ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimenti di Informatica - Scienza e Ingegneria
Corso di Laurea in Informatica

Resource Estimation of Quantum Programs through Type Inference: QuRA and Shor's Algorithm

Supervisor:
Chiar.mo Prof.
Ugo dal Lago

Co-Supervisor:
Dott.
Andrea Colledan

Presented by:
Omar Ayache

I Sessione
Anno Accademico 2024/2025

The best feeling
is when you know
how to be lucky

The original phrase "know how to be lucky" is grammatically awkward because luck is typically seen as something that happens to you rather than something you actively do.

Contents

Sommario **i**

Abstract **iii**

1 Introduction **1**

2 Quantum Computing Principles **5**
2.1 A Little Bit of History 5
2.2 Postulates of Quantum Mechanics 6
2.3 What's a qubit? 8
2.3.1 No-cloning Theorem 10
2.4 Quantum Circuits 11
2.4.1 Quantum Gates 12
2.4.2 Universal Gates 12
2.5 Quantum Algorithms 13

3 Shor's Algorithm **15**
3.1 The Impact on Cryptography 15
3.1.1 Classical State of the Art 15
3.1.2 Cryptographic Protocols Under Threat 16
3.2 RSA: The Factorization Fortress 16
3.2.1 The New Advantage 17
3.3 The Protocol 17
3.4 The Period-finding Algorithm 17

4 Quantum Programming Languages **19**
4.1 Introduction 19
4.1.1 Categories of Quantum Programming Languages 19
4.2 Quipper 21
4.2.1 Proto-Quipper-RA 22

4.2.2 Entangling Two Qubits in Proto-Quipper-RA 23
4.3 QuRA 24
4.3.1 Core Functionality 25
4.3.2 Execution Example and Analysis 25

5 Width Analysis of Shor's Quantum Subroutine **27**
5.1 Circuit structure 27
5.2 Quantum Phase Estimation 29
5.3 Oracle Function 30
5.3.1 Adder 31
5.3.2 Modular Adder 37
5.3.3 Controlled Modular Multiplier 41
5.3.4 Modular Exponentiation 48
5.3.5 Root of the Oracle Function 50
5.3.6 Shor's Width Estimation 51

6 Results and Future Work **53**
6.1 Conclusions 53
6.2 Ongoing and Future Work 53

Bibliography **58**

List of Programs

4.1 Proto-Quipper-RA Bell pair 23
4.2 Applying hadamard gates to a list of qubits 25
5.1 The Shor function 29
5.2 Adder circuit 32
5.3 Adder first phase circuit 33
5.4 Adder second phase circuit 34
5.5 Subtractor circuit 35
5.6 Carry operation 36
5.7 Sum operation 36
5.8 Inverse carry operation 36
5.9 Modular adder circuit 38
5.10 Conditional deleting subroutine 39
5.11 Inverse modualr adder 40
5.12 Control modular multiplier circuit 43
5.13 Find bit subroutine 44
5.14 Bit wise Toffoli subroutine 45
5.15 Conditional register transfer subroutine 46
5.16 Inverse control modular multiplier 47
5.17 Modular exponentiation circuit 49
5.18 Oracle function 50

List of Figures

2.1 Bloch sphere representation. Image adapted from Glosser.ca [16] . . . 9

5.1 Shor's algorithm circuit structure. Adapted from [4] 28
5.2 Adder circuit structure. Reproduced from [39] 32
5.3 Carry and sum operation. Reproduced from [39] 36
5.4 Modular adder circuit structure. Reproduced from [39] 38
5.5 Control modular multiplier circuit structure. Reproduced from [39] . 42
5.6 Modular exponentiation circuit structure. Reproduced from [39] . . . 49

Sommario

Il calcolo quantistico promette di risolvere alcuni problemi complessi sfruttando i principi della meccanica quantistica, come la sovrapposizione, che consente ai qubit di sfruttare il parallelismo in un modo diverso da quello disponibile nel calcolo classico. Nonostante questo potenziale, i dispositivi quantistici odierni presentano limitazioni significative, come alti tassi di errore, tempi di coerenza brevi e numero limitato di qubit. Stimare con precisione le risorse necessarie agli algoritmi quantistici, in particolare il numero di qubit e di operazioni, è fondamentale per determinarne la fattibilità sull'hardware esistente e su quello del prossimo futuro. Questa tesi affronta tali sfide con QuRA, un tool per l'analisi del consumo di risorse di programmi quantistici. L'approccio è dimostrato attraverso un'analisi dettagliata dell'algoritmo di Shor, un algoritmo quantistico chiave con importanti implicazioni per la crittografia grazie alla sua capacità di scomporre efficacemente grandi numeri interi, minacciando di rompere protocolli come RSA. Fornendo garanzie matematicamente supportate sui limiti delle risorse, questo lavoro chiarisce le esigenze pratiche dell'implementazione dell'algoritmo di Shor. Questi contributi hanno implicazioni significative sia per lo studio teorico che per la realizzazione pratica del calcolo quantistico.

Abstract

Quantum computing promises to solve certain complex problems by exploiting the principles of quantum mechanics, such as superposition, which allows qubits to leverage parallelism in a way that differs from what is available in classical computing. Despite this potential, today's quantum devices have significant limitations, such as high error rates, short coherence times, and a limited number of qubits. Accurately estimating the resources required by quantum algorithms — in particular, the number of qubits and operations — is essential to determine their feasibility on current and near-future hardware. This thesis addresses these challenges with QuRA, a tool for analysing the resource consumption of quantum programs. The approach is demonstrated through a detailed analysis of Shor's algorithm, a key quantum algorithm with important implications for cryptography thanks to its ability to efficiently factor large integers, threatening to break protocols such as RSA. By providing mathematically supported guarantees on resource bounds, this work clarifies the practical requirements for implementing Shor's algorithm. These contributions have significant implications both for the theoretical study and the practical realization of quantum computing.

Chapter 1

Introduction

Some problems have too many paths to attempt — but what if a computer could take all of them at once? This is the promise of quantum computing. On the other hand, classical computers can only dream of this possibility. These astonishing machines, built on the discovery of quantum mechanics, have different fundamentals; they use the so-called "Modern Physics" in order to unravel an enormous potential. This fundamental difference enables quantum computers to achieve exponential speedups for various problems in cryptography [36], optimization [19], and simulation of quantum systems.

At the heart of this difference lies the unit of information itself. Classical computers process information using bits, which can take the value of either 0 or 1. Quantum computers, in contrast, use quantum bits, or qubits, which can exist in a linear combination of both 0 and 1 simultaneously — a property known as superposition. This allows qubits to encode and process exponentially more information than classical bits, enabling quantum algorithms to realize the so-called *quantum advantage* [41].

Today's quantum devices still struggle with high error rates, short coherence times, and a limited number of logical qubits. These constraints make it crucial to understand how many resources a quantum algorithm truly needs. Knowing precisely how many qubits and how many operations are required helps us determine whether an algorithm can realistically run on the quantum hardware we currently have or may have in the near future.

This work is motivated by the quest for more effective ways to analyze and optimize the resources used by quantum algorithms. As quantum hardware improves, predicting and reducing resource requirements becomes increasingly important if we want to run practical quantum algorithms. Currently, estimating these resources is often done manually or through classical simulations, which can be slow, prone to

errors, and difficult to scale for more complex programs.

This thesis addresses these challenges by examining resource estimation through type inference, using QuRA (Quantum Resource Analysis) [7] to mechanically verify the resource requirements of Shor's algorithm. The main contribution of this work is a detailed analysis of the space complexity of Shor's algorithm, performed semi-automatically through QuRA.

Shor's algorithm is central to this study because of its profound impact on cryptography. Its ability to factor large integers efficiently poses a serious threat to widely used cryptographic protocols like RSA, which underpins much of today's digital security. By analysing the resources required to implement Shor's algorithm, this thesis shines a light on what it would realistically take to execute a quantum algorithm capable of compromising current cryptographic systems.

QuRA does not just *estimate* resources — it formally *verifies* them. When one writes a program in QuRA's language, Proto-Quipper-RA, the type checker infers how many qubits and operations are needed to run the program in a worst-case scenario, automatically discharching mathematical proofs to an SMT solver [5], to ensure that these bounds are correct.

The conclusions drawn here impact both theoretical quantum computing research and the practical implementation of quantum algorithms. On the theoretical side, this work presents, to the best of our knowledge, the first implementation of Shor's algorithm, whose resource requirements are mechanically guaranteed by construction, thanks to QuRA's support for rich type annotations.

Related Work Many works have tackled the problem of quantum resource estimation, both examining the theory behind it and creating practical tools [11, 40]. Key developments in this sense include quantum programming languages that can analyze resource use as you write code, methods to optimize quantum circuits automatically [21], and compilers that convert high-level quantum programs into efficient circuits [20]. However, systematic approaches to resource estimation through type inference remain relatively underexplored, representing a significant opportunity for advancing the field [2, 17, 39].

Outline The structure of this thesis is designed to take the reader step by step from the foundations of quantum computing to the specific results of this research.

- *Chapter 2* begins by setting the stage with a brief historical look at quantum mechanics and the principles that underpin it. The discussion moves on to the qubit, exploring its peculiar properties — such as superposition and entanglement — and the implications of the no-cloning theorem. The chapter wraps

up with an introduction to quantum circuits and gates, as well as a brief look at quantum algorithms.

- *Chapter 3* turns to Shor's algorithm, with a focus on its profound implications for cryptography. It contrasts classical and quantum approaches to factorization, highlights which cryptographic protocols become vulnerable in light of Shor's breakthrough, and explains the algorithm itself, paying close attention to the period-finding routine at its core.
- *Chapter 4* shifts the focus to the tools used to implement quantum algorithms — quantum programming languages. After categorizing different paradigms, it delves into two notable examples: Quipper (and its extension, Proto-Quipper-RA) and QuRA tool. Through code snippets and explanations, the chapter shows how these languages express quantum computations in practice.
- *Chapter 5* is at the heart of this work, presenting an analysis of the circuit width complexity of Shor's algorithm. It starts by outlining the overall structure of the circuit before zooming in on its key components, such as Quantum Phase Estimation and the oracle function. The oracle itself is broken down into its building blocks — the adder, modular adder, controlled modular multiplier, modular exponentiation, and the root of the oracle — with each analysed in detail.
- *Chapter 6* reviews the key results of this thesis and reflects on their broader impact. It closes by exploring promising avenues for future research.

Chapter 2

Quantum Computing Principles

2.1 A Little Bit of History

In the beginning of the 20$^{\text{th}}$ century, physicists realized that the microscopic world — atoms, electrons, photons — is governed by different rules than those we are used to in the macroscopic world that surrounds us every day. This finding eventually became the formal theory of quantum mechanics and, generations later, the notion of quantum-based computing.

For a few centuries prior to that, the main tool people used to describe the world was classical physics. Isaac Newton described it in the 17$^{\text{th}}$ century [27] and it was able to exactly predict the movement of planets, projectiles, and other observable objects. At the beginning of the 20$^{\text{th}}$ century, this paradigm was expanded considerably with Albert Einstein's theory of relativity, which revolutionized our understanding of space, time, and gravitation [12].

But at extremely small scales, experiments produced results that classical theories could not account for. To deal with this, physicists came up with the so-called *old quantum physics* [30] — a combination of early models and concepts — that eventually became the complete theory of quantum mechanics by the mid 20$^{\text{th}}$ century.

Quantum mechanics brought new notions quite different from classical intuition. One of them is the *Heisenberg uncertainty principle*, according to which certain pairs of physical observables, such as position and momentum, cannot simultaneously have a precise determination [22]. Another is *superposition*, the ability of a quantum system to be in many states at once until someone makes a measurement. And a third key concept is *entanglement*, a type of relationship between particles that withstand explanation by classical physics. Together, they formed the backbone of

quantum computing.

Quantum computers represent an extraordinary leap forward that could help us overcome the fundamental limitations of classical computing. That said, it's important to remember that quantum computers are not simply a more efficient version of classical computers. They are governed by entirely different rules, which open doors to new possibilities as well as new challenges that researchers are working on to this day.

2.2 Postulates of Quantum Mechanics

These foundational principles, adapted from Nielsen and Chuang [28], provide the mathematical backbone of quantum computing:

1. **State space**

   Each isolated quantum system corresponds to a complex Hilbert space. The system's state is represented by a unit vector in that space. For a qubit (two-level system) this looks like:

   $$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \quad \text{s.t.} \quad |\alpha|^2 + |\beta|^2 = 1. \tag{2.1}$$

2. **Evolution**

   In the absence of measurement or external interaction, the system evolves deterministically according to the Schrödinger equation. If the system is in the state $|\psi(t)\rangle$ at time t, its evolution is governed by the Hamiltonian H of the system:

   $$i\hbar\frac{d}{dt}\,|\psi(t)\rangle = H\,|\psi(t)\rangle\,. \tag{2.2}$$

   Equivalently, the evolution over a time interval can also be expressed as the application of a unitary operator $U(t_2, t_1)$:

   $$|\psi(t_2)\rangle = U(t_2, t_1)\,|\psi(t_1)\rangle \quad \text{s.t.} \quad U^\dagger U = I. \tag{2.3}$$

   The unitary operator U can be interpreted as a *quantum gate* that transforms the state according to this postulate.

3. **Measurement**

   Observation, i.e. every interaction with the environment, collapses the quantum state in a way described by a set of measurement operators $\{M_m\}$. If the

state before measurement is $|\psi\rangle$, the probability of

$$p(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle , \tag{2.4}$$

$$|\psi_m\rangle = \frac{M_m |\psi\rangle}{\sqrt{p(m)}}, \tag{2.5}$$

with the completeness condition ensuring valid probabilities:

$$\sum_m M_m^\dagger M_m = I. \tag{2.6}$$

4. **Composite systems**

The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. For example, for two qubits the overall state space is

$$\mathcal{H} = \mathbb{C}^2 \otimes \mathbb{C}^2 = \mathbb{C}^4. \tag{2.7}$$

If system 1 is prepared in state $|\psi_1\rangle$ and system 2 in state $|\psi_2\rangle$, then the joint state of the total system is the tensor product

$$|\psi_1\rangle \otimes |\psi_2\rangle . \tag{2.8}$$

Superposition The tensor product structure of composite systems can be understood by considering the superposition principle. If $|A\rangle$ is a state of system A and $|B\rangle$ a state of system B, then the combined system should include the state $|A\rangle \otimes |B\rangle$. Because quantum mechanics allows any linear combination (superposition) of states to also be a valid state, the joint state space must contain all possible superpositions of such product states. This requirement naturally leads to the tensor product construction for the state space of composite systems.

Importantly, not all states in the composite space can be written as a simple product of states from each subsystem. Such states, which can be expressed as

$$|\psi\rangle_{AB} = |\psi\rangle_A \otimes |\phi\rangle_B , \tag{2.9}$$

are called *separable* or *product* states. For example,

$$|0\rangle_A \otimes |1\rangle_B \tag{2.10}$$

is a separable state.

Entanglement In contrast, *entangled* states cannot be factored into states of individual subsystems. A canonical example is the Bell state

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}} \left(|00\rangle + |11\rangle \right), \tag{2.11}$$

which exhibits correlations between the two qubits that cannot be explained by classical means.

2.3 What's a qubit?

A **qubit** (quantum bit) is the fundamental unit of information in quantum computation. It is a quantum system with a two-dimensional Hilbert space, meaning it can be described by two orthonormal basis states, typically denoted $|0\rangle$ and $|1\rangle$. Unlike classical bits, a qubit can exist in a *superposition* 4 of these two states, which lives in the state space 1.

The Bloch Sphere A qubit's state can also be represented geometrically on the *Bloch sphere*, a unit sphere in $\mathbb{R}^3$ that provides an intuitive visualization of its state space. Any pure qubit state $|\psi\rangle$ can be expressed using two real parameters, θ and ϕ, as

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle,$$

where $0 \leq \theta \leq \pi$ and $0 \leq \phi < 2\pi$.

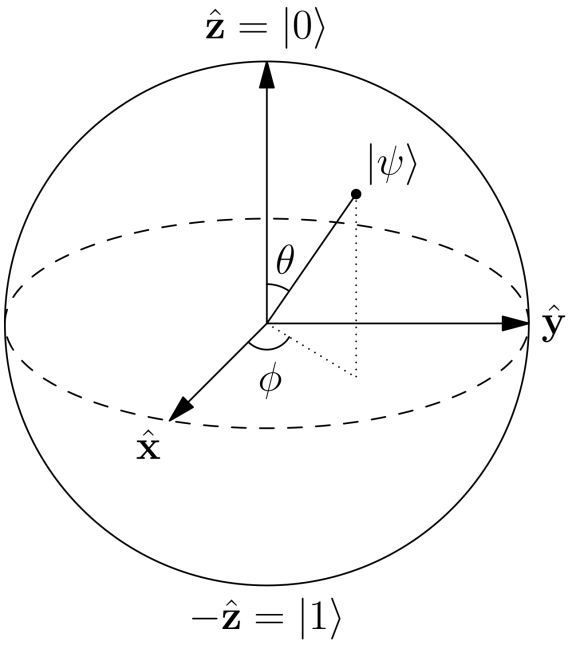

Figure 2.1: Bloch sphere representation. Image adapted from Glosser.ca [16]

Here, θ corresponds to the polar angle (measured from the positive z-axis), and ϕ is the azimuthal angle in the xy-plane. This parametrization maps the qubit state to a unique point on the surface of the sphere. The north pole ($\theta = 0$) represents the basis state $|0\rangle$, and the south pole ($\theta = \pi$) corresponds to $|1\rangle$. Superpositions lie on other points on the sphere's surface, with the relative phase between $|0\rangle$ and $|1\rangle$ encoded in the azimuthal angle ϕ.

The Bloch sphere is a powerful tool to visualize single-qubit operations as rotations around axes, aiding intuition for quantum gates and state evolution.

How are they made?

Physically, a qubit is realized using any quantum system that has exactly two well-defined energy levels that can be manipulated and measured. These systems are governed by the Hamiltonian, which describes their energy structure and dynamics. Several distinct physical implementations of qubits exist, each with advantages and limitations in terms of coherence time, scalability, and controllability:

- **Superconducting qubits**: These are built using Josephson junctions in superconducting circuits. They are among the most widely used in current quantum processors (e.g., by IBM and Google). They operate at millikelvin temperatures and offer fast gate times.

- **Trapped ion qubits**: Individual ions are confined using electromagnetic fields in a vacuum trap. Quantum information is encoded in the ion's internal electronic states. Trapped ions offer high-fidelity operations and long coherence times.
- **Photonic qubits**: Quantum information is carried Quantum Programming Languages by individual photons, with polarization, time-bin, or path encoding. Photons are robust to decoherence and ideal for quantum communication, although implementing two-qubit gates is challenging.

Each of these platforms satisfies the fundamental requirements for qubit implementation, making them suitable for quantum computation under different contexts.

2.3.1 No-cloning Theorem

Another fundamental result in quantum mechanics is the **no-cloning theorem**, which states that it is impossible to create an identical copy of an arbitrary unknown quantum state.

Suppose we have an unknown state $|\psi\rangle$ and a fixed "blank" state $|e\rangle$ intended as a target for cloning. We want to check if there exists a unitary operator U such that

$$U\left(|\psi\rangle \otimes |e\rangle\right) = |\psi\rangle \otimes |\psi\rangle\,, \tag{2.12}$$

meaning U copies the state $|\psi\rangle$ from the fist system into the second.

Proof: Consider two arbitrary normalized states $|\psi\rangle$ and $|\phi\rangle$. If cloning were possible, then

$$U(|\psi\rangle \otimes |e\rangle) = |\psi\rangle \otimes |\psi\rangle\,, \quad \text{and} \quad U(|\phi\rangle \otimes |e\rangle) = |\phi\rangle \otimes |\phi\rangle\,.$$

Taking the inner product of these two results and using unitarity of U, we get

$$\langle\psi|\phi\rangle = \left(\langle\psi|\phi\rangle\right)^2,$$

which implies

$$|\langle\psi|\phi\rangle| = 0 \quad \text{or} \quad 1.$$

Thus, $|\psi\rangle$ and $|\phi\rangle$ must be either orthogonal or identical states.

To see why this is a contradiction for general states, consider the superposition state

$|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$. Then, by linearity,

$$\begin{aligned} U(|+\rangle \otimes |e\rangle) &= U\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |e\rangle\right) \\ &= \frac{1}{\sqrt{2}}\Big(U(|0\rangle \otimes |e\rangle) + U(|1\rangle \otimes |e\rangle)\Big) \\ &= \frac{1}{\sqrt{2}}\Big(|0\rangle \otimes |0\rangle + |1\rangle \otimes |1\rangle\Big), \end{aligned}$$

which is *not* equal to the cloned state $|+\rangle \otimes |+\rangle = \frac{1}{2}(|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle)$.

Therefore, no such unitary U can exist that clones an arbitrary unknown quantum state, proving the no-cloning theorem.

2.4 Quantum Circuits

Quantum circuits provide a framework for performing computations on quantum systems. They are the quantum analogue of classical Boolean circuits (see Section 6 of the textbook [3]), but follow the laws of quantum mechanics.

A standard computation begins with n qubits initialized in the state $|0\rangle^{\otimes n}$ (or sometimes the dual basis state $|1\rangle^{\otimes n}$). A sequence of quantum operations — reversible, unitary gates — is then applied to the qubits, modifying the probability amplitudes and relative phases of the quantum state. These gates exploit essential quantum phenomena such as superposition, entanglement, and interference, which enable quantum algorithms to achieve speed-ups over classical algorithms for certain problems.

As the qubits evolve under the action of the gates, the solution to the computational problem is encoded in the amplitudes of the quantum state. Finally, a measurement projects the quantum state onto one of the computational basis states, revealing the result with a probability given by the squared amplitude of that state.

Designing quantum circuits that are both correct and efficient is crucial. The choice and sequence of gates must guide the computation towards the desired result while minimizing the consumption of physical resources and the accumulation of errors. This makes quantum circuit design both a theoretical and an engineering challenge.

When analyzing and implementing quantum circuits, three key resource measures are often considered:

- **Width:** the number of qubits (wires) required by the circuit.

- **Depth:** the length of the longest path of gates that must be applied sequentially — that is, the number of time steps assuming parallel execution where possible.
- **Gate count:** the total number of elementary gates used in the circuit.

These measures are critical when studying the feasibility of implementing quantum algorithms on real hardware, where the number of available qubits is limited, coherence times are finite, and error rates are significant. Proving bounds on these resources is therefore an essential part of both theoretical analysis and practical design of quantum algorithms.

2.4.1 Quantum Gates

A **quantum logic gate** can be seen as the quantum analogue of a classical logic gate, with some fundamental differences and constraints. Most importantly, quantum gates must be **unitary** operations 2, ensuring that they are reversible and preserve the total probability of the quantum system. This means they cannot create or destroy information, and hence certain classical operations are forbidden in quantum circuits:

- **Fan-out:** In classical circuits, the output of a gate can be freely copied to multiple wires (fan-out). In quantum mechanics, the no-cloning theorem forbids copying an arbitrary quantum state, so fan-out is not allowed.
- **Fan-in:** In classical circuits, multiple wires can be merged (e.g., OR or AND gates). In quantum circuits, such irreversible merging is not possible because it would not correspond to a unitary (reversible) operation.

Quantum gates are mathematically represented by unitary matrices acting on the state vector of the qubits. A single-qubit gate is represented by a 2×2 unitary matrix, while a two-qubit gate is a 4×4 unitary matrix, and so on.

2.4.2 Universal Gates

In classical computation, the NAND gate (or equivalently NOR) is universal, meaning any Boolean function can be implemented using only NAND gates. Quantum computation also has the concept of universality, but it requires a different set of gates because of the need for unitarity and reversibility.

In quantum computation, a set of gates is said to be **universal** if any unitary transformation can be approximated to arbitrary precision by a finite sequence of gates from this set. Some examples of universal set for quantum computing is:

- All single-qubit gates (e.g., the Pauli gates X, Y, Z, and the Hadamard gate H) plus at least one entangling two-qubit gate, such as the controlled-NOT (CNOT) gate.
- Another way to have universal gate set is with Toffoli gate and Hadamard gate (see [1])

Together, these gates sets can approximate any unitary operation on n qubits. Unlike classical gates, where universality is exact and discrete, in quantum circuits universality is usually approximate because the space of unitary operations is continuous.

2.5 Quantum Algorithms

Quantum algorithms are systematic procedures designed to solve computational problems by exploiting the unique features of quantum mechanics, such as superposition, entanglement, and interference.

While a quantum circuit represents a specific implementation of a computation — a concrete sequence of gates acting on qubits — it only operates on inputs of a fixed size. This means that a single circuit alone is not sufficient to fully describe a quantum algorithm, which instead is a more general, abstract recipe capable of handling inputs of arbitrary size.

Just as classical algorithms translate into different circuits depending on the size of the input or other parameters. To describe quantum algorithms within the circuit model, we therefore consider *families of circuits*: sets of circuits where each member corresponds to a specific input size and correctly implements the desired computation for that size. In this way, a quantum algorithm can be seen as specifying how to construct, for any given input size, the appropriate circuit to solve the problem.

This distinction between the abstract algorithm and the concrete family of circuits that realize it is fundamental for designing scalable and meaningful quantum solutions.

As an example, in Chapter 5 we will present an implementation of Shor's algorithm, where classical parameters are used to generate the specific quantum circuit corresponding to the desired input size. This illustrates how a single algorithm gives rise to a family of circuits rather than just one fixed instance.

Chapter 3

Shor's Algorithm

3.1 The Impact on Cryptography

In 1994, Peter Williston Shor, an American mathematician, published a paper that changed cryptographic security [36]. His work demonstrated that quantum computers could outperform the best known classical algorithms for two mathematical problems, solving them exponentially faster: modular exponentiation and prime factorization.

Modern cryptography relies heavily on a fundamental mathematical challenge: given a large integer N, finding its prime factors is computationally hard. For small numbers, this task is straightforward. For example, it is evident that $15 = 3 \times 5$. However, as numbers increase in size, this problem becomes computationally challenging. Attempting to factor a 2048-bit number would require enormous computational resources using classical methods.

3.1.1 Classical State of the Art

Before Shor's breakthrough, the most efficient classical algorithm for integer factorization was the General Number Field Sieve (GNFS)[6], which runs in time:

$$O\left(\exp\left((64/9)^{1/3}(\log N)^{1/3}(\log\log N)^{2/3}\right)\right).$$

While this is technically sub-exponential, it is still too slow for the large numbers used in cryptography like 1024-bit numbers. This computational barrier has allowed

cryptographers to build secure systems on the assumption that factoring large integers is intractable. It is a classic example of using computational complexity as a security foundation.

3.1.2 Cryptographic Protocols Under Threat

Two of the most widely used cryptographic protocols directly depend on the hardness of number-theoretic problems that Shor's algorithm can solve efficiently.

3.2 RSA: The Factorization Fortress

The RSA cryptosystem builds its security directly on the integer factorization problem. In RSA, we begin by generating two large prime numbers p and q, and computing their product $N = pq$, called the modulus. The modulus N is made public, while the prime factors p and q are kept secret.

From p and q, we compute Euler's totient function $\varphi(N) = (p-1)(q-1)$, which counts the number of integers coprime with N. To generate the public and private keys, we then choose an integer e, called the public exponent, such that $1 < e < \varphi(N)$ and $\gcd(e, \varphi(N)) = 1$. This ensures that e has a multiplicative inverse modulo $\varphi(N)$.

The private key d is computed as the modular inverse of e modulo $\varphi(N)$, i.e., it satisfies the congruence:

$$ed \equiv 1 \pmod{\varphi(N)}.$$

This means d can be computed using the extended Euclidean algorithm, which efficiently finds d such that $ed - k\varphi(N) = 1$ for some integer k.

The pair (N, e) forms the public key and is published, while d (and equivalently p and q) must remain secret. If an attacker manages to factor N into p and q, they can compute $\varphi(N)$ and thus also compute d from e, breaking the encryption. With d in hand, the attacker can decrypt any message that was encrypted with the corresponding public key.

Diffie-Hellman: The Discrete Logarithm Challenge

The Diffie-Hellman key exchange protocol is based on the discrete logarithm problem. Given elements g and h in a finite group, the discrete logarithm problem asks us to find an integer x such that $g^x = h$. In Diffie-Hellman, two parties agree on a prime p and generator g, then exchange public keys $g^a \bmod p$ and $g^b \bmod p$ to establish a shared secret $g^{ab} \bmod p$.

Shor's algorithm efficiently solves both the factorization and discrete logarithm problems, making it a threat to both RSA and Diffie-Hellman.

3.2.1 The New Advantage

The key insight behind Shor's polynomial-time complexity of $O((\log N)^3)$ is based on the quantum computing's ability to solve the period-finding problem more efficiently than classical ones. While bit-based computers must essentially search through exponentially many possibilities to find the period of a function, qubit-based computers can exploit quantum interference to perform this task in polynomial time. By preparing a superposition of all possible inputs and applying the Quantum Fourier Transform—which transforms from the computational basis to the frequency domain—the quantum system can efficiently extract the hidden period through interference patterns (see Section 3.5 of the textbook [25]), the quantum system naturally amplifies the correct period through constructive interference while suppressing incorrect answers through destructive interference.

3.3 The Protocol

Peter Shor found a way to convert the factorization problem into a period finding problem, even it could seem a useless operation, this is actually the key factor in making this problem solvable from a quantum computer. The next steps shows how the algorithm changes the factorization of a number N (possibly prime) into an order finding problem:

1. Pick a number a, $1 < a < N$ that is coprime with N (which means that $gcd(a, N) = 1$);
2. Find the "order" r of the function $f_a, N(x) = a^r \mod N$, where r is the smallest exponent such that $a^r = 1 \mod N$;
3. If r is even compute $x \equiv a^{r/2} \mod N$.

   If $\texttt{gcd}(a^{r/2} + 1, N) \neq 1$, then the factors $\{p, q\}$ are in $\{\texttt{gcd}(x + 1, N), \texttt{gcd}(x - 1, N)\}$

   Else find another a

3.4 The Period-finding Algorithm

Now we explain how the algorithm works on a quantum computer. The general idea is to prepare quantum registers: one to encode the number N, and another

to explore all possible exponents in superposition. Thanks to the phenomena of superposition and interference, the outcome of the quantum circuit gives, with high probability, the information we need to determine the period. This quantum part corresponds to step 2 of the aforementioned protocol.

We can break down the period-finding algorithm into two main steps:

1. **Quantum Phase Estimation:** We apply the *quantum phase estimation* algorithm, using a unitary operator U which performs the modular multiplication by a modulo N. We start with a state of the form

$$|0\rangle^{\otimes 2n} \otimes |1\rangle ,$$

   where the first register has $2n$ qubits and the second register has n qubits initialized to $|1\rangle$. The eigenvalues of U encode the period r, and the initial state $|1\rangle$ can be expressed as a sum of eigenvectors of U. Because of this, after applying the phase estimation circuit, we obtain an output in the first register of the form

$$\frac{j}{r} 2^{2n},$$

   where j is an integer chosen uniformly at random from $0, 1, \ldots, r-1$. This result contains information about the period r hidden in the fraction j/r.

2. **Post-processing with continued fractions:** Once we measure the first register and obtain an integer that approximates $j/r\ 2^{2n}$, we use the *continued fractions algorithm* (on a classical computer) to approximate the ratio j/r and deduce the value of r. This classical step is necessary because the quantum output is only an approximation to the actual period, and continued fractions provide an efficient way to recover it exactly.

In essence, the quantum part gives us a good approximation of a rational number whose denominator is the period r, and the classical part computes the exact value of r from this approximation.

In Chapter 5, we will discuss in more detail how the quantum circuit can be implemented, analyzing the width of its routines and subroutines.

Chapter 4

Quantum Programming Languages

4.1 Introduction

Classical programming languages, although fit for digital computation, are not equipped to describe the principles and constraints of quantum mechanics. Fundamental quantum phenomena—such as superposition 4, entanglement 4, and measurement 3—have no direct counterpart in classical logic or memory models. Moreover, quantum operations must be unitary 2 and reversible 5.3, and the no-cloning theorem 2.3.1 prohibits copying arbitrary quantum states—breaking assumptions that be the reason for many classical programming paradigms.

Quantum programming languages (QPLs) are designed to bridge this gap by providing abstractions that naturally reflect the behaviour of quantum systems. Unlike in classical computation, where variables can often be duplicated, reassigned, or discarded freely, quantum data must be manipulated according to the rules of quantum mechanics. For instance, if a variable represents a qubit, it cannot be arbitrarily duplicated or reassigned before it is consumed—such as by applying a gate or performing a measurement. Once a qubit has been used in an operation, the program must track its usage to ensure it is not accidentally reused in a way that would violate the principles of quantum physics.

4.1.1 Categories of Quantum Programming Languages

Quantum programming languages can be broadly divided into two categories, depending on their level of abstraction and their intended execution model:

- **QRAM-Oriented Languages**

  These languages assume an abstract *Quantum Random Access Machine (QRAM)*

model, extending the classical von Neumann model to include quantum memory and instructions. Examples include *Q#* and *OpenQASM*, which are designed to express quantum programs that interact with quantum hardware or simulators while maintaining a clear separation between classical and quantum components.

QRAM languages usually expose primitives for allocating qubits, applying unitary operations, measuring qubits, and integrating classical control flow. They often feature syntax that resembles low-level assembly, allowing the programmer to explicitly orchestrate both classical and quantum operations.

These languages also serve as intermediate representations for quantum programs, making them suitable targets for compilers and toolchains that translate high-level quantum algorithms into executable instructions for specific hardware backends. For example, OpenQASM is often used to specify quantum circuits that can be fed directly to quantum processors.

- **Circuit-Generation Languages**

  In contrast, circuit-generation languages focus on the *construction and manipulation of quantum circuits* at a higher level of abstraction. They provide expressive, often functional, constructs to define and compose quantum circuits programmatically, which are then compiled into lower-level representations (such as QRAM instructions) for run time execution.

  Popular examples include *Qiskit*, *Cirq* and *Quipper*. These frameworks typically offer features such as:

  – Detailed control over the circuit layout and gate sequence

  – Classical control structures and loops for scalable circuit construction

  – Back-end integration for executing circuits on real or simulated quantum devices

  – Resource analysis with classical parameters and circuit optimization tools

  Notably, some circuit-generation languages also include toolchains to *transpile* circuits (see *Qiskit Transpiling documentation* [31])— transforming a high-level circuit into an equivalent one that conforms to the native gate set and connectivity of a given quantum device. Other critical steps supported include *qubit mapping* and *circuit optimization*, aimed at minimizing depth, gate count, and susceptibility to noise, which are crucial in the NISQ (Noisy Intermediate-Scale Quantum) era.

  Functional-style circuit-generation languages like *Quipper*, and *QWire* choose

a different approach, offering high-level abstractions inspired by functional programming. They support powerful type systems that enforce quantum constraints, enable formal reasoning about programs, and encourage modular, compositional design. These properties are especially useful as quantum algorithms grow in complexity, and they contribute to the development of robust and verifiable quantum software.

Languages from this last category, and in particular *Quipper*, will provide the framework for constructing circuits and for establishing the formal statements of our work.

4.2 Quipper

Quipper [35] is a domain-specific quantum programming language embedded in Haskell, designed for the scalable construction of quantum circuits. A key conceptual distinction in Quipper is between *circuit generation time* and *circuit execution time* [18]. During circuit generation, the programmer defines how a circuit should be shaped for a family of classical inputs, including the size of registers or subcircuits. This phase may include *parameters* known at generation time (e.g., an integer specifying the number of qubits). During circuit execution, instead *quantum states* represent inputs that will only be instantiated at execution time on a quantum device. This distinction allows Quipper to define general circuit families parametrized by input size or structure, which is crucial for studying resource usage such as circuit width.

One of the strengths of Quipper lies in the rich set of primitives it provides for manipulating qubits, gates, and circuits. It supports common quantum gates, controlled operations, ancilla management and measurement. Furthermore, it includes higher-level features such as boxed subcircuits, which promote modularity and scalability, as well as functions to estimate circuit resources like gate count, qubit usage, and circuit depth.

Despite its many strengths, Quipper has some limitations. While Haskell provides a strong type system and supports higher-order functional programming—both useful in expressing abstract circuit construction—it lacks two type-theoretic features that are particularly desirable in quantum programming: *linear types* and *dependent types*. Linear types enforce that quantum data is neither duplicated nor discarded without measurement, in accordance with the no-cloning theorem. Dependent types can further enhance correctness guarantees by allowing types to depend on values, for instance, to encode circuit sizes directly in the type system.

Proto-Quipper These shortcomings motivated the development of *Proto-Quipper*, a family of research-oriented languages based on Quipper [13, 8, 24, 33, 14, 32, 10, 9]. Each member of the Proto-Quipper family targets specific aspects of language design, offering a structured environment to study theoretical principles and to model extensions of the original language.

Proto-Quipper incorporates *linear types* to enforce the no-cloning theorem at the type level, ensuring that quantum data cannot be duplicated or discarded arbitrarily. It also employs *dependent types* to allow classical parameters to describe and control the structure of quantum circuits, encoding relationships between classical and quantum data directly in the type system.

Overall, the Proto-Quipper family demonstrates how advanced type-theoretic techniques—such as linearity and dependency—can provide strong correctness guarantees and formal reasoning capabilities, while also offering a flexible framework for exploring and extending the semantics of quantum programming languages.

4.2.1 Proto-Quipper-RA

Proto-Quipper-RA is a type-safe, functional quantum programming language designed to support flexible and compositional resource estimation of quantum circuits. It builds on the foundations of Proto-Quipper but introduces a more expressive type system that enables the static analysis of quantum resources such as gate count, qubit usage, and circuit depth.

The type system of Proto-Quipper-RA combines three powerful techniques: refinement types [37, 38], effect typing [26, 29], and closure types [34].

- **Refinement Types** Refinement types allow for attaching quantitive information to types in order to express constraints on quantum resources. For example, a type might encode the number of qubits a wire carries or the expected size of a subcircuit. These indices are parametric, in that they can depend on classical input parameters, enabling precise and scalable reasoning about families of circuits.
- **Effect Typing** Effect typing extends the type system to track the global resource usage of a circuit-producing function. Each function is annotated not only with the types of its inputs and outputs, but also with a symbolic expression summarizing its side effects, such as how many gates it adds or how many qubits it allocates. These effects are composed according to the structure of the program, allowing the estimation of total circuit cost.
- **Closure Types** Closure are used to safely handle higher-order functions that generate subcircuits. When such functions capture wires from the surround-

ing scope, closure types ensure that the size of these captured resources is correctly reflected in the analysis. This mechanism is essential for maintaining soundness in the presence of abstraction and partial application.

4.2.2 Entangling Two Qubits in Proto-Quipper-RA

Now we slow down a bit and take time to understand how to build a quantum circuit written in Proto-Quipper-RA. The following example aims is to produce a Bell pair, i.e., to entangle two qubits 4 in the state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

```
bell :: ![0](
    forall [0,0] dq.
    forall [0,0] dp.
    (Qubit{dq}, Qubit{dp})
    -o[2,0]
    (Qubit{max(dq+1, dp) + 1}, Qubit{max(dq+1, dp) + 1}))
bell dq dp (q, p) =
    let q = (force hadamard @dq) q in
    let (q,p) = (force cnot @dq+1 @dp) q p in
    (q,p)
```

Listing 4.1: Proto-Quipper-RA Bell pair

Function Body Description

We now describe the operations performed by the circuit, line by line.

1. `let q = (force hadamard @dq) q` applies a Hadamard gate to qubit `q`.
2. `let (q,p) = (force cnot @dq+1 @dp) q p` applies a CNOT gate with control `q` and target `p`.
3. The final result is the pair `(q, p)`, now entangled.

Type Signature Explanation

Going deeper with the explanation, we proceed talking about the "elements" of the language (e.g. `!`, `forall`, `Qubit`, `-o`.) and then the usage of the parameters.

- `-o` denotes a **linear implication** called *lollipop function*, it distinguish between the shape of input the and the shape output.
- `forall dq. forall dp.` introduces a symbolic index variable `dq` and `dp` which is the **dependent type** property of the language, these parameters are classical, which means they are known at generation time.

- `!` is a operator from **linear logic**; it indicates that the function does not contains linear resources bits nor qubits so there is no violation of the No-cloning theorem 2.3.1.
- `(Qubit, Qubit)` is the input **type**: a pair of qubits.

Resource–Tracking Annotations

Now that we have grasped the constructs and symbols of the language, we can proceed to explain the dependent classical parameters.

- `-o[2,0]` the index 2 in `-o[2,0]` is the **effect annotation** of the function. It tells us that bell function produces a circuit of width at most 2 when applied. On the other hand the 0 in `-o[2,0]` is the **closure annotation**, the function's closure has width 0, this means that no additional resources are being added from the environment.
- `![0]` The annotation `0` on the bang is the *closure index*. It indicates that no extra qubit resources are captured from the surrounding context.
- `forall[0,0] dq. forall[0,0] dp.` the two indices means the same of the indices in `-o`
- `(Qubit{dq}, Qubit{dp})` is the input type, where the indices are computed to reflect resource usage after the application of gates. In this case the parameters inside the brackets define the depth of the qubit. Similarly, `(Qubit{max(dq+1, dp) + 1}, Qubit{max(dq+1, dp) + 1})` is the output type, where the indices are updated to reflect the new resource usage after applying the circuit operations. This way of annotating qubit types with symbolic expressions, such as `dq`, `dp` and their combinations, represent the use of **refinement types** in Proto-Quipper-RA: types are refined with quantitative information (like depth), enabling precise reasoning about local circuit metrics.

This example demonstrates how Proto-Quipper-RA expresses not only the behaviour of a quantum circuit but also the symbolic tracking of resources such as width and wire depth. Having a lot of details makes the language particularly useful for formal reasoning and resource estimation.

4.3 QuRA

QuRA is a specialized tool designed for analysing the resource consumption of quantum circuits produced by quantum circuit description programs. Implemented in

Haskell, QuRA is originally based on the foundational work of Colledan and Dal Lago [9], focusing on estimating the *width*, *depth* and *gate counts* of quantum circuits generated by programs written in the Proto-Quipper language.

4.3.1 Core Functionality

The tool operates by taking as input programs written in Proto-Quipper-RA, and returns both the program's type and the width and depth of the circuit it constructs. QuRA employs a sophisticated approach combining a *type inference algorithm* and *SMT-solving* (see textbook [5]) to automatically infer circuit properties with minimal program annotations, making it highly accessible for quantum software developers.

Instead of synthesizing indices that describe specifically the width of constructed circuits, the system now synthesizes indices based on *abstract resource composition operations*. When performing checks on these indices, the operations are translated into concrete arithmetic operators depending on the specific resource being analysed.

Finally, QuRA's analysis is underpinned by a sound theoretical foundation: type checking in QuRA ensures that the inferred bounds are correct for every possible assignment of the classical parameters, guaranteeing that resource constraints hold parametrically, not just for particular inputs.

4.3.2 Execution Example and Analysis

To illustrate the execution of an algorithm written in Proto-Quipper-RA and its analysis with QuRA, consider the following example, which applies Hadamard gates to a list of n qubits. This program demonstrates how dependent and linear types are used to reason about resources such as depth and width symbolically.

```
1 mapHadamard :: ![0](
2     forall[0,0] d.
3     forall[0,0] n.
4     (List[_<n] Qubit{d})
5     -o[n,0]
6     (List[_<n] Qubit{d+1}))
7 mapHadamard d n list =
8   let hadaStep = lift forall step. \(qs,q) :: (List[_<step] Qubit{d
      +1}, Qubit{d}). qs: (force hadamard @d) q
9   in fold(hadaStep, [], list)
```

Listing 4.2: Applying hadamard gates to a list of qubits

Running the QuRA tool on this program with the command:

```
$ qura mapHadamard.pq -g width -l depth
```

produces the following output:

```
Output:
Analyzing file ‘mapHadamard.pq’.
Checked type, width, depth.

mapHadamard :: ![0](forall[0, 0] d. forall[0, 0] n.
    List[_ < n] Qubit{d} -o[n, 0] List[_ < n] Qubit{d + 1})
```

Let us analyse this output in detail:

- The type reported by QuRA matches the declared type of the function, confirming that the program is well-typed and that the type constraints imposed by the type system hold for all values of the classical parameters d and n. This reflects the theoretical guarantee of correctness for all possible instantiations of these parameters.
- The annotation `![0]` indicates that the function is duplicable and does not consume linear resources itself, in line with **linear type** discipline and the no-cloning theorem.
- The **dependent types** appear in the quantified variables `forall[0, 0] d. forall[0, 0] n.`, which allow the function to express the behaviour of the circuit parametrically in terms of d (the initial depth of qubits) and n (the length of the qubit list).
- The **refinement types** are visible in the list length annotation `List[_ < n]` and the qubit indices `Qubit{d}` and `Qubit{d+1}`, which track, respectively, the bounds on list size and the depth of each qubit after applying a Hadamard gate.
- The **linear implication** `-o[n, 0]` shows that the function consumes its input linearly and produces an output while increasing the circuit width by at most n and without adding external closure resources. Here the `n` corresponds to the number of Hadamard gates applied.

This example highlights how QuRA leverages the expressiveness of Proto-Quipper-RA's type system to automatically verify and infer tight upper bounds on the resource usage of the circuit: width, depth, and gate count. Notably, the correctness of the output is guaranteed parametrically for all classical inputs d and n, reflecting the soundness of the underlying type system.

Chapter 5

Width Analysis of Shor's Quantum Subroutine

5.1 Circuit structure

The general structure of the circuit presents three stages. It begins with an initialization phase where n Hadamard gates create a uniform superposition across the upper register qubits, establishing the quantum parallelism foundation for the algorithm.

The middle section contains the oracle function implementation, consisting of controlled operations that encode the modular exponentiation problem into the quantum state through entanglement between the control (the upper quantum register) and target register (the lower quantum register), that we are going to call respectively $|x\rangle$ and $|w\rangle$.

The final stage applies the inverse quantum Fourier transform to extract the relative phase from the quantum state, followed by measurement operations that provide the classical output needed for period determination. This overall structure represents the standard quantum approach to period finding, where quantum superposition enables the parallel evaluation of the function, quantum entanglement encodes the periodicity, and quantum interference through the $\text{QFT}^\dagger$ reveals the structure of the hidden period.

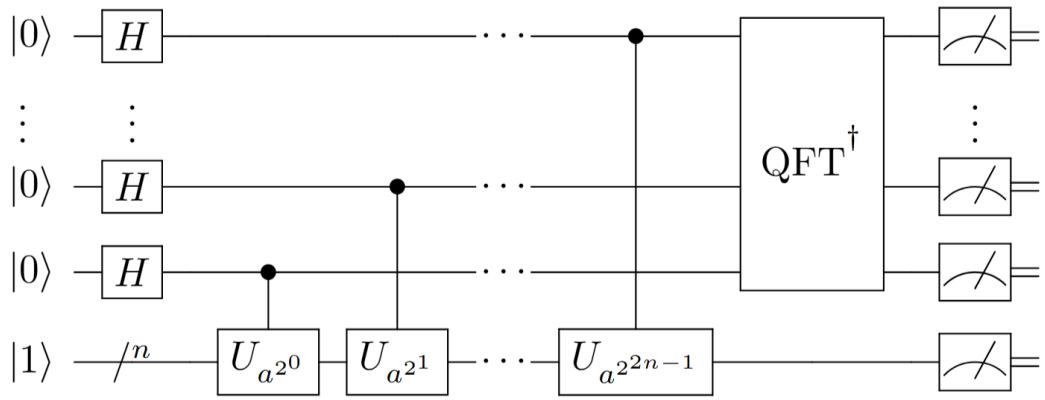

Figure 5.1: Shor's algorithm circuit structure. Adapted from [4]

Complexity at first sight

The computational bottleneck of Shor's algorithm lies fundamentally in its modular exponentiation subroutine, which is represented as the oracle function U in the period-finding component. This operation, which computes $a^x \mod N$ for large integers, dominates both the time and space complexity of the entire algorithm. While the quantum Fourier transform and other quantum operations contribute with a polynomial overhead, the modular exponentiation requires the most significant allocation of quantum resources, particularly in terms of qubit requirements and gate depth. The implementation of this subroutine necessitates careful optimization of quantum arithmetic circuits, as it directly determines the algorithm's practical feasibility for factoring large composite numbers.

Circuit Implementation

The `shor` higher-order function takes as input two parameters: `n`, the bit-length of the number to factor, and `we`, the qubit width required by the `oracle`; and two inputs: the `oracle` function, which has been passed as function.

The routine constructs two qubit register of length n: the $|x\rangle$ register in the state $|0\rangle$, then applies Hadamard gates to create superposition and the ancillary register $|w\rangle$ prepared in $|1\rangle$. Afterwards it applies the `oracle` subroutine and performs inverse Quantum Fourier Transform (`iqft`) for phase estimation.

The last step after we have computed the modular exponentiation is to measure the registers, collapsing them into $|0\rangle$ or $|1\rangle$ for every qubit in the register $|x\rangle$ and $|w\rangle$, the outcome is the classical bit register r, which is the period we were looking for and the base of the power a.

```
shor :: ![0]
    (forall[0, 0] n.
    forall[0, 0] we.
    (Circ[we](
        (List[_ < n + 1] Qubit,
        List[_ < n + 1] Qubit,
        List[_ < n + 1] Bit),
        (List[_ < n + 1] Qubit,
        List[_ < n + 1] Qubit,
        List[_ < n + 1] Bit)),
    List[_ < n + 1] Bit)
    -o[max(n + 1 + n + 1 + n + 1, we), 0]
    (List[i < n + 1] Bit,
    List[i < n + 1] Bit,
    List[_ < n + 1] Bit))
shor n we (oracle, regMod) =
  -- prepare n qubits in the state |0> then apply hadamard
  let x = (force mapHadamard @ n) (force qinit0Many @ n) in
  -- Init register 1
  let w = (force qinit1Many @n) in
  -- oracle function
  let (x, a, regMod) = apply(oracle, (x, w, regMod)) in
  -- phase estimation throught inverse QFT
  let r = (force iqft @ n @ 0) x in
  -- measure the results
  ((force mapMeasure @ n) r, (force mapMeasure @ n) a, regMod)
```

Listing 5.1: The Shor function

General Width Complexity

The type signature expresses the space complexity, that is, the width of the circuit, in the *effect notation* (see Section 4.2.2) of `-o[max(n+1 + n+1 + n+1, we), 0]`. This describes the width of Shor as the maximum value between `3*(n+1)` and `we`. The first term accounts for the working registers, without counting the oracle: one register for the control qubits `x` prepared in the state $|0\rangle$ and Hadamard transformed, the register `w` used as target of the modular exponentiation and the `regMod` for auxiliary classical bits to store N, which will simplify further operations (namely `cModMult` see Section 5.3.3). On the other hand, `we` captures the width of the function `oracle`.

5.2 Quantum Phase Estimation

The estimation of the quantum phase happens through the inverse QFT which is the bridge between quantum computation and classical readout in phase estimation

algorithms. Before we apply the $\text{QFT}^\dagger$, the quantum state exists in a superposition where phase information is encoded in the complex amplitudes of the quantum state. These interference patterns contain the very information we seek to extract, the unknown phase ϕ, but in a form that is completely inaccessible through conventional measurement techniques.

The mathematical explanation of the inverse QFT exceeds the scope of this thesis so the interested reader is advised to go deeper into the topic with the textbook *Quantum Computation and Quantum Information* [28]. Regarding the implementation of the circuit, it is already implemented in QuRA's standard library [9]. The relevant aspect regarding the width of this subroutine is that the signature type is `-o[n, 0]` which means that the QFT has width linear i nits input size.

5.3 Oracle Function

The oracle function U implements the modular exponentiation operation that forms the computational base of Shor's algorithm. This subroutine represents the most technically complex component of the factorization protocol, requiring precise quantum arithmetic circuits to perform modular operations while maintaining quantum coherence throughout the computation.

The implementation approach presented in this thesis follows the theoretical framework developed by V. Vedral, A. Barenco and A. Ekert [39], with additional implementation details drawn from [23].

The oracle performs the unitary transformation

$$U_{a,N} \left|x\right\rangle \left|w\right\rangle \longmapsto \left|x\right\rangle \left|a^x \mod N\right\rangle ,$$

where the parameter a represents the exponential base that varies according to the quantum state encoded in register $|w\rangle$, and N denotes the composite integer targeted for factorization. The base a is determined by the superposition state of the $|w\rangle$ register, generating the periodic sequence $a^x \mod N$ that enables period finding. The parameter N serves as both the modulus for all arithmetic operations and the number whose prime factors the algorithm seeks to determine.

The implementation follows a hierarchical decomposition that can be visualized as a computational tree structure, where each level represents increasing specialization of quantum arithmetic operations.

- The root level, the oracle function 5.3.5 manages the entire modular exponentiation process. This root function calls the modular exponentiation subroutine

5.3.4, which implements the repeated squaring algorithm necessary for efficient exponentiation.

- The modular exponentiation, in turn, relies on the controlled modular multiplier subroutine 5.3.3 to perform individual multiplication operations while maintaining the modular constraint.
- Moving further down the hierarchy, the controlled modular multiplier invokes the modular adder 5.3.2 subroutine to execute addition operations within the specified modulus.
- Finally, at the leaf level of this dependency tree, the modular adder utilizes the fundamental quantum adder 5.3.1 subroutine that performs binary addition operations on quantum registers.

We will start from the bottom—the leaf—the quantum adder, and proceed upwards towards root. This bottom-up approach allows us to establish the fundamental building blocks before constructing the more complex operations that depend upon them.

The following sections will examine each level of the tree structure systematically, beginning with the quantum addition circuits at the leaves and progressively building toward the complete modular exponentiation at the root.

Reversibility Constraint

Given the reversibility constraint mentioned in 2, quantum mechanics imposes the requirement that all operations be reversible, and thus must be implemented using unitary transformations. This constraint shapes the design of the quantum circuit developed in this work. In particular, for every computational subroutine employed, such as addition, modular addition, or controlled modular multiplication, the corresponding inverse operation is implemented. For instance, the adder is complemented by a subtractor, the modular adder by its inverse counterpart, and the controlled modular multiplier by its inverse controlled operation. This pairing ensures that all steps in the algorithm can be cleanly reversed when necessary, a critical feature for operations such as uncomputation and modular inversion, and essential to maintain coherence and prevent the accumulation of garbage states.

5.3.1 Adder

The quantum adder circuit represents the foundational element of the hierarchical implementation, serving as the leaf node in the computational tree that ultimately enables Shor's factorization algorithm. As the most elementary arithmetic

operation, the adder provides the basic building block upon which all higher-level operations: modular addition, modular multiplication, and ultimately modular exponentiation are constructed.

Circuit Architecture

The adder circuit accepts two n-bit quantum registers a and b representing the operands to be summed, along with an overflow qubit b_{n+1} to handle potential carry-out from the most significant bit position. The output consists of the sum distributed across the input registers, with the overflow qubit indicating whether the result exceeds the representational capacity of n bits. This seemingly simple operation requires a slightly unconventional quantum circuit design to achieve the necessary reversibility while maintaining computational efficiency.

The quantum adder follows a two-phase architectural design that manages carry propagation while maintaining reversibility throughout the computation. This implementation, shown in 5.2, divides the addition process into distinct phases that handle carry generation and sum computation one at a time, providing precise control over ancillary qubit usage and enabling proper management of all intermediate states.

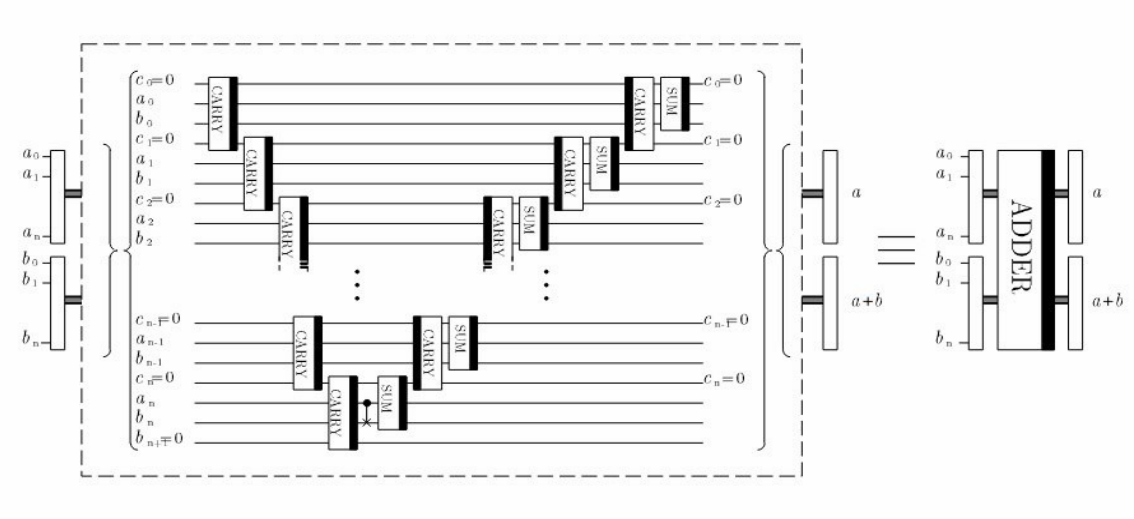

Figure 5.2: Adder circuit structure. Reproduced from [39]

Circuit Implementation

```
adder :: ![0](
    forall[0,0] n.
    (List[_<n+1] Qubit,
```

```
    List[_<n+1] Qubit,
    Qubit)
    -o[3*(n+1) + 1, 0]
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    Qubit))
adder n (a, b, overflow) =
  -- first phase
  let (rest:((c,a),b),overflow) = (force adderFirstPhase @ n) (a,b,
   overflow) in
  -- middle operations
  let (a,b) = (force cnot @ 0) a b in
  let (c,a,b) = (force csum @ 0) (c,a,b) in
  -- second phase
  let (lastc, final) = (force adderSecondPhase @ n) (c, rest) in
  let _ = (force qdiscard @ 0) lastc in
  -- add the first block back in
  let complete = (((force revpair @ n) final) : (a,b)) in
  -- separate a and b
  let (a,b) = (force qunzip @ n+1) complete in
  -- rearrange the bits in the right order
  ((force rev @ n+1) a, (force rev @ n+1) b, overflow)
```

Listing 5.2: Adder circuit

First Phase The `adderFirstPhase` function implements the carry propagation logic using a cascade of `carry` gates applied to each bit position. The circuit begins by organizing the input operands into structured triplets consisting of the two operand bits and an associated carry bit. Each bit position requires an ancillary qubit to store the outgoing carry, which is initialized to $|0\rangle$ for all positions except where carry-in from the previous position is required.

```
adderFirstPhase :: ![0](
    forall[0,0] n.
    (List[_ < n + 1] Qubit,
    List[_ < n + 1] Qubit,
    Qubit)
    -o[3*(n + 1) + 1, 0]
    (List[_ < n + 1]
        ((Qubit, Qubit), Qubit),
    Qubit))
adderFirstPhase n (a, b, overflow) =
  let ab = (force qzip @ n @ 0) (a,b) in
  let reslist:(a,b) = ab in
  let reslist = (force interleave @ n @ 0) reslist in --every first
    position is a, every second is b, every third is c
  let abc = reslist:((a,b),overflow) in
  let cfirst = force qinit0 in
```

```
16   let adderStepFirst = lift forall step . \((reslist, c), ((a,b),
      cnext)) :: ((List[i<step] ((Qubit,Qubit),Qubit), Qubit), ((Qubit
      ,Qubit),Qubit)) .
17     let (c, a, b, cnext) = (force carry @ 0)) (c, a, b, cnext) in
18     (reslist : ((c, a), b), cnext) in
19   let (reslist,overflow) = fold(adderStepFirst, ([], cfirst), abc)
      in
20   -- note that the output triples are reversed so return them in
      the right order
21   ((force revtriplets @ n) (reslist :: List[i<n+1] ((Qubit,Qubit),
      Qubit)), overflow)
```

Listing 5.3: Adder first phase circuit

Second Phase The `adderSecondPhase` function performs the dual function of computing the final sum bits and uncomputing the intermediate carry bits to restore reversibility. This phase applies a sequence of inverse carry operations `icarry` followed by `sum` operations to each bit position, working in reverse order from the msb to the lsb.

```
1  adderSecondPhase :: ![0](
2      forall[0,0] n.
3      (Qubit,
4      List[_<n] ((Qubit, Qubit), Qubit))
5      -o[3*n + 1, 0]
6      (Qubit,
7      List[_<n] (Qubit, Qubit)))
8  adderSecondPhase n (cfirst, abc) =
9    --carry followed by sum, carry discarded
10   let csum = lift \(c,a,b,cnext) :: (Qubit,Qubit,Qubit,Qubit).
11     let (c,a,b,cnext) = force icarry (c,a,b,cnext) in
12     let (c,a,b) = force csum (c,a,b) in
13     let _ = (force qdiscard) cnext in
14     (c,a,b) in
15   let boxedSum = box csum in
16   --step function:
17   let adderStepSecond = lift $ forall step. \((cnext, reslist),((c,
      a),b)) :: ((Qubit, (List[_<step] (Qubit,Qubit))),((Qubit,Qubit),
      Qubit)).
18     let (c,a,b,cnext) = apply(boxedSum, (c,a,b,cnext)) in
19     (c, reslist:(a,b))
20   in fold(adderStepSecond, (cfirst,[]), abc)
```

Listing 5.4: Adder second phase circuit

Subtractor

As we discussed earlier, it is necessary to create the inverse operation of adder, which is the `subtractor`. It is implemented by applying the inverse of each gate in the adder circuit in reverse sequence. Both circuits have identical width complexity, as shown in the type signature `-o[3*(n+1)+1, 0]`, requiring the same number of ancillary qubits and producing outputs with identical qubit overhead.

```
subtractor :: ![0](
    forall[0,0] n.
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    Qubit)
    -o[3*(n+1) + 1, 0]
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    Qubit))
subtractor n (a, b, overflow) =
  -- First Phase
  let ab = (force qzip @ n @ 0) (a,b) in
  -- This operation will be done n times (on list_n+1) so we save
   up the last pair (a,b) then single handedly manage it
  let ab:(alast,blast) = ab in
  let (list,clast) = (force subtractorFirstPhase @ n @ 0) ab in
  -- bottom of the valley, individually handle
  let (clast,alast,blast) = (force csum @ 0) (clast,alast,blast) in
  let (alast,blast) = (force cnot @ 0) alast blast in
  let (clast,alast,blast,overflow) = (force icarry @ 0) (clast,
   alast,blast,overflow) in
  -- Second Phase
  let (list,cfirst) = (force subtractorSecondPhase @ n @ 0) ((force
    revtriplets @ n) list, clast) in
  -- discard the last carry qubit
  let _ = (force qdiscard @ 0) cfirst in
  -- reassemble the last pair into the list
  let list = list:(alast,blast) in
  -- separate a and b
  let (a,b) = (force qunzip @ n+1 @ 0) list in
   --rearrange the bits in the right order
   ((force rev @ n+1) a, (force rev @ n+1) b, overflow)
```

Listing 5.5: Subtractor circuit

Key Subroutines

Carry and Sum The `carry` operation itself is implemented using a combination of Toffoli and CNOT gates that compute the carry-out based on the current bit values and carry-in. The circuit processes each bit position sequentially, with the

carry output from position i serving as the carry input to position $i+1$. This creates a dependency chain that must be maintained throughout the computation to ensure correctness.

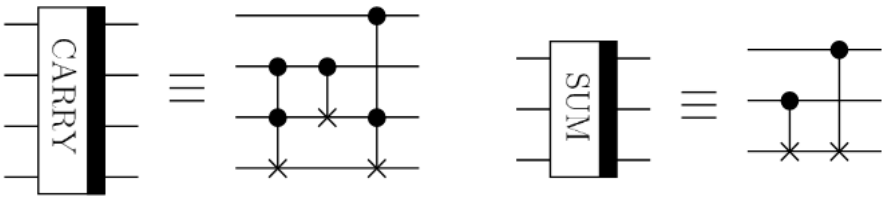

Figure 5.3: Carry and sum operation. Reproduced from [39]

```
carry :: ![0](
    (Qubit, Qubit, Qubit, Qubit)
    -o[4, 0]
    (Qubit, Qubit, Qubit, Qubit))
carry (c, a, b, cnext) =
      let (a, b, cnext) = (force toffoli) a b cnext in
      let (a,b) = (force cnot) a b in
      let (c, b, cnext) = (force toffoli) c b cnext in
      (c, a, b, cnext)
```

Listing 5.6: Carry operation

```
csum :: ![0](
    (Qubit, Qubit, Qubit)
    -o[3, 0]
    (Qubit, Qubit, Qubit))
csum (c, a, b) =
    let (a,b) = (force cnot) a b in
    let (c,b) = (force cnot) c b in
    (c, a, b)
```

Listing 5.7: Sum operation

Inverse Carry The `icarry` operation "uncomputes" the carry bits from the first phase, reversing the carry computation while preserving the information needed for sum calculation. This uncomputation maintains the reversibility of the overall operation.

```
icarry :: ![0](
    (Qubit, Qubit, Qubit, Qubit)
    -o[4, 0]
    (Qubit, Qubit, Qubit, Qubit))
icarry (c, a, b, cnext) =
```

```
6     let (c, b, cnext) = (force toffoli ) c b cnext in
7     let (a,b) = (force cnot) a b in
8     let (a, b, cnext) = (force toffoli) a b cnext in
9     (c, a, b, cnext)
```

Listing 5.8: Inverse carry operation

Width Complexity

The `adder` circuit requires 2 qubit registers for the sum, 1 ancillary register for the carry, and 1 additional bit for overflow detection. This totals `3 * (n + 1) + 1` qubits for the addition process, where `n+1` accounts for the assumption that the input length is greater than zero.

5.3.2 Modular Adder

The quantum modular adder is the middle ground operation of the whole structure. This important subroutine forces the modular constrains by a factor N into the `adder` function. The unitary operation can be summarized with the following formula

$$|a\rangle |b\rangle \longmapsto |a\rangle |a + b \mod N\rangle ,$$

for integers $0 \leq a, b < N$. In order to get this result it is important to notice that we have to use also the `subtractor`. This way, it is possible to perform the operation in place, without use new "memory" (qubit) and guaranteeing the reversibility.

Circuit Architecture

The circuit is built by a sequence of adder and subtractor, to perform an in-place and reversible operation. The process follows 5 steps:

1. The first adder sums the two register with `adder` subroutine performing

$$|a\rangle |b\rangle \longmapsto |a\rangle |a + b\rangle .$$

2. Then a register is swapped with N and subtracted

$$|N\rangle |a + b\rangle \longmapsto |N\rangle |a + b - N\rangle .$$

3. Moving forward to the part of the circuit that makes it reservable, if $a+b-N < 0$ means that we need to add back in the modulus

$$|N\rangle |a + b - N\rangle \longmapsto |N\rangle |a + b\rangle$$

on the other hand if $a + b - N \geq 0$ the N register is set to $|0\rangle$ because we already have the right value

$$|0\rangle |a + b - N\rangle \longmapsto |0\rangle |a + b - N\rangle .$$

4. Moreover, to complete the unitary process and restore the temporary qubit to its initial state $|0\rangle$ we subtract a

$$|a\rangle |a + b \mod N\rangle \longmapsto |a\rangle |(a + b \mod N) - a\rangle$$

and check if an overflow occurs. If this is the case it means that earlier the *temp* bit was in $|1\rangle$ so it is restored to $|0\rangle$, otherwise it is left unchanged.

5. Ultimately, we add a back in, getting the final result of modular addition:

$$|a\rangle |(a + b \mod N) - a\rangle \longmapsto |a\rangle |a + b \mod N\rangle .$$

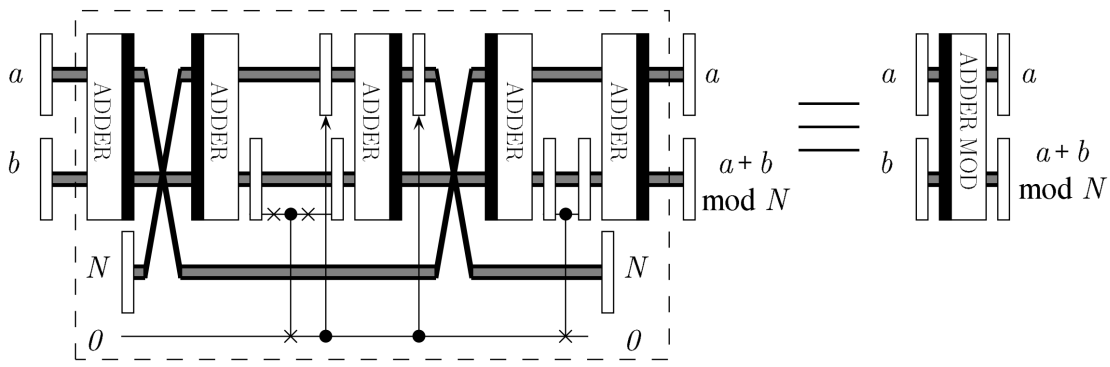

Figure 5.4: Modular adder circuit structure. Reproduced from [39]

Circuit Implementation

```
modAdder :: ![0](
    forall[0,0] n.
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Qubit)
    -o[4*(n + 1) + 3,0]
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Qubit))
modAdder n (a, b, x) =
  -- init overflow qubit
```

```
  let overflow = (force qinit0) in
  -- init temp qubit
  let temp = (force qinit0) in
  -- a+b
  let (a, ab, overflow) = (force adder @n) (a, b, overflow) in
  -- a+b-x
  let (x, abx, overflow) = (force subtractor @n) (x, ab, overflow)
   in
  -- Inverse CNOT
  let overflow = (force qnot) overflow in
  let (overflow,temp) = (force cnot) overflow temp in
  let overflow = (force qnot) overflow in
  -- "cancel" block
  let (tx, temp) = (force deleteBlock @n) (x, temp) in
  -- a+b-x+t*x
  let (tx, abxtx, overflow) = (force adder @n) (tx, abx, overflow)
   in
  -- "cancel" block
  let (x, temp) = (force deleteBlock @n) (tx, temp) in
  -- a+b-x+t*x-a
  let (a, abxtxa, overflow) = (force subtractor @n) (a, abxtx,
   overflow) in
  -- Make temp qubit return to |0>
  let (overflow,temp) = (force cnot) overflow temp in
  -- a+b-x+t*x-a+a
  let (a, abxtxaa) = (force adder @n) (a, abxtxa, overflow) in
  let _ = (force qdiscard) temp in
  (a, abxtxaa, x)
```

Listing 5.9: Modular adder circuit

Key subroutines

Conditional Deleting function The most important subroutine besides `adder` and `subtractor` is `deleteBlock` which is the subroutine that avoids the additions of N if the overflow bit is set to 1. The value of the state is "saved" into a temporary qubit `temp` (initially prepared in state $|0\rangle$) through a series of CNOT gate. This process is being discussed briefly in [39] and implemented in [23].

```
deleteBlock :: ![0](
    forall[0,0] n.
    (List[_<n+1] Qubit, Qubit)
    -o[n + 1 + 1,0]
    (List[_<n+1] Qubit, Qubit))
deleteBlock n (list, temp) =
  let deleteStep = lift forall step. \((qs,temp), q) :: ((List[_<
   step] Qubit, Qubit), Qubit).
    let (temp, q) = (force cnot) temp q in (qs:q, temp)
```

```
  in fold(deleteStep, ([],temp), list)
```

Listing 5.10: Conditional deleting subroutine

Inverse Modular Adder

To perform modular subtraction, we use the inverse of the modular adder circuit. This operation transforms the input state

$$|a\rangle |b\rangle \longrightarrow |a\rangle |a - b \bmod N\rangle$$

and serves as the logical reverse of modular addition.

The structure of the circuit closely mirrors that of the modular adder, but with the key steps executed in the opposite order. It makes use of three subtractors and two adders, along with auxiliary registers and temporary qubits. These are essential for handling conditional operations (such as whether to add or subtract the modulus N) and ensuring the entire process remains unitary and reversible.

As in the case of modular addition, we introduce a temporary register to store N, and a helper qubit to keep track of whether an overflow (or in this case, an underflow) occurred. Throughout the computation, intermediate states are carefully "uncomputed" to clean up any temporary values and return all ancillary qubits to their original states, so that only the desired result $a - b \bmod N$ remains in the output register.

```
iModAdder :: ![0](
    forall[0,0] n.
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Qubit)
    -o[4*(n + 1) + 2,0]
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Qubit))
iModAdder n (a, b, x) =
  -- init overflow qubit
  let overflow = (force qinit0) in
  -- init temp qubit
  let temp = (force qinit0) in
  -- a-b
  let (a, ab, overflow) = (force subtractor @n) (a, b, overflow) in
  -- Make temp qubit return to |0>
  let (overflow,temp) = (force cnot) overflow temp in
  -- a-b+a
  let (a, aba, overflow) = (force adder @n) (a, ab, overflow) in
  let (tx, temp) = (force deleteBlock @n) (x, temp) in
```

```
22  -- a-b+a-t*x
23  let (tx, abatx, overflow) = (force subtractor @n) (tx, aba,
     overflow) in
24  let (x, temp) = (force deleteBlock @n) (tx, temp) in
25  -- a-b+a-t*x+t*x
26  let (x, abatxtx, overflow) = (force adder @n) (x, abatx, overflow
     ) in
27  -- Inverse CNOT and overflow operation
28  let overflow = (force qnot) overflow in
29  let (overflow,temp) = (force cnot) overflow temp in
30  let overflow = (force qnot) overflow in
31   -- a-b+a-t*x+t*x-a
32  let (a, abatxtxa) = (force subtractor @n) (a, abatxtx, overflow)
     in
33  let _ = (force qdiscard) temp in
34  (a, abatxtxa, x)
```

Listing 5.11: Inverse modualr adder

Width Complexity

As shown earlier, the type signature determines the width of the circuit is `4*(n+1)+2`, this is because there been used 3 register of `n+1` qubits for `a, b` and `x`, 1 `temp` qubit for modular operation and the `n+1` carry register plus the `overflow` qubit used during `adder` and `subtractor` subroutines.

5.3.3 Controlled Modular Multiplier

The step before reaching the modular exponentiation is the controlled multiplication, which is done like the others with a set modulo. The function that multiplies two registers modulo N is $f_{a,N}(x) = ax \mod N$, which can be performed with a series of controlled sums and subtractions using the aforementioned *modular adder* subroutine 5.3.2 and its inverse, which are the building block of this subroutine.

Therefore, the multiplication can be seen like this

$$ax \mod N = 2^0 ax_0 + 2^1 ax_1 + \dots 2^{n-1} ax_{n-1} = \sum_{i=0}^{n-1} 2^i ax_i,$$

where $|x\rangle$ is the register that will be used as controller. Additionally, there is another control qubit $|c\rangle$ which is being passed down from the root subroutine. These two registers, at iteration i, encode 2^i into an ancillary register, which is used as an addend in the modular sum.

Circuit Architecture

The circuit operates on three quantum registers: a control register $|c\rangle$, an input register $|x\rangle$, and an ancillary register initialized to $|0\rangle$. The goal is to compute $ax \mod N$, where a and N are classical constants. The operation is performed conditionally, based on the value of the control qubit $|c\rangle$. If $|c\rangle = 1$, the circuit computes multiplication. Otherwise, the state remains unchanged.

The overall structure includes a series of quantum `modAdder` blocks, which are controlled by the individual bits $|x_i\rangle$ and $|c\rangle$. Each modular adder performs a conditional addition of the constant value $a \cdot 2^i \mod N$, where i is the bit index. After each addition, the same controlled operation is performed again to restate the old value of the targeted qubit of ancillary register, to maintain the reversibility of the circuit.

In the end, there is a conditional operation that happens only if $|c\rangle$ was set to $|0\rangle$ and performs a "*copy*".[1]

The entire subroutine can thus be summarized as follows:

$$\begin{cases} |x\rangle\,|0\rangle \longmapsto |x\rangle\,|x\rangle & if\ |c\rangle = |0\rangle\,, \\ |x\rangle\,|0\rangle \longmapsto |x\rangle\,|ax \mod N\rangle & otherwise. \end{cases}$$

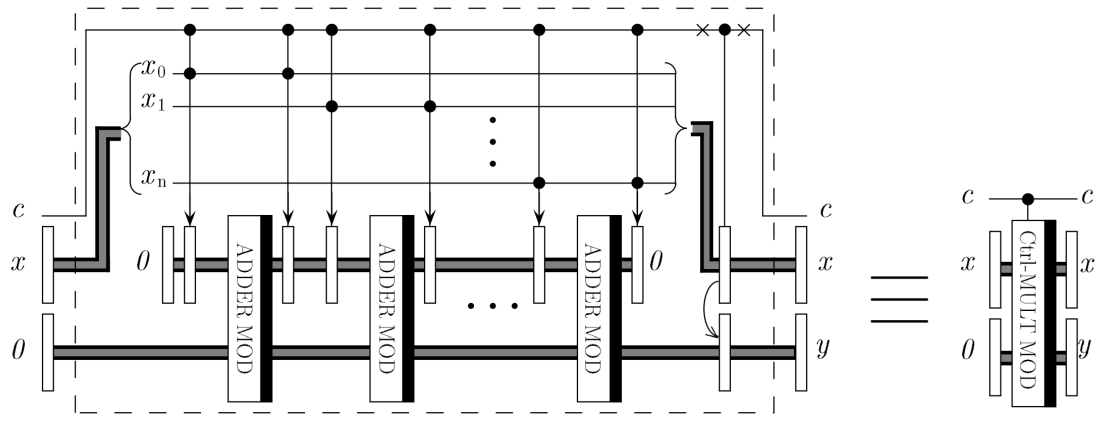

Figure 5.5: Control modular multiplier circuit structure. Reproduced from [39]

Circuit Implementation

[1] This is not an actual copy because of 2.3.1. It is more of a move of the value from the register $|x\rangle$ to the register $|y\rangle$.

```
cModMult :: ![0](
    forall[0, 0] n.
    (Qubit,
    List[_ <n+1] Qubit,
    List[_ <n+1] Qubit,
    List[_ <n+1] Bit)
    -o[6*(n+1)+4, 0]
    (Qubit,
    List[_ <n+1] Qubit,
    List[_ <n+1] Qubit,
    List[_ <n+1] Bit))
cModMult n (c, x, regB, regMod) =
  -- Init working register (a*2^k)
  let regA = (force qinit0Many @ n) in
  -- Control Modular Step Function
  let cModMultStep = lift forall step. \((c,qs,regA,regB,regMod),q)
    :: ((Qubit, List [_<step] Qubit, List[_<n+1] Qubit, List[_<n+1]
    Qubit, List[_<n+1] Bit), Qubit).
    -- Prepare the qubits register from classic register
    let (regMod, regQ) = (force qinitFromBits @ n) regMod in
    -- First toffoli on regA's k-th bit
    let (c,q,regA) = (force bitWiseToffoli @ n @ step) (c, q, regA)
    in
    -- Modular Adder
    let (regA,regB,regQ) = (force modAdder @ n) (regA, regB, regQ)
   in
    -- Discard regQ
    let _ = (force qdiscardMany @ n) regQ in
    -- Second toffoli on regA's k-th bit
    let (c,q,regA) = (force bitWiseToffoli @ n @ step) (c, q, regA)
    in
    (c,qs:q,regA,regB,regMod) in
  -- Fold loop
  let (c,x,regA,regB,regMod) = fold(cModMultStep, (c,[],regA,regB,
   regMod), x) in
  -- Discard regA
  let _ = (force qdiscardMany @ n) regA in
  -- moveBlock
  let (c,x,regB) = (force moveBlock @ n) (c, x, regB) in
  (c, x, regB, regMod)
```

Listing 5.12: Control modular multiplier circuit

Key subroutines

Finding a qubit in a register During the development of the circuit, it became necessary to explicitly apply type coercion, as QuRA was unable to fully infer the types of certain intermediate expressions. This situation emerged particularly in the

`findBit` function, which is used to split a quantum register into two sublists and extract the qubit located at a specific index. The coercion is required on *line 5* of `findBit`, where the expression

```
    let rlist:q = (rlist !:: List[_<(n-step)+1] Qubit)
```

assumes that `rlist` is non-empty. This condition is critical because we de-structure the list to extract the last qubit before reassembling the sublists.

QuRA trusts that this coercion is valid — that is, the list `rlist` indeed contains at least one element at this point. This can be justified by pen-and-paper as follows:

1. The parameter `i` is introduced in the function `bitWiseToffoli`, which is itself called inside the fold loop within `cModMult`.
2. The loop in `cModMult` iterates over the input register `x`, which is of type `List[_ < n + 1] Qubit`.
3. Therefore, the number of iterations, and thus the maximum value of `i`, is at most n.
4. Inside `findBit`, the folding range is defined as `range @ (i + 1)`, which produces the list $\{0, 1, \ldots, i\}$ of length $i + 1$.
5. This implies that during each iteration of the loop in `findBit`, the variable `step` ranges from 0 to i.
6. Since $i \leq n$, it follows that $i + 1 \leq n + 1$, and in particular, for every $0 \leq \text{step} < i + 1$, we have $\text{step} \leq n$.

Initially, `rlist` has type `List[_ < n + 1] Qubit`, and in each iteration one qubit is removed and added to `llist`. After k steps, the size of `rlist` is $n + 1 - k$, which remains strictly positive for all $k < i + 1 \leq n + 1$. Hence, when the destructuring pattern `rlist : q` is applied, the list is guaranteed to be non-empty.

As a result, the coercion `rlist !:: List[_ < (n - step) + 1] Qubit` is valid, and the same argument applies symmetrically to `llist`. Thus, despite the limitations of automatic type inference in this case, the coercions are semantically correct and do not invalidate the analysis.

```
1 findBit :: ![0](
2     forall[0, 0] n.
3     forall[0, 0] i.
4     List[_ < n + 1] Qubit
5     -o[max[step < i + 1]step + 1 + (n - step + 1) - 1, 0]
6     (List[_ < i + 1] Qubit,
7     List[_ < (n + 1) - (i + 1)] Qubit))
8 findBit n i regA =
```

```
  let regA = (force rev @ n+1) regA in -- in order to correctly
   search from the end
  let findStep = lift forall step. \((llist,rlist),_) :: ((List [_<
   step] Qubit, List [_<(n+1)-step] Qubit),()).
    let rlist:q = (rlist !:: List[_<(n-step)+1] Qubit) in --
   coercition type
    let llist = llist:q in (llist,rlist)
  let (llist,rlist) = fold(findStep, ([],regA), (force range @ i+1)
   ) in -- i >= 1
  (llist, (force rev @(n+1)-(i+1)) rlist)
```

Listing 5.13: Find bit subroutine

Bit-Wise Toffoli This function applies Toffili gates on the `regA`'s k-th bit based on the `i`-th iteration of `cMultModStep` function.

At the end of the `bitWiseToffoli` function, a coercion is performed to reassign the type of the reconstructed register `regA` as `List[_ < n + 1] Qubit`. This coercion is:

```
    (c, q, regA !:: List[_ < n + 1] Qubit)
```

This type coercion is a direct consequence of the previous structural guarantees established in the `findBit` function. Specifically, we have already proven that the register is split into two parts whose lengths add up to exactly $n+1$. The reversal and recombination of these sublists through the `concat` function reconstructs the original list with the same length, namely $n+1$. Since no qubits are added or discarded, and all list operations preserve the total length of the register, the coercion to `List[_ < n + 1] Qubit` is both semantically valid and structurally justified.

```
bitWiseToffoli :: ![0](
    forall[0, 0] n.
    forall[0, 0] i.
    (Qubit,
    Qubit,
    List[_ < n + 1] Qubit)
    -o[(max[step < i + 1]step + 1 + (n - step + 1) - 1) + 2, 0]
    (Qubit,
    Qubit,
    List[_ < n + 1] Qubit))
bitWiseToffoli n i (c, q, regA) =
  -- find the qubit for the toffoli operation (placed at the end of
    llist)
  let (llist,rlist) = (force findBit @ n @ i) regA in
  -- get last qubit from llist (target)
  let llist:trg = llist in
  let (c,q,trg) = force toffoli c q trg in
```

```
  -- reassemble the list in the opposite way because of the findBit
     function
  let w = llist : trg in
  let regA = (force concat @ i+1 @ (n+1)-(i+1)) (w, rlist) in
  (c,q,regA !:: List[_<n+1] Qubit)
```

Listing 5.14: Bit wise Toffoli subroutine

Conditional Register Transfer The last subroutine used in the `cModMult` routine is `moveBlock`, responsible for conditionally transferring the quantum state stored in the $|x\rangle$ register into the $|y\rangle$ register (implemented in the code as `regB`), provided that the control qubit $|c\rangle$ is in the $|0\rangle$ state. This corresponds to the final section of the controlled modular multiplication circuit depicted in Figure 5.5, where the thick double wires indicate the flow of multi-qubit registers. In the visual representation, this corresponds to the curved wires connecting $|x\rangle$ to $|y\rangle$ on the right side of the main circuit block, where the condition on `c` ensures that the move only occurs if no operation was performed (i.e. if `c` was 0).

From a quantum programming perspective, this transfer is not a classical copy but a controlled operation that modifies $|y\rangle$ based on the state of $|x\rangle$ and the value of $|c\rangle$. Since direct copying is not permitted by the no-cloning theorem, the move is implemented in the `moveBlock` which will be discuss in the next paragraph.

```
moveBlock :: ![0](
    forall[0,0] n.
    (Qubit, List[_<n+1] Qubit, List[_<n+1] Qubit)
    -o[2*n + 3,0]
    (Qubit, List[_<n+1] Qubit, List[_<n+1] Qubit))
moveBlock n (c, x, regB) =
  -- First X gate for inverse CNOT
  let c = force qnot c in
  -- Step function
  let moveStep = lift forall step. \((c,xs,qs),(x,q)) :: ((Qubit,
   List[_<step] Qubit, List[_<step] Qubit),(Qubit,Qubit)).
    let (c,x,q) = (force toffoli) c x q in (c,xs:x,qs:q) in
  let (c,x,regB) = fold(moveStep, (c,[],[]),(force qzip @n) (x,
   regB)) in
  -- Second X gate for inverse CNOT
  let c = force qnot c in
  (c, (force rev @n+1) x, (force rev @n+1) regB)
```

Listing 5.15: Conditional register transfer subroutine

Inverse Control Modular Multiplier

The `iCModMult` routine mirrors the structure of `cModMult`, but replaces each modular addition with its inverse counterpart `iModAdder`, effectively performing modular

subtraction. As in the forward case, the operation is conditioned on the control qubit `c`, and the `moveBlock` subroutine ensures that the register `regB` is correctly initialized when `c = 0`. The rest of the structure remains symmetric with respect to the forward version.

```
iCModMult :: ![0](
    forall[0, 0] n.
    (Qubit, List[_ <n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Bit)
    -o[6*(n+1)+4, 0]
    (Qubit, List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Bit))
iCModMult n (c, x, regB, regMod) =
  -- Init working register (a*2^k)
  let regA = (force qinit0Many @ n) in
  -- Move the prepared register x into regB if c = 0
  let (c,x,regB) = (force moveBlock @ n) (c, x, regB) in
  -- Control Modular Step Function
  let cModMultStep = lift forall step. \((c,qs,regA,regB,regMod),q)
    :: ((Qubit, List [_<step] Qubit, List[_<n+1] Qubit, List[_<n+1]
    Qubit, List[_<n+1] Bit), Qubit).
    -- Prepare the qubits register from classic register
    let (regMod, regQ) = (force qinitFromBits @ n) regMod in
    -- First toffoli on regA's k-th bit
    let (c,q,regA) = (force bitWiseToffoli @ n @ step) (c, q, regA)
    in
    -- Inverse Modular Adder
    let (regA,regB,regQ) = (force iModAdder @ n) (regA, regB, regQ)
    in
    -- Discard regQ
    let _ = (force qdiscardMany @ n) regQ in
    -- Second toffoli on regA's k-th bit
    let (c,q,regA) = (force bitWiseToffoli @ n @ step) (c, q, regA)
    in
    (c,qs:q,regA,regB,regMod) in
  -- Fold loop
  let (c,x,regA,regB,regMod) = fold(cModMultStep, (c,[],regA,regB,
   regMod), x) in
  -- discard regA
  let _ = (force qdiscardMany @ n) regA in
  (c, x, regB, regMod)
```

Listing 5.16: Inverse control modular multiplier

Width Complexity

Analysing the width of the `cModMult`, QuRA tells us is that the memory requirement for this subroutine is `6*(n+1)+4`, this comes from the `modAdder` which was `4*(n+1)+3`, in addition to that we have a control qubit `c`, a `n+1` qubit register for `regQ` qubits and a `n+1` for bits to be transformed into qubits, store in `regMod`.

5.3.4 Modular Exponentiation

The final component of the algorithm is the *modular exponentiation* subroutine, which composes the previously discussed primitives `cModMult` and `iCModMult`. Its purpose is to compute the function

$$f_{a,N}(x) = a^x \bmod N,$$

which is central to the factorization problem and the core of the quantum speed-up in Shor's algorithm.

Let $x = 2^0x_0 + 2^1x_1 + \cdots + 2^{m-1}x_{m-1}$ be an integer encoded in binary using the qubits $|x_0\rangle, \ldots, |x_{m-1}\rangle$. Using exponent laws, we rewrite a^x as:

$$a^x = a^{x_0} \cdot a^{2x_1} \cdot a^{4x_2} \cdot \ldots \cdot a^{2^{m-1}x_{m-1}} = \prod_{j=0}^{m-1} a^{2^j x_j}.$$

This decomposition is essential because it allows us to implement modular exponentiation using a *sequence of controlled modular multiplications*. For each qubit x_j, a controlled modular multiplication by $a^{2^j} \bmod N$ is performed if $x_j = 1$.

Mathematically, the controlled modular multiplication transforms the state as follows:

$$|x_j\rangle |y\rangle |0\rangle \longmapsto |x_j\rangle |y\rangle |y \cdot a^{2^j x_j} \bmod N\rangle$$

Because $x_j \in \{0, 1\}$, this either multiplies y by $a^{2^j} \bmod N$ (if $x_j = 1$) or leaves it unchanged (if $x_j = 0$).

To ensure unitarity and reversibility, each multiplication step is followed by an *inverse operation* that clears any temporary values in the auxiliary registers (e.g., restoring previously empty register to $|0\rangle$).

Circuit Architecture

Each iteration of the modular exponentiation therefore consists of:

1. Applying `cModMult` controlled on qubit x_j,
2. Swapping the result into the correct register,
3. Applying `iCModMult` to undo temporary computation and clean ancilla.

The transformation for one bit x_k within the full modular exponentiation looks like:

$$|x_k\rangle\,|a^e\rangle\,|0\rangle \longmapsto |x_k\rangle\,|a^e\rangle\,|a^{e+2^k x_k} \bmod N\rangle \longmapsto |x_k\rangle\,|a^{e+2^k x_k} \bmod N\rangle\,|0\rangle\,,$$

where $e = \sum_{j=0}^{k-1} 2^j x_j$ is the partial exponent computed so far. At each stage, the output register accumulates the product $a^x \bmod N$, built from the contributions of all qubits.

This approach guarantees that the output register ends up in the state $|a^x \bmod N\rangle$ and all ancilla qubits are returned to their initial state $|0\rangle$.

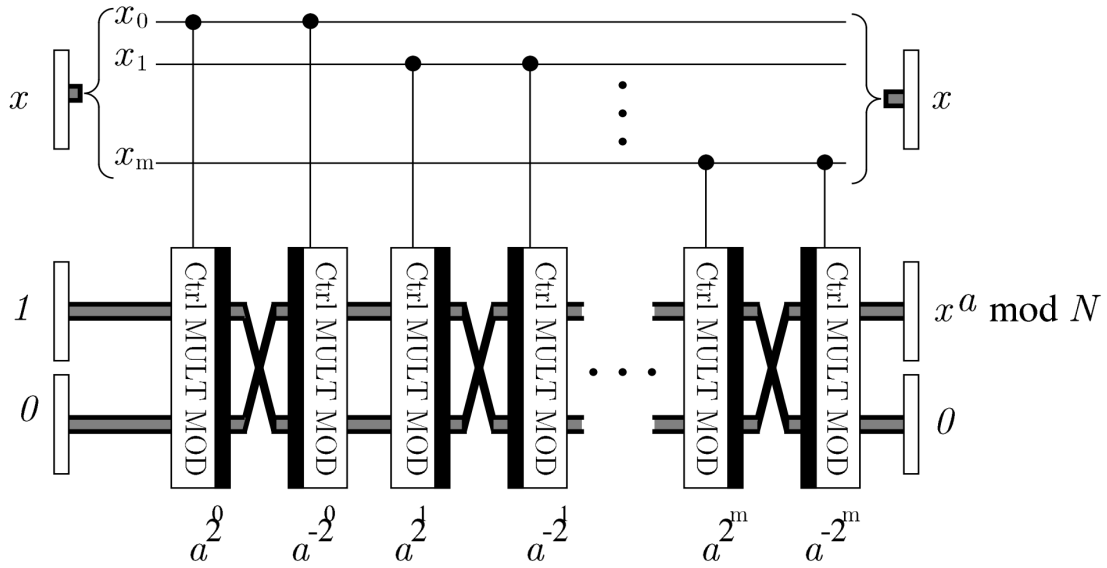

Figure 5.6: Modular exponentiation circuit structure. Reproduced from [39]

Circuit Implementation

```
modularExp :: ![0](forall[0, 0] n.
    (List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Qubit,
    List[_<n+1] Bit)
    -o[7*(n+1)+ 4, 0]
    (List[_<n+1] Qubit,
```

```
8       List[_<n+1] Qubit,
9       List[_<n+1] Qubit,
10      List[_<n+1] Bit))
11  modularExp n (x, reg1, reg0, regMod) =
12    let modularStep = lift forall step . \((qs,reg0,reg1,regMod),q)
        :: ((List[_<step] Qubit, List[_<n+1] Qubit, List[_<n+1] Qubit,
        List[_<n+1] Bit), Qubit).
13      -- Controlled Multiplication
14      let (q, reg1, reg0, regMod) = (force cModMult @n) (q,reg1,reg0,
        regMod) in
15      -- Inverse operation
16      let (q, reg0, reg1, regMod) = (force iCModMult @n) (q,reg0,reg1
        ,regMod) in
17      (qs:q,reg1,reg0,regMod)
18      in fold(modularStep, ([],reg1,reg0,regMod), x)
```

Listing 5.17: Modular exponentiation circuit

Width Complexity

The only new thing compared to others subroutines is that there is a register `x` in which is store the modulo N. So adding `n+1` to the previous complexity of `cModMult`, which was `6*(n+1)+4`, resulting in `7*(n+1)+4`.

5.3.5 Root of the Oracle Function

To conclude the modular exponentiation subroutine and embed it in the broader context of Shor's algorithm, we define the `oracle`, which simply initialize the registers and calls the `modualrExp` routine, discarding the ancillary register `reg0`.

Circuit Implementation

```
1   oracle :: ![0](forall[0, 0] n.
2       Circ[7 * (n+1) + 4]
3           ((List[_< n + 1] Qubit,
4           List[_< n + 1] Qubit,
5           List[_< n + 1] Bit),
6           (List[_< n + 1] Qubit,
7           List[_< n + 1] Qubit,
8           List[_< n + 1] Bit)))
9   oracle n = box $ lift \(x, reg1, regMod) ::
10  (List[_< n + 1] Qubit, List[_< n + 1] Qubit, List[_< n + 1] Bit).
11    -- Prepare ancillay register
12    let reg0 = force qinit0Many @ n in
13    -- Modular exponentiation
14    let (r, a, reg0, regMod) = (force modularExp @n) (x,reg1,reg0,
      regMod) in
```

```
15    -- Discard register 0
16    let _ = (force qdiscardMany @n) reg0 in
17    (r, a, regMod)
```

Listing 5.18: Oracle function

5.3.6 Shor's Width Estimation

With this result it is possible to call the `shor` higher-order function on the oracle and estimate the width of the whole of the Shor's quantum subroutine, which is given by `7*(n+1)+4`. This corresponds to `6*(n+1)+4` qubits and `n+1` bits, meaning that with the implementation of Vedral et al. [39], it is possible to run Shor's algorithm using at most `6*(n+1)+4` logical qubits, where `n+1` is the input size of the number to factorize.

This result can be considered a correct estimation because it has been verified by QuRA's type inference system. Which is formally proven to be sound [9].

Chapter 6

Results and Future Work

6.1 Conclusions

We have demonstrated that the number of qubits required to execute the Shor's algorithm on a quantum device is linear in the bit-size of N. This result is consistent with what Vedral et al. proved in their work [39], where they also showed that the number of qubits scales linearly with the bit-size of N.

The main contribution of this thesis has been the formal estimation of the width of Shor's algorithm, carried out with a verifiable proof. By employing the QuRA tool, we have performed an automatic and static analysis of the quantum circuit, which rigorously infers the resource requirements based on formal type rules.

This automated, proof-based approach not only confirms the expected linear scaling but also provides a trustworthy and reproducible method for estimating circuit width, validated by the soundness of QuRA's type system. To the best of our knowledge, this represents a novel application of static analysis and formal verification techniques to quantify the resource usage of Shor's algorithm.

6.2 Ongoing and Future Work

An aspect that has not yet been fully explored in this dissertation is the estimation of the circuit depth and the total gate count of Shor's quantum subroutine. This is currently the focus of ongoing work. Preliminary analyses suggest that both the depth and the number of gates are expected to be upper-bounded by a polynomial function of the bit-size of N consistently with the known asymptotic complexity of the algorithm [36]. A more detailed study of these complexity measures, as well as

potential optimizations, is planned as future research.

Additionally, the techniques presented in [15] show that it is possible to reduce the required number of qubits further, achieving an upper bound of approximately $3n + 0.002n \log n$. Investigating whether the QuRA tool can be extended to formally verify such more advanced and optimized constructions represents a natural next step. Such work could further strengthen the connection between formal verification and cutting-edge resource-efficient quantum algorithm design.

Bibliography

[1] D. Aharonov. A simple proof that toffoli and hadamard are quantum universal, 2003. URL `https://arxiv.org/abs/quant-ph/0301040`.

[2] M. Amy, O. Di Matteo, V. Gheorghiu, M. Mosca, A. Parent, and J. Schanck. Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3. In *Proc. of SAC 2016*, 2017.

[3] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edition, 2009. ISBN 0521424267.

[4] Bender2k14. Quantum circuit for Shor's algorithm (modular exponentiation). `https://commons.wikimedia.org/w/index.php?curid=34319883`, 2014. Created in LaTeX using Q-circuit. Licensed under CC BY-SA 4.0.

[5] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability - Second Edition*. IOS Press, 2021. ISBN 978-1-64368-160-3.

[6] J. P. Buhler, H. W. Lenstra, and C. Pomerance. Factoring integers with the number field sieve. In A. K. Lenstra and H. W. Lenstra, editors, *The development of the number field sieve*, pages 50–94, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg. ISBN 978-3-540-47892-8.

[7] A. Colledan. Qura documentation, 2025. URL `https://qura.readthedocs.io/en/latest/`.

[8] A. Colledan and U. Dal Lago. On Dynamic Lifting and Effect Typing in Circuit Description Languages. In *Proc. of TYPES 2022*, 2023. ISBN 978-3-95977-285-3.

[9] A. Colledan and U. Dal Lago. Flexible type-based resource estimation in quantum circuit description languages. In *Proc. of POPL 2025*, 2025.

[10] A. Colledan, U. Dal Lago, and N. Vazou. Circuit width estimation via effect typing and linear dependency. *ACM Trans. Program. Lang. Syst.*, May 2025.

ISSN 0164-0925. doi: 10.1145/3737282. URL https://doi.org/10.1145/3737282.

[11] U. Dal Lago, A. Masini, and M. Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010.

[12] A. Einstein. Die grundlage der allgemeinen relativitätstheorie. *Annalen der Physik*, 354(7):769–822, 1916. doi: 10.1002/andp.19163540702. URL https://doi.org/10.1002/andp.19163540702. Archived from the original.

[13] P. Fu, K. Kishida, and P. Selinger. Linear dependent type theory for quantum programming languages: Extended abstract. In *Proc. of LICS 2020*, 2020.

[14] P. Fu, K. Kishida, N. J. Ross, and P. Selinger. Proto-quipper with dynamic lifting. In *Proc. of POPL 2023*, 2023.

[15] C. Gidney and M. Ekerå. How to factor 2048 bit rsa integers in 8 hours using 20 million noisy qubits. *Quantum*, 5:433, Apr. 2021. ISSN 2521-327X. doi: 10.22331/q-2021-04-15-433. URL http://dx.doi.org/10.22331/q-2021-04-15-433.

[16] G. Glosser. Glosser.ca - Own work. Wikimedia Commons, 2010. URL https://commons.wikimedia.org/w/index.php?curid=23263326. CC BY-SA 3.0.

[17] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt. Applying grover's algorithm to aes: Quantum resource estimates. In *Proc. of PQCrypto 2016*, 2016.

[18] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron. *An Introduction to Quantum Programming in Quipper*, page 110–124. Springer Berlin Heidelberg, 2013. ISBN 9783642389863. doi: 10.1007/978-3-642-38986-3_10. URL http://dx.doi.org/10.1007/978-3-642-38986-3_10.

[19] L. K. Grover. A fast quantum mechanical algorithm for database search, 1996. URL https://arxiv.org/abs/quant-ph/9605043.

[20] E. Hainry, R. Péchoux, and M. Silva. A programming language characterizing quantum polynomial time. In *Proc. of FoSSaCS 2023*, 2023. ISBN 978-3-031-30828-4.

[21] T. Häner, T. Hoefler, and M. Troyer. Assertion-based optimization of quantum programs. In *Proc. of OOPSLA 2020*, 2020. doi: 10.1145/3428201. URL https://doi.org/10.1145/3428201.

[22] W. Heisenberg. Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik. *Zeitschrift für Physik*, 43(3):172–198, 1927. ISSN

0044-3328. doi: 10.1007/BF01397280. URL https://doi.org/10.1007/BF01397280.

[23] H. T. Larasati and H. Kim. Simulation of modular exponentiation circuit for shor's algorithm in qiskit. In *2020 14th International Conference on Telecommunication Systems, Services, and Applications (TSSA*, pages 1–7, 2020. doi: 10.1109/TSSA51342.2020.9310794.

[24] D. Lee, V. Perrelle, B. Valiron, and Z. Xu. Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In *Proc. of FSTTCS 2021*, 2021. ISBN 978-3-95977-215-0.

[25] N. D. Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, USA, 2007. ISBN 0521876583.

[26] A. Mycroft, D. Orchard, and T. Petricek. Effect systems revisited—control-flow algebra and semantics. In *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*. Springer International Publishing, 2016. ISBN 978-3-319-27810-0.

[27] I. Newton. *The Mathematical Principles of Natural Philosophy*. Daniel Adee, New York, 1846. URL https://redlightrobber.com/red/links_pdf/Isaac-Newton-Principia-English-1846.pdf. Translated into English with commentary by Andrew Motte; revised by N. W. Chittenden.

[28] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[29] F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design: Recent Insights and Advances*. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-48092-1.

[30] A. Pais. *Subtle is the Lord: The Science and the Life of Albert Einstein*. OUP E-Books. OUP Oxford, 2005. ISBN 9780192806727. URL https://books.google.it/books?id=0QYTDAAAQBAJ.

[31] I. Quantum. Qiskit transpiling documentation, 2025. URL https://quantum.cloud.ibm.com/docs/en/guides/transpile.

[32] F. Rios and P. Selinger. A categorical model for a quantum circuit description language. In *Proc. of QPL 2017*, 2017.

[33] N. Ross. *Algebraic and Logical Methods in Quantum Computation*. PhD thesis, Dalhousie University, 2015.

[34] G. Scherer and J. Hoffmann. Tracking data-flow with open closure types. In *Proc. of LPAR 2013*, 2013.

[35] P. Selinger. Quipper: A scalable quantum programming language, 2013. URL `https://www.mathstat.dal.ca/~selinger/quipper/`.

[36] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi: 10.1109/SFCS.1994.365700.

[37] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Proc. of Haskell 2014*, 2014. ISBN 9781450330411.

[38] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *Proc. of ICFP 2014*, 2014. ISBN 978-1-4503-2873-9.

[39] V. Vedral, A. Barenco, and A. Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147–153, July 1996. ISSN 1094-1622. doi: 10.1103/physreva.54.147. URL `http://dx.doi.org/10.1103/PhysRevA.54.147`.

[40] T. YAMAKAMI. A schematic definition of quantum polynomial time computability. *The Journal of Symbolic Logic*, 85(4):1546–1587, Sept. 2020. ISSN 1943-5886. doi: 10.1017/jsl.2020.45. URL `http://dx.doi.org/10.1017/jsl.2020.45`.

[41] H.-S. Zhong, H. Wang, Y.-H. Deng, M.-C. Chen, L.-C. Peng, Y.-H. Luo, J. Qin, D. Wu, X. Ding, Y. Hu, P. Hu, X.-Y. Yang, W.-J. Zhang, H. Li, Y. Li, X. Jiang, L. Gan, G. Yang, L. You, Z. Wang, L. Li, N.-L. Liu, C.-Y. Lu, and J.-W. Pan. Quantum computational advantage using photons. *Science*, 370(6523):1460–1463, 2020. doi: 10.1126/science.abe8770. URL `https://www.science.org/doi/abs/10.1126/science.abe8770`.

Riconoscimenti

Mi prendo un momento per ringraziare tutta la mia famiglia, in particolare Daniela, Camelia, Angela, Federica, Gregorio, Carlo e Gianluigi. I miei amici Diego, Fabio, Gregorio, Anreea, Davide, Davide†, in particolare, Alice, Emanuele, Lorenzo, Samuele e tutte le persone che mi hanno sostenuto fino a questo momento nella mia vita, coloro che lo stanno ancora facendo e chi lo farà in futuro.

Un ringraziamento speciale va al mio Prof. Ugo dal Lago e al Dott. Andrea Colledan i quali mi hanno sostenuto e fatto crescere in questo mio primo piccolo passo verso un futuro nel campo della ricerca scientifica.

Infine volevo ringraziare me stesso che è riuscito a farcela nonostante tutto.