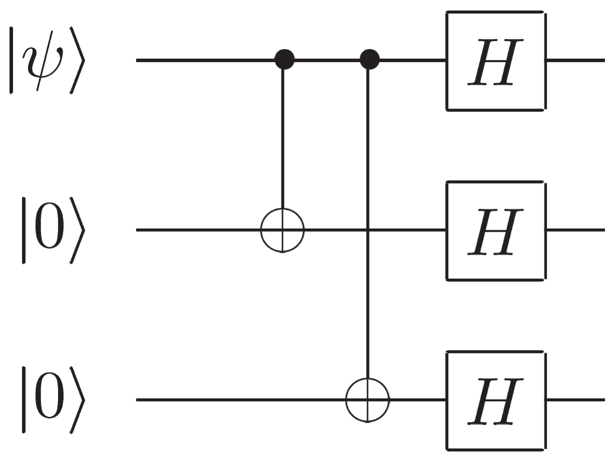

Figure 3.3: Phase flip encoding circuit [10]

The circuit shown in figure 3.4 detects the error using two auxiliary qubits that are entangled with the logical qubit such that measurements on those qubits yield two classical bits of information whose values map to a specific recovery operation that is then applied. In this case, instead of using the X gate to correct errors, the Z gate is used as the recovery operation for the qubit that was flipped.

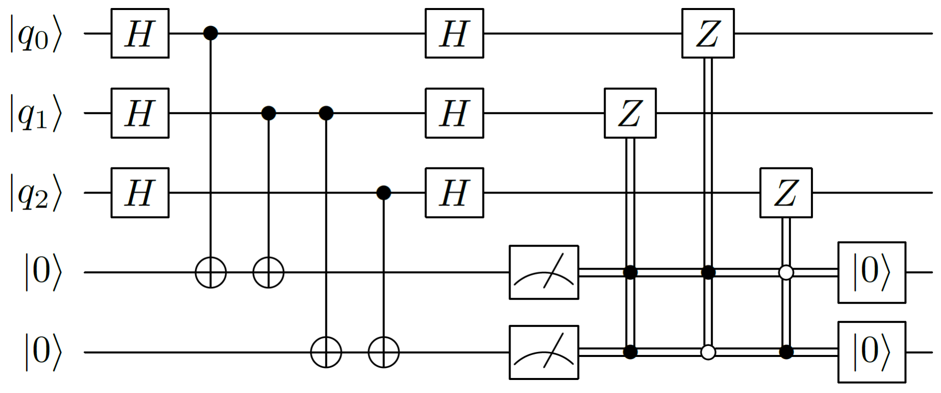

Figure 3.4: Bit flip detection and recovery circuit [18]

3.4 Shor Code

There is a quantum code which can protect against the effects of an arbitrary error on a single qubit knwon as the Shor code. The code is a combination of three qubit phase flip and bit flip codes. First, we encode the qubit using the phase flip code such that $|0\rangle \rightarrow |+++\rangle$ and $|1\rangle \rightarrow |---\rangle$. Next, we encode each one of these qubits using the three qubit bit flip code such that $|+\rangle \rightarrow \frac{|000\rangle+|111\rangle}{\sqrt{2}}$ and $|-\rangle \rightarrow \frac{|000\rangle-|111\rangle}{\sqrt{2}}$. This results in the following logical qubits:

$$|0_L\rangle = \frac{(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)(|000\rangle + |111\rangle)}{2\sqrt{2}}$$

$$|1_L\rangle = \frac{(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)(|000\rangle - |111\rangle)}{2\sqrt{2}}$$

A circuit that performs this encoding is illustrated in figure 3.5:

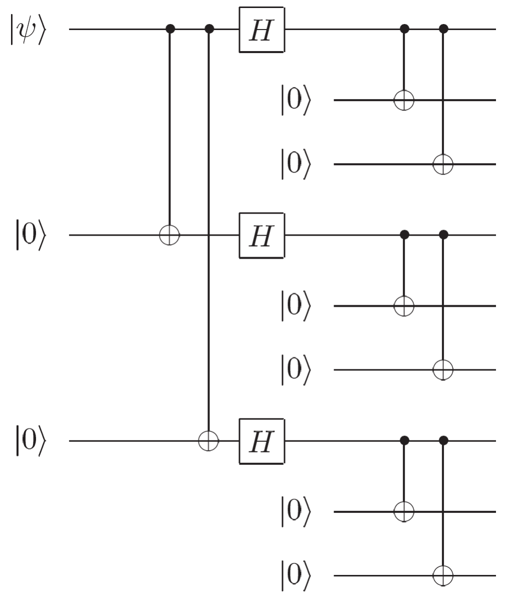

Figure 3.5: Phase flip encoding circuit [10]

Syndrome detection and recovery for the Shor code is performed by combining bit flip and phase flip detection and recovery. Figure 3.6 shows bit flip detection and recovery. To do the same for phase flips we would just have to use the previously shown phase flip detection and recovery circuit on the first qubit of each bit flip encoded block.

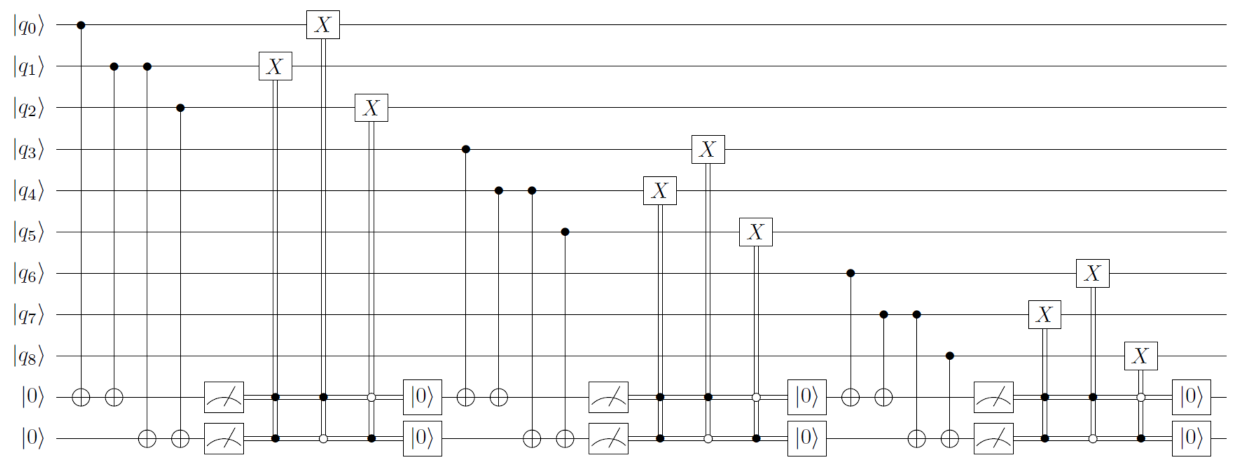

Figure 3.6: Bit flip detection and recovery circuit on a Shor encoded qubit [18]

Note that the Shor code corrects all quantum errors assuming each triplet experiences at most one bit flip error per correction cycle, and at most one triplet experiences a phase flip error per correction cycle.

3.5 Quantum Error-Correction Without Measurement

We have described quantum error-correction as a two stage process: a syndrome detection step that uses quantum measurement followed by a recovery step that applies unitary operations based on the results of the measurement. It is posible to perform error-correction using only unitary operations and auxiliary qubits prepared in standard states.

For example, figure 3.7 shows a delayed measurement circuits that perform syndrome detection and error recovery for bit flip errors:

The advantage this provides is that for some real-world quantum systems it is very difficult to apply different unitary operations based on quantum measurements so an alternate procedure is needed.

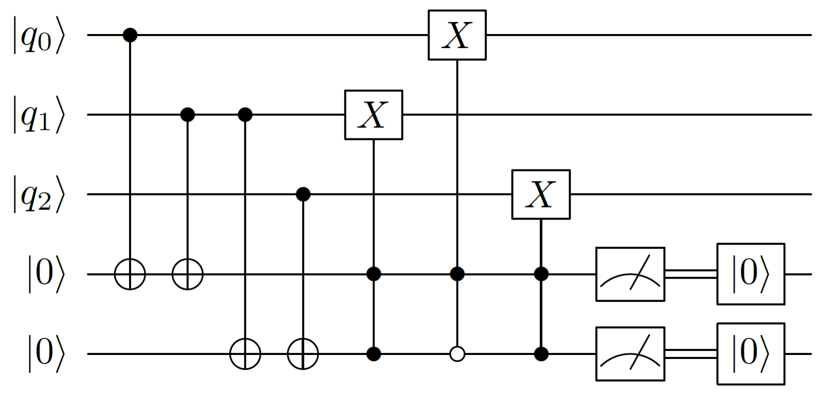

Figure 3.7: Bit flip detection and recovery circuit using delayed measurement [18]

3.6 Fault-Tolerant Quantum Computation

One of the most useful applications of quantum error correction is the protection of quantum information as it dynamically undergoes computation. An arbitrary good quantum computation can be achieved provided only that the error probability per gate is below a certain constant threshold. A quantum computer that accumulates error slow enough that error can be corrected in called fault-tolerant.

The basic idea behind fault-tolerant quantum computation is to perform computations directly on encoded logical qubits such that decoding is never required.

Figure 3.8 shows a circuit using fault-tolerant logical operations. In this specific case, seven physical qubits are being used to encode and error-correct each logical qubit. One thing to note is that the reason the second error-correction step performed in the second qubit is that simply storing qubits for a period of time introduces errors and should periodically be error-corrected in order to prevent acumulation of errors.

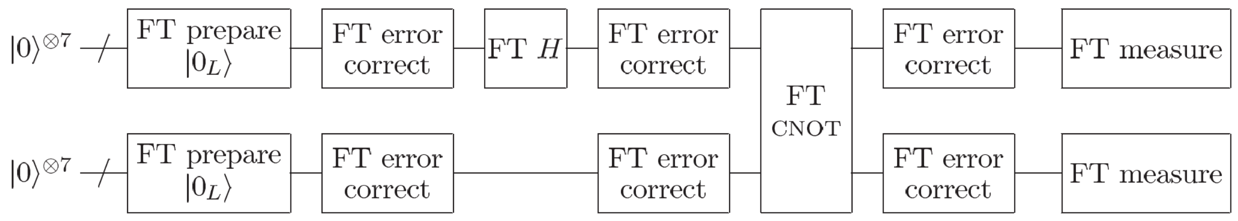

Figure 3.8: Example of a fault-tolerant circuit [10]

Chapter 4
SHOR'S SEMIPRIME INTEGER FACTORIZATION ALGORITHM

Shor's algorithm is a polynomial-time quantum algorithm for semiprime integer factorization. In comparison, the time complexity of the most efficient known classical factoring algorithm is superpolynomial.

We chose Shor's algorithm because it is one of the most significant quantum algorithms due to its exponential improvement on efficiency compared to its classical counterparts, and because both the number of qubits and the number of quantum operations required to run it are proportional to the input size. The number of operations is particullarly relevant since it makes the errors introduced by the physical gates an important consideration.

4.1 Algorithm

The problem that Shor's algorithm solves is the following: given a semiprime integer N, find its two prime factor p and q.

Shor's algorithm combines both classical and quantum computations, and the procedure to perform it is the following:

1. Pick a random integer $1 < a < N$.

2. Check whether a is a factor by determining whether a and N are comprime. If they are, we can compute the factors. Otherwise continue with the rest of the algorithm.

3. Use a period-finding quantum subroutine to find the period r of the function $f(x) = a^x \; mod \; N$.

4. If r is odd, then go back to step 1. If r is even, go to the next step.

5. If $a^{\frac{r}{2}} \equiv -1 \ mod\ N$, then go back to step 1.

6. Either $a^{\frac{r}{2}} - 1$ or $a^{\frac{r}{2}} + 1$ shares a factor with N.

Figure 4.1 shows the circuit that implements the quantum subroutine in Shor's algorithm. The quantum circuits Ua^{2^k} used for this algorithm are custom designed for each value of N and a.

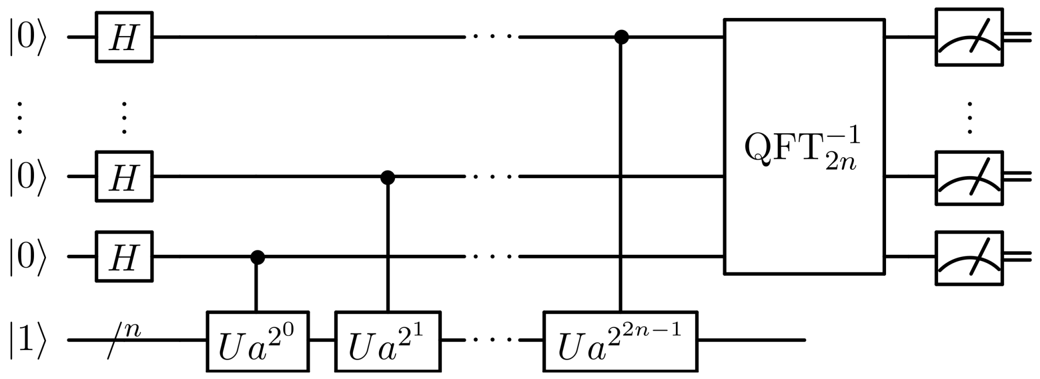

Figure 4.1: Period finding quantum subroutine [6]

Stephane Beauregard[8] provides a detailed description on how the Ua^{2^k} quantum circuits are built.

Chapter 5
ANALYSIS FRAMEWORK

The strategy we use to analyze Shor's integer factorization algorithm is very similar to the one proposed by Soeken et al.[15]. The overview of the process is the following:

1. Implement the quantum algorithm using a high-level programming language.

2. Verify the correctness of the implementation by executing the algorithm in a full state simulator.

3. Use, extend and build tools that can use the high-level algorithm implementation to estimate the amount of logical resources the algorithm requires depeding on its input size.

    (a) As part of logical resources estimation, consider the effects of introducing error-corrected qubits and fault-tolerant quantum gates on the amount of required resources.

4. For each physical platform do the following:

    (a) Create a tool that uses platform specific parameters to calculate the amount of physical qubits, physical gates, and runtime required to run the algorithm.

    (b) Analyze the data produced by the tool to determine the following:

        i. The maximum input size for the algorithm to run on a real NISQ device.

        ii. The characteristics that a device should have to run the algorithm for a specific input size, and determine what would be the runtime.

5. Compare the results obtained for each physical platform.

5.1 Layered Architecture for Quantum Computing

As a basis for the analysis, we will use a layered architecture model like the one proposed by N. Cody Jones et al.[14]. Figure 5.1 shows the five different layers of the architecture.

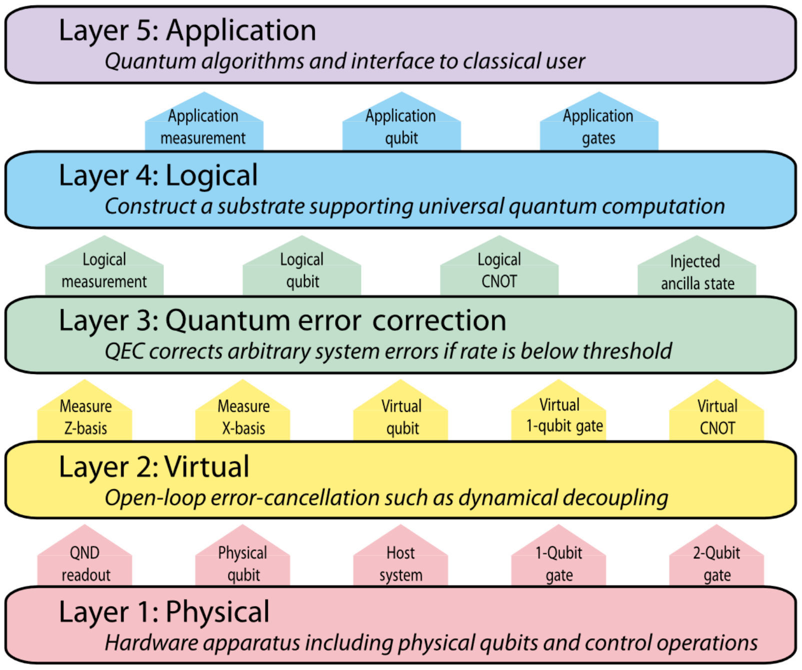
Layer 5: Application
Quantum algorithms and interface to classical user
Application measurement
Application qubit
Application gates
Layer 4: Logical
Construct a substrate supporting universal quantum computation
Logical measurement
Logical qubit
Logical CNOT
Injected ancilla state
Layer 3: Quantum error correction
QEC corrects arbitrary system errors if rate is below threshold
Measure Z-basis
Measure X-basis
Virtual qubit
Virtual 1-qubit gate
Virtual CNOT
Layer 2: Virtual
Open-loop error-cancellation such as dynamical decoupling
QND readout
Physical qubit
Host system
1-Qubit gate
2-Qubit gate
Layer 1: Physical
Hardware apparatus including physical qubits and control operations

Figure 5.1: Layered architecture for quantum computing [14]

Briefly describing each layer:

1. Physical: Consists of the physical qubits and the apparatus needed to manipulate them. At this layer, the gates that are native to the physical platform are the ones used for qubit manipulation.

2. Virtual: Provides the interface to interact with the hardware such that the specifics of controlling it are abstracted away.

3. Quantum error correction: Implements logical qubit encoding and fault-tolerant operations.

4. Logical: Tools that allow the development of quantum algorithms at a higher level of abstraction. This layer exposes higher level gates that are decomposed into fault-tolerant sequences of native gates. Programming languages, frameworks and compilers belong to this layer.

5. Application: Quantum algorithms like Shor's or Grover's implemented on top of the logical layer.

The main advatage of a layered architecture like this one is modularity. Since each layer encapsulates its functionalities, the internal implementation of each layer can be changed as requirements and technology evolve. For example, the quantum error correcting code in the third layer can be changed depending on what works best for a specific physical platform.

5.2 Q# and the Quantum Development Kit (QDK)

Microsoft's Quantum Development Kit (QDK)[5] is a set of tools for quantum application development. It includes Q#, a high-level programming language, different quantum simulators and libraries that are useful for development and verification of quantum algorithms.

Quantum simulators are software programs that run on classical computers and act as the target machine for a Q# program, making it possible to run and test quantum programs in an environment that handles qubits and operations that act on them[?]. Each type of quantum simulator can provide different implementations of quantum primitive operations such as Hadamard, Pauli-X, Pauli-Y, Pauli-Z and CNOT. The ones we take advatage of are:

- Full-state simulator: Runs quantum algorithms by tracking the full quantum state vector. It is limited to 30 qubits.

- Resources estimator: Performs a top level analysis of the resources needed to run a quantum algorithm.

- Trace-based resource estimator: Runs advanced analysis of resources consumptions for the algorithm's entire call-graph.

Using Q# and the QDK allows the algorithm implementation to remain constant even when the way it is executed changes, which is incredibly useful to verify the correctness of the implementation and to have confidence on the results it yields.

Additionally, the QDK supports the implementation of custom simulators that can be used to run Q# programs. We leverage this capability and implement custom simulators that make calculations based on additional parameters.

5.3 Command line application for verification and resources estimation

In order to facilitate the verification of the correctness of the algorithm implementation and the calculation of resources making different assumptions, we developed a command line application that integrates the quantum algorithm implementation with both out-of-the-box QDK simulators and custom simulators. This way we can do the analysis from a single place.

Source code of a working version can be found in GitHub.

Chapter 6
SHOR'S ALGORITHM LOGICAL ANALYSIS

6.1 Q# Implementation

In order to take advantage of the tools built around Q#, we use a modified implementation of Shor's algorithm found in Microsoft's quantum samples GitHub repository[4].

The *FactorSemiprimeInteger* (6.1) operation represents the top-level implementation of Shor's algorithm. The main part of the algorithm is within a cycle because it depends on the coprime guess whether the prime factors can be found. The *EstimatePeriod* operation is the one that contains the quantum logic, before and after it some classical computation is done.

Listing 6.1: Top level implementation of Shor's algorithm in Q#

```
operation FactorSemiprimeInteger(N : Int) : (Int, Int) {

    // Check the most trivial case where N is pair.
    if (N % 2 == 0) {
        return (2, N / 2);
    }

    mutable factors = (1, 1);
    mutable foundFactors = false;

    // Keep trying until we find the factors.
    repeat {
        // Start by guessing a coprime to N.
        let coprimeGuess = DrawRandomInt(1, N - 1);

        // If the guess number is a coprime, use a quantum algorithm for peri
        // Otherwise, the GCD between N and the coprime guess number is one o
        if (IsCoprimeI(N, coprimeGuess)) {
            let period = EstimatePeriod(N, coprimeGuess);
            set (foundFactors, factors) = CalculateFactorsFromPeriod(N, coprin
        } else {
            let gcd = GreatestCommonDivisorI(N, coprimeGuess);
            set (foundFactors, factors) = (true, (gcd, N / gcd));
        }
    }
    until foundFactors

    return factors;
}
```

The *EstimatePeriod* (6.2) operation implements a cycle that performs quantum period estimation until a viable period is found.

Listing 6.2: Q# implementation of the EstimatePeriod operation

```
operation EstimatePeriod(N : Int, a : Int) : Int {
    mutable period = 1;
    repeat {
        let (instanceSucceeded, instancePeriod) = EstimatePeriodInstance(N, a
        if (instanceSucceeded) {
            set period = instancePeriod;
        }
    }
    until (ExpModI(a, period, N) == 1)
    fixup {
        Message("Period estimate failed.");
    }
    return period;
}
```

The *EstimatePeriodInstance* (6.3) operation is the one that actually runs on a quantum device. It allocates all the qubits that will be needed to calculate the period depeding on the number of bits that are needed to represent the integer to factor. It then uses an oracle that implements the function $|x\rangle \rightarrow \left|(a^k x) mod N\right\rangle$.

Listing 6.3: Q# implementation of the EstimatePeriodInstance operation

```
operation EstimatePeriodInstance(N : Int, a : Int) : (Bool, Int) {

    // Prepare eigenstate register.
    let bitSize = BitSizeI(N);
    use eigenstateRegister = Qubit[bitSize];
    let eigenstateRegisterLE = LittleEndian(eigenstateRegister);
    ApplyXorInPlace(1, eigenstateRegisterLE);

    // Prepare phase register.
    let bitsPrecision = 2 * bitSize + 1;
    use phaseRegister = Qubit[bitsPrecision];
    let phaseRegisterLE = LittleEndian(phaseRegister);

    // Prepare oracle for quantum phase estimation.
    let oracle = DiscreteOracle(ApplyOrderFindingOracle(N, a, _, _));
    // Execute quantum phase estimation.
    QuantumPhaseEstimation(oracle, eigenstateRegisterLE!, LittleEndianAsBigEn
    let phaseEstimate = MeasureInteger(phaseRegisterLE);

    // Reset qubit registers
    ResetAll(eigenstateRegister);

    // Return period calculation based on estimated phase.
    return CalculatePeriodFromPhaseEstimate(N, bitsPrecision, phaseEstimate);
}
```

The *ApplyOrderFindingOracle* (6.4) operation is the one that implements and applies the oracle function to the target register.

Listing 6.4: Q# implementation of the ApplyOrderFindingOracle operation

```
operation ApplyOrderFindingOracle(
    N : Int, a : Int, power : Int, target : Qubit[])
: Unit is Adj + Ctl {
    MultiplyByModularInteger(ExpModI(a, power, N), N, LittleEndian(target));
}
```

6.2 Logical Resources Estimation

One of the smallest semiprime integers to factor is 15. The number of qubits and unitary operations needed to factor this number using Shor's algorithm is small enough that we can simulate it using the full state simulator. Running a simulation through the console application we built yields the result in listing 6.5. The important thing to note here is that 4 is used as coprime guess a, which is what we are going to use to estimate quantum logical resources for this particular execution.

Listing 6.5: Output of simulating Shor's algorithm on a full-state simulator

```
> dotnet run -- simulate -N 15
Console App
Factoring semiprime integer 15 using a full state simulator...
Shor's integer factorization algorithm
Starting cycle 1 using 4 as a coprime guess...
Estimating period using a quantum algorithm...
Starting estimate period cycle 1
Estimated phase: 256
Estimated period: 2
Simulation ended
Factors are 5 and 3
```

In order to estimate resources, we have to provide both the integer to factor N and a coprime guess a to the console application *estimatelogicalresources* command as shown in listing .

Listing 6.6: Output of estimating logical resources for Shor's algorithm

```
> dotnet run -- estimatelogicalresources -N 15 --generator 4
Metric          Sum     Max
CNOT            49610   49610
QubitClifford   11675   11675
R               4041    4041
Measure         13      13
T               31137   31137
Depth           20897   20897
Width           21      21
QubitCount      21      21
BorrowedWidth   0       0
```

The meaning of the reported metrics is the following:

- CNOT: The run count of CNOT operations (also known as Controlled Pauli X operations).
- QubitClifford: The run count of any single qubit Clifford and Pauli operations.
- Measure: The run count of any measurements.
- R: The run count of any single-qubit rotations, excluding T, Clifford and Pauli operations.
- T: The run count of T operations and their conjugates, including the T operations, $T_x = H.T.H$, and $T_y = Hy.T.Hy$.
- Depth: Depth of the quantum circuit run by the Q# operation. By default, the depth metric only counts T gates.
- Width: Width of the quantum circuit run by the Q# operation.
- QubitCount: The lower bound for the maximum number of qubits allocated during the run of the Q# operation.
- BorrowedWidth: The maximum number of qubits borrowed inside the Q# operation.

One of the main limitiations of using this approach to calculate resources is that qubit measurements yield each one of the computational basis states $|0\rangle$ and $|1\rangle$ with 50% probability.

Chapter 7

TRAPPED-IONS PHYSICAL PLATFORM

Trapped ions is one of the most promising approaches to the physical implementation of a quantum computer[9]. The basic requirements for a universal quantum computer, as outlined by David DiVicenzo[12], have been demostrated with trapped ions, and a few companies are using this approach to build their quantum computers (e.g. IonQ [3], Honeywell [2]).

7.1 Platform Characteristics

In this physical platform, qubits are ions confined and suspended in free space using electromagnetic fields, and its internal electronic states are used as qubit states $|0\rangle$ and $|1\rangle$. Initialization of the qubits is performed by laser manipulation. State $|0\rangle$ is a long-lived ground state and state $|1\rangle$ can be prepared by optically pumping the ion to an auxiliary state $|e\rangle_{SP}$ that rapidly decays into state $|1\rangle$ as shown in figure 7.1.

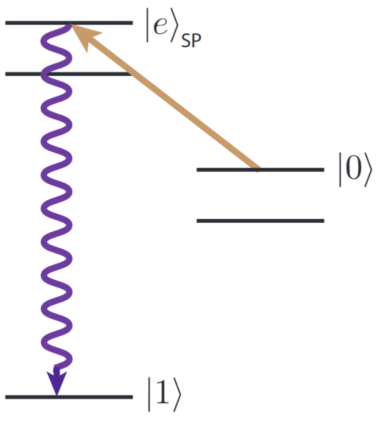

Figure 7.1: Qubit initialization [9]

As shown in figure 7.2, qubit readout is done by using a resonant laser that couples the $|1\rangle$ state to a $|e\rangle_R$ state which scatters many photons that can be collected by a detector. When the ion is in state $|0\rangle$, it does not interact with the laser so no photons are emitted and the image produced by the detector is dark.

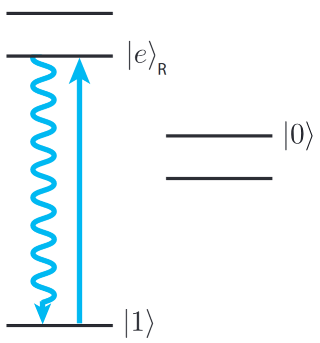

Figure 7.2: Qubit readout [9]

A universal gate set is achieved by providing two qubit manipulation mechanisms:

- Arbitrary single qubit rotations: laser or microwave drives applied to the ions allow arbitrary and high fidelity single qubit rotations to be performed.
- An entangling operation: motional modes of two or more ions are used as a bus for transfering quantum information among ions.

Single qubit gate times are in the order of a few microseconds while two qubit gate times typically take between tens and hundreds of microseconds. Ion coherence times range from 0.2 seconds in optical qubits to up to 600 seconds for hyperfine qubits. Long coherence times relative to gate times fulfill the remaining criteria for a universal quantum computer.

7.2 Native Gates

The native single qubit gate[13] in this physical platform is defined as:

$$R(\theta, \phi) = \begin{bmatrix} \cos\frac{\theta}{2} & -\imath e^{-\imath\phi}\sin\frac{\theta}{2} \\ -\imath e^{-\imath\phi}\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix}$$

Rotations about the X-axis ($R_x(\theta)$) are achieved by setting $\phi = 0$. Rotations about the Y-axis ($R_y(\theta)$) are achieved by setting $\phi = \frac{\pi}{2}$. Rotations about the Z-axis ($R_z(\theta)$) are achieved by three rotations about axes in the XY plane as shown in figure 7.3:

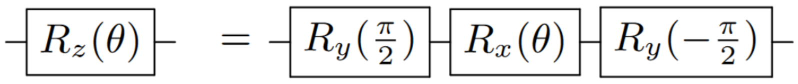

Figure 7.3: Rz gate [13] decomposition

Note that Pauli gates can be constructed from rotations around the X, Y and Z axis:

- $X = \imath R_x(\pi)$
- $Y = \imath R_y(\pi)$
- $Z = \imath R_z(\pi)$

Another widely used quantum gate is the Hadamard gate, which can be also be decomposed from rotations around the X and Y axis[1]:

$$H = R_x(\pi)R_y(\frac{\pi}{2})$$

The native two qubit entangling gate[13] in this physical platform is defined as:

$$XX(\chi) = \begin{bmatrix} \cos\chi & 0 & 0 & -\imath\sin\chi \\ 0 & \cos\chi & -\imath\sin\chi & 0 \\ 0 & -\imath\sin\chi & cos\chi & 0 \\ -\imath\sin\chi & 0 & 0 & cos\chi \end{bmatrix}$$

The CNOT gate, which is more commonly used as an entangling gate when describing quantum algorithms, can be constructed using the $XX(\chi)$ and $R(\theta, \phi)$ gates as shown in figure 7.4:

Using the results demonstrated by C. J. Ballance et al. [7], the gate times and fidelities in this physical platform are the following:

- Single qubit arbitrary rotation gate ($R(\theta, \phi)$):

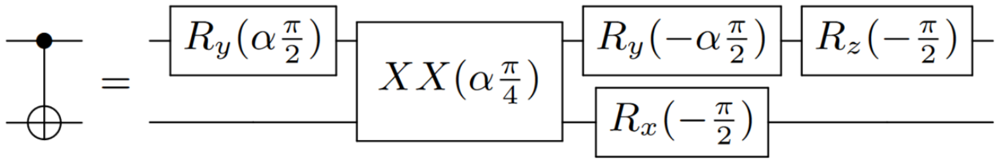
=
$R_y(\alpha\frac{\pi}{2})$
$XX(\alpha\frac{\pi}{4})$
$R_y(-\alpha\frac{\pi}{2})$
$R_z(-\frac{\pi}{2})$
$R_x(-\frac{\pi}{2})$

Figure 7.4: CNOT gate [13] decomposition where $\alpha = sgn(\chi)$

  - Time: 7.5 μs
  - Fidelity: 0.99993

- Two qubit entangling gate ($XX(\chi)$): .
  - Time: 100 μs
  - Fidelity: 0.999

7.3 Resources Estimation Analysis

*ToDo: Show (using tables and/or plots) how resources escalate as the size and pattern of the input changes.

7.4 Analysis of Execution in NISQ Devices

*ToDo: Analyze what would be the maximum input size that can be successfully run on a NISQ device based on this platform.

7.5 Analysis of Execution of Algorithm for Input of Specific Size

*ToDo: Use a resource estimator to determine the characteristics that a device should have to run the algorithm for a specific input size, and determine what would be the runtime.

Chapter 8
FUTURE WORK

*ToDo: Mention how this framework can be used to analyze and compare other hardware platforms.

BIBLIOGRAPHY

[1] Hadamard gate. `https://www.quantum-inspire.com/kbase/hadamard/`. Accessed: 2022-02-25.

[2] Honeywell systemmodel h1. `https://www.honeywell.com/us/en/news/2020/10/get-to-know-honeywell-s-latest-quantum-computer-system-model-h1`. Accessed: 2022-02-25.

[3] Ionq technology. `https://ionq.com/technology`. Accessed: 2022-02-25.

[4] Microsoft qdk samples. `https://github.com/Microsoft/Quantum`. Accessed: 2022-02-25.

[5] Microsoft's quantum development kit (qdk). `https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing/#overview`. Accessed: 2022-02-25.

[6] Shor quantum subroutine image. `https://en.wikipedia.org/wiki/File:Shor%27s_algorithm.svg`. Accessed: 2022-02-25.

[7] C. J. Ballance, T. P. Harty, N. M. Linke, M. A. Sepiol, and D. M. Lucas. High-fidelity quantum logic gates using trapped-ion hyperfine qubits. *Phys. Rev. Lett.*, 117:060504, Aug 2016.

[8] Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits. 2003.

[9] Colin D. Bruzewicz, John Chiaverini, Robert McConnell, and Jeremy M. Sage. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews*, 6(2):021314, Jun 2019.

[10] Isaac L Chuang and Michael A Nielsen. *Quantum computation and quantum information.* Cambridge University Press, 2012.

[11] Cirq Developers. Cirq, May 2021. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

[12] David P. DiVincenzo. The physical implementation of quantum computation. *Fortschritte der Physik*, 48(9–11):771–783, Sep 2000.

[13] C. Figgatt, D. Maslov, K. A. Landsman, N. M. Linke, S. Debnath, and C. Monroe. Complete 3-qubit grover search on a programmable quantum computer. *Nature Communications*, 8(1), Dec 2017.

[14] N. Cody Jones, Rodney Van Meter, Austin G Fowler, Peter L McMahon, Jungsang Kim, Thaddeus D Ladd, and Yoshihisa Yamamoto. Layered architecture for quantum computing. *Physical review. X*, 2(3):031007, 2012.

[15] Mathias Soeken, Mariia Mykhailova, Vadym Kliuchnikov, Christopher Granade, and Alexander Vaschillo. A resource estimation andverification workflow in q#. *Design, Automation and Test in Europe Conference*, 2021.

[16] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Matthew Treinish, Jay Gambetta, Paul Nation, Paul Kassebaum, qiskit bot, Diego M. Rodríguez, Salvador de la Puente González, Shaohan Hu, Kevin Krsulich, Laura Zdanski, Jessie Yu, David McKay, Juan Gomez, Lauren Capelluto, Travis-S-IBM, Julien Gacon, Ashish Panigrahi, lerongil, Rafey Iqbal Rahman, Steve Wood, Luciano Bello, Divyanshu Singh, Drew, Joachim Schwarm, MELVIN GEORGE, Manoel Marques, Omar Costa Hamido, RohitMidha23, Sean Dague, and Shelly Garion. Qiskit/qiskit: Qiskit 0.26.2, May 2021.

[18] Thomas G Wong. *Introduction to classical and quantum computing.* Rooted Grove, 2022.

[19] W. K Wootters and W. H Zurek. A single quantum cannot be cloned. *Nature (London)*, 299(5886):802–803, 1982.

Appendix A

IMPLEMENTATION OF GENERIC PHYSICAL RESOURCES ESTIMATION FRAMEWORK

*ToDo: Add source code that implements the physical resources estimation framework.

Source code can also be found in GitHub.