ABSTRACT

Title of Dissertation: ENGINEERING A CONTROL SYSTEM FOR A LOGICAL QUBIT-SCALE TRAPPED ION QUANTUM COMPUTER

Andrew Russ Risinger
Doctor of Philosophy, 2022

Dissertation Directed by: Professor Christopher Monroe
Department of Physics
Joint Quantum Institute

Quantum computing is a promising field for continuing to develop new computing capabilities, both in its own right and for continued gains as Moore's Law growth ends. Trapped ion quantum computing is a leading technology in the field of quantum computing, as it combines the important characteristics of high fidelity operations, individual addressing, and long coherence times. However, quantum computers are still in their infancy; the first quantum computers to have more than a handful of quantum bits (qubits) are less than a decade old. As research groups push the boundaries of the number of qubits in a system, they are consistently running into engineering obstacles preventing them from achieving their goals. There is effectively a knowledge gap between the physicists who have the capability to push the field of quantum computing forward, and the engineers who can design the large-scale & reliable systems that enable pushing those envelopes. This thesis is an attempt to bridge that gap by framing trapped ion quantum computing in

a manner accessible to engineers, as well as improving on the state-of-the-art in quantum computer digital and RF control systems.

We also consider some of the practical and theoretical engineering challenges that arise when developing a leading-edge trapped ion quantum computer capable of demonstrating error-corrected logical qubits, using trapped $^{171}Yb^{+}$ qubits. There are many fundamental quantum operations that quantum information theory assumes, yet which are quite complicated to implement in reality. First, we address the time cost of rearranging a chain of ions after a scrambling collision with background gases. Then we consider a gate waveform generator that reduces programming time while supporting conditional quantum gates. Next, we discuss the development of a digital control system custom-designed for quantum computing and quantum networking applications. Finally, we demonstrate experimental results of the waveform generator executing novel gate schemes on a chain of trapped ions. These building blocks together will unlock new capabilities in the field of trapped ion quantum computers.

ENGINEERING A CONTROL SYSTEM FOR A LOGICAL QUBIT-SCALE TRAPPED ION QUANTUM COMPUTER

by

Andrew Russ Risinger

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2023

Advisory Committee:
Professor Christopher Monroe, Chair/Advisor
Professor Ronald Walsworth
Professor Donald Yeung
Professor Norbert M. Linke
Professor Christopher Jarzynski

Dedication

To my grandmother, Darlene Floss.

Acknowledgements

All of the work that is presented here is the culmination of many people's designs, efforts, and discussions. To everyone who made this work possible, a great many thanks.

First, I would like to thank my family who encouraged my interest in engineering, and made it a viable career path. Mom, Dad, Zach, and Josh, I am ever grateful for my formative years growing up with you.

My wife, Kate, is simply the most supportive person. She encouraged me as I chose a career path that caused us to be apart during the beginning of graduate school, and then sacrificed to move to Maryland as I finished graduate school. She was continuously uplifting as I struggled with long days, long nights, and doubts about my career path. Words cannot express how thankful I am.

There are too many people to name who exposed me to the potential of a graduate degree in quantum computing, but a few deserve special mention. First, I am thankful for Jeff Leavitt and Charley Adams who allowed a college intern to transition from test engineering to learning about superconducting circuits. There were many Northrop Grumman employees who taught me about superconducting circuits and encouraged me to pursue a graduate degree, especially Sergey Novikov, Zach Keene, Moe Khalil, and John Fusco. Finally, Ken Zick gave me a chance to work on the exciting IARPA QEO program, and my first taste of graduate research into computer architectures for quantum computers.

On top of individuals, I am also thankful for Northrop Grumman Corporation and its employees as a whole, who provided scholarships, internships when circumstances were difficult, and opened career paths that I did not foresee out of high school.

Next, I would like to thank my advisor Christopher Monroe for accepting a computer engineer with little physics background into his research group. I grew so much through exposure to many different topics that I had no experience in. I am also thankful for the many students comprising the diverse group of ion trappers at UMD, who were unendingly friendly and open to sharing knowledge both work-related and fun-related.

I would like to acknowledge all the EURIQA labmates, both current and former, whose tireless work brought our lab from an idea to a (usually) functioning ion trap quantum computer. In rough chronological order: Jason Amini, Jonathan Mizrahi, Kai Hudek, Marko Cetina, Kristin Beck, Michael Goldman, Kevin Landsman, Laird Egan, Daiwei Zhu, Bahaa Harraz, Crystal Noel, Debopriyo Biswas, Or Katz, Lei Feng, and Yichao Yu. While I did not work directly with all of you, your efforts and dedication made the work of EURIQA Breadboard possible. You are all also responsible for teaching me everything that I know about trapping ions.

Kristi Beck, Wen Lin Tan, Mika Chmielewski, and Crystal Noel were all especially kind and welcoming. We had many great lunches and random events, discussing a wide variety of shared interests such as board games, mac and cheese, Ultimate Frisbee, climbing, art, and many others.

I am also thankful for my Electrical and Computer Engineering graduate student friend group: Rajdeep, Devesh, Omid, Abhishek, Ankit, Ananth, Jooik, Sheung, & Austin. Thank you for the friendship during the many classes, hangout sessions, and

study breaks. You helped UMCP feel like home. Thank you also to David and Tristan for the friendship and great experiences playing IM volleyball together, at a time when I was feeling very stressed. Thank you to the UMD ECE staff, who were essential in fostering a welcoming and healthy atmosphere, and for a lot of guidance: Melanie Prange, Emily Irwin, and Bill Churma.

To the many friends that I have made over the years who supported me, you know who you are and you are ever in my heart. Friends from Mt. Hebron High School, 1MC, Alpha Sigma $A\Sigma$, Grove City College, Aletheia College Park, and many others.

Finally, I must again acknowledge the support and love of my wife Kate. A PhD is a long never-ending journey, and you were there every step of the way.

Table of Contents

List of Tables

List of Figures

Listings

List of Abbreviations and Definitions

Abbreviations

2D 2-Dimensional (planar).

3D 3-Dimensional

ADC Analog-to-Digital Converter

AESA Active Electronically-Scanned Array. A class of radar.

AI Artificial Intelligence

AM Amplitude Modulation

AOM Acousto-Optical Modulator

AQC Adiabatic Quantum Computing

ARTIQ Advanced Real-Time Infrastructure for Quantum physics. Control software and hardware ecosystem optimized for academic quantum physics experiments, Python-based.

AWG Arbitrary Waveform Generator

BOM Bill of Materials. List of all parts that are included in a system.

C Celsius

CMOS Complementary Metal-Oxide-Semiconductor

COTS Commercial Off-the-Shelf. Referring to a solution that can be commercially purchased without requiring modification.

CPU Central Processing Unit. The main compute unit in a computer/PC.

Cryo Cryogenic. Generally referring to very low-temperature environments, approximately $\leq 10\,\mathrm{K}$ in this context.

DAC Digital-to-Analog Converter. Produces analog output voltages.

DC Direct Current. Can also mean low-frequency (compared to RF).

DDS Direct Digital Synthesis. Method of producing arbitrary-frequency RF output using a digital device (e.g. FPGA).

DIO Digital Input/Output. Synonym for GPIO.

DMA Direct Memory Access. Method of transferring memory blocks directly from one device to another without per-block CPU commands.

DRTIO Distributed Real-Time Input/Output. System for remotely commanding Input/Output events across ARTIQ FPGAs.

DSL Domain-Specific Language

EOM Electro-Optical Modulator

F Fahrenheit

FIFO First-In-First-Out. Similar to a queue at an amusement park.

FM Frequency Modulation

FMC FPGA Mezzanine Card. Standardized high-density FPGA breakout connector.

FPGA Field-Programmable Gate Array. Reconfigurable digital logic IC, typically mounted on a PCB.

GLUT Gate LUT

GPIO General Purpose Input/Output pins. Generally refers to digital I/O pins.

GRPC Go RPC protocol

GUI Graphical User Interface. A user-facing software interface.

HDL Hardware Description Language

HOA High-Optical Access. Typically refers to the HOA 2.1.1 produced by Sandia National Laboratories

HPC High-Performance Computing. Sometimes also known as a supercomputer, where the primary goal is number-crunching and not data storage/retrieval.

HVAC Heating, Ventilation, and Air Conditioning

I/O Input(s)/Output(s)

IC Integrated Circuit

IDE Integrated Development Environment. Generally a text editor with extra features optimized for software development.

IF Intermediate Frequency. A signal that has been mixed down from its nominal frequency to be easier to process.

IP Intellectual Property

ISA Instruction Set Architecture. Set of instructions that are natively available on a computing system, e.g. `x86`.

LAN Local Area Network

LogiQ IARPA Program aiming to create & characterize an error-corrected logical qubit.

LUT Look-Up Table. A data structure commonly used in FPGAs where entries are given an index (address), by which they can be retrieved. Generally very fast access times but small storage space.

MB Megabytes. The capital B refers to bytes ($8\,\mathrm{bit}$), in contrast to Mb. Equivalent to 2^{20} bytes.

Mb Megabits. 2^{20} bits.

ML Machine Learning

MLUT Memory Map LUT. Sometimes called Sequence LUT (SLUT).

MR Merge Request. See also Pull Request.

MS Refers to a Mølmer-Sørensen gate, described in Section 3.7.3, from [1, 2, 3]

mu Machine Units. See entry for Machine Units.

NA Numerical Aperture. A measure of how much angle that a lens will collect light from. Higher values collect light from a large angle.

NUR Nix User Repository. Collection of unofficial, user-submitted Nix packages.

PC Personal Computer

PCB Printed Circuit Board

PID Proportional-Integral-Derivative. A standard type of locking mechanism [4].

PLUT Pulse LUT

PM Phase Modulation

PMT Photo-Multiplier Tube. Used for collecting weak light signals, e.g. collecting single photons.

PR Pull Request

PWB Printed Wiring Board. Synonym for PCB.

QUBO Quadratic Unconstrained Binary Optimization.

QUBO Quadratic Unconstrained Binary Optimization

RC Resistor-Capacitor. A common low-pass filter design.

RF Radio Frequency. Typically $\geq 1\,\mathrm{MHz}$

RFSoC RF System-on-a-Chip (SOC). FPGA System that combines radio-frequency ADC & DAC I/Os with FPGA reconfigurable logic & an embedded Arm CPU.

RISC Reduced Instruction Set Computer [5, 6].

RPC Remote Procedure Call. Command from one device to another to execute some function.

Sa Sample. For an AWG, the output at one unit of time.

SBC Sideband Cooling. Cooling scheme to reduce temperature beyond the Doppler limit.

SI Système Internationale. Refers to International System of Units, roughly the metric system.

SNR Signal-to-Noise Ratio. Measure of how differentiated a desired measurement signal is against background noise.

SPAM State Preparation & Measurement. Errors in either preparing the initial qubit states, or measuring the state of a qubit.

SPI Serial Peripheral Interface. Real-time digital communication protocol.

UML Unified Markup Language

URL Uniform Resource Locator. Generally the address that a website/file can be found on the Internet.

VM Virtual Machine

WSL Windows Subsystem for Linux. A VM that can be run easily on a Microsoft Windows machine.

Companies/Organizations

AMD Advanced Micro Devices, Inc.CPU designer and manufacturer, owner of Xilinx.

Arm Semiconductor company specializing in embedded CPU designs & instruction sets.

EURIQA Error-corrected Universal Reconfigurable Ion-trap Quantum Archetype. A collaboration on the IARPA LogiQ program. Collaboration between the research groups of Jungsang Kim, Christopher Monroe, and Ken Brown, as well as the companies AOSense, ColdQuanta, and L3Harris.

IARPA Intelligence Advanced Research Projects Agency. US Government agency.

IBM International Business Machines. Makes superconducting quantum computers.

L3Harris Defense Contracting Company. In this context, they develop AOM Cells.

SNL Sandia National Laboratories. Part of the US Department of Energy (DOE).

Xilinx FPGA Manufacturer. Subsidiary of AMD.

Definitions

Gateware Refers to the digital logic programming encoded on an FPGA. This is equivalent to the code that is written in HDLs such as Verilog or VHDL.

Machine Units Binary integer corresponding to native digital units that an embedded device uses to approximate a physical quantity. E.g. a digital device cannot represent an exact arbitrary frequency, but will approximate it to the nearest frequency in its resolution.

Nix A software program to deterministically build any file/software package. See Appendix C for more information.

Octet Waveform generation software written by SNL that runs on Xilinx RFSoCs. This is a combination of "gateware" (HDL) & firmware running on the FPGA's CPU.

Python A popular programming language. See more information in Appendix A.

Qiskit IBM Qiskit python package. Used for defining and executing quantum circuits & pulse schedules.

Number sets

i, j Imaginary Numbers

$\mathbb{C}$ Class of all complex numbers

Other symbols

$\exp$ Exponential. Shorthand for raising the following argument as the exponent for e. $\exp x \equiv e^x$.

0_2 A value in base-2 (binary) consisting of the value zero (0). A subscript of 10 denotes a base-10 (decimal) number.

PyPi Python Packaging Index. Repository for source code and pre-built Python packages that can be easily downloaded.

q[0] Denotes the first qubit in a quantum register, sorted by index.

Physics constants/symbols

$\bar{n}$ Average motional quanta (phonons) in a quantum system.

$\dot{\bar{n}}$ Derivative of the number of motional quanta (phonons) in a system. If positive, the system is "heating", so this quantity is often called the "heating rate".

λ Wavelength, typically of light.

ν Frequency, typically of light.

τ Lifetime in a particular atomic state, measured as the $\frac{1}{e}$ decay time. Assumes exponential decay.

$f_{carrier}$ The difference in frequency between $|F = 0, m_F = 0\rangle$ and $|F = 1, m_F = 0\rangle$, equal to $12.642\,812\,118\,466\,\mathrm{GHz}$.

$|\Psi\rangle$ Arbitrary quantum state.

$|1\rangle$ Qubit computational state in the z-basis, analogous to 1_2. Opposite of $|0\rangle$.

$|0\rangle$ Qubit computational state in the z-basis, analogous to 0_2. Default value that qubits are initialized to.

$^{171}\mathrm{Yb}^{+}$ Ytterbium weighing 171 atomic mass units, singly-ionized. This is the primary ion considered in this work.

Ar Argon

Ba Barium (atomic element)

H Hydrogen

K Kelvin. Unit of temperature measurement, where $0\,\mathrm{K}$ is absolute zero, the lowest possible temperature. $0\,\mathrm{K} \equiv -273.15\,^{\circ}\mathrm{C}$

N Nitrogen

Yb Ytterbium (atomic element)

Chapter 1: Introduction

1.1 Classical Computers

Classical computers have driven many of the new capabilities of the 20th and 21st centuries. It is difficult to find a research field that has not greatly benefited or been fundamentally changed by the advent of computers, in fields as diverse as mathematics [7] and agriculture [8].

The original computers were originally optimized for specific tasks such as ballistic targeting or cracking encryption codes [9]. The growth explosion truly began once computers changed their medium to electronic. Electronic computers have been built primarily on the back of the transistor, which is at the heart of every modern electronic device. The most popular transistor was originally the vacuum tube, but these were superseded by the semiconductor transistor, which enabled the rise of the microprocessor and ubiquitous personal computers [10].

While these innovations have come about due to the efforts of many millions of people over the past century, they all use the same fundamental computing paradigm, which we call classical computing. Classical computing is based on the fundamentals of boolean logic [11]. While boolean logic is quite useful, it is not a native way of describing our world. The world is analog, not binary. Digital systems are an approximation of the

complexities of the world, and that approximation has limitations.

In addition to this approximation, ever-more-complex applications and computations depend on impressive increases in computational power. Commonly referred to as Moore's law, this idea extrapolates that twice as many semiconductor transistors will fit in the same area every two years [12]. [1] Despite the best efforts of the semiconductor industry, this exponential growth cannot continue forever. [2]

To continue increasing computational power, there are now essentially two paths:

- *Continual Incremental Gains*: Continue down the path of researching improved materials, computer architectures, cooling solutions, etc. This will continue to provide gains, but they will likely be incremental and of diminishing returns.

- *Rethink the Computing Paradigm*: An alternative is a complete redesign of how we compute. Some leading candidates include Artificial Intelligence/Machine Learning (AI/ML) [15], optical computing [16], cryogenic computing [14], and quantum computing [17, 18].

[1] An often-overlooked part of Moore's law is that by doubling the transistors per unit area, the heat generated per unit area is also increased. There is a thermodynamic limit to how much heat energy can be removed using common cooling solutions (e.g. fans) [13]. More exotic cooling solutions are possible, but they are difficult to mass-produce, either due to exotic materials such as cryogenics, or high power requirements. Approaches such as digital cryogenic computing [14] attempt to significantly lower the power dissipation of computing, while retaining the same fundamental digital approach of classical computers.

[2] There are other ways to increase compute speed even if transistor density cannot be increased at the desired rate. One such field is referred to as *computer architecture* [5, 6]. Put simply, *computer architecture* is the field of contriving better ways to process instructions and data to compute faster or more efficiently.

1.2 Quantum Computing

Quantum computing is a field which leverages quantum physics to perform operations that would not be otherwise possible. Instead of restricting values to either 0_2 or 1_2[3], the fundamental value in quantum computing, a qubit, can be both $|0\rangle$ & $|1\rangle$, or an arbitrary superposition of those two (Eq. (2.1)). Quantum computing will be described more fully in Chapter 2.

Using quantum information instead of digital information should make feasible certain classes of problems which are intractable on classical computers [17]. However, not every problem will benefit from running on a quantum computer, and quantum computers are not yet powerful enough to demonstrate all of their potential. In the meantime, more scientific and engineering research needs to be performed to enable the potential groundbreaking capabilities. The goal of this thesis is to provide an overview of the current state-of-the-art of one particular implementation of a quantum computer, based on trapped ion technology, and describe some of the steps that we have made to improve future trapped ion quantum computer implementations.

Building and operating a trapped ion quantum computer requires a wide variety of expertise, including the fields of physics, computer science, electrical engineering, computer engineering, and mechanical engineering. Quantum computing pushes the boundaries of each of these fields due to its extreme requirements.

This thesis focuses on the electrical and computer engineering implications of trapped ion quantum computing. A particular emphasis is placed on the control hardware, soft-

[3]The N_2 subscript denotes a binary-encoded number (i.e. base-2).

ware, and computer architecture requirements needed to operate a high-fidelity ($\geq 99\,\%$ 2-qubit gate fidelity) trapped ion quantum computer with more than 10 qubits.

1.2.1 Applications of Quantum Computing

Quantum computing's applications are still being defined. Some examples of these include simulating quantum physics/systems [19], chemical interactions between atoms/-molecules [20, 21], and nuclear physics and material simulations, as summarized in [22].

Even if none of these applications prove practical, there is still a benefit to researching applications of quantum computing. Lessons learned from developing quantum algorithms have been fed back into improving classical algorithms [23, 24].

1.2.2 Current State of Quantum Computing

It is important to not mistake the great potential of quantum computing for the current state of quantum computers. The state of the quantum computing industry is still relatively in its infancy. [4]

1.3 Chapter Summaries

1.3.1 Chapter 1: Introduction

This chapter covers an introduction to classical computing and quantum computing. We also discuss applications where quantum computing shows promise to yield impressive

[4] The analogy that I like to use is that quantum computers are currently in the same state as classical computers in the 1960s. They currently occupy massive amounts of space (roughly the size of a room), they are inefficient, and are only useful for a few highly specialized computations. Only with much engineering work, and the advent of the semiconductor transistor, were they able to shrink and proliferate.

speedups over classical approaches, or to enable solving certain problems that would not be possible using the traditional approaches.

1.3.2 Chapter 2: Quantum Computing Basics

Next, we introduce many of the fundamental quantum computing terms and concepts that are needed to understand the concepts in this thesis. This information is presented to be understandable to those with a typical electrical/computer engineering background, assuming familiarity with linear algebra. We discuss the fundamentals of quantum information, including how a specific instance of a quantum algorithm is specified in a quantum circuit. Then we introduce some of the practical problems that arise when trying to implement these theoretical concepts on a physical system. Finally, we briefly introduce alternative paradigms of quantum computing other than the circuit model.

1.3.3 Chapter 3: Ion Trap Quantum Computing

In this chapter we discuss one implementation of a quantum computer, using qubits encoded in trapped ions. We cover the physical hardware that comprises a trapped ion quantum computer and how we elected to encode information in trapped Ytterbium-171 ions ($^{171}Yb^{+}$). [5] Finally, we discuss how those states are manipulated to execute quantum operations (*gates*).

[5] There are many different ion species other than $^{171}Yb^{+}$ that can be used in quantum computing, but we will primarily focus on this particular ion.

1.3.4 Chapter 4: Control System Design

After understanding how an $^{171}Yb^{+}$ trapped ion quantum computer is designed at a high level, it is important to consider how that system is controlled to actually execute the desired quantum operations. There are two primary real-time control systems for a quantum computer:

- *Digital Control*: sequences digital input & output operations, and controls the initialization and readout of the ions.
- *Qubit Control*: controls the quantum state of a qubit. This is generally performed using radio frequency signals to manipulate the quantum state (see Chapter 5).

This control system is built primarily on top of ARTIQ [25], and uses an FPGA for input/output logic coupled with a soft CPU core[6] to generate real-time control outputs and read in real-time input signals. We detail the design considerations of a trapped ion control system, and consider the different devices that must be synchronized to control a trapped ion quantum computer.

1.3.5 Chapter 5: RFSoC-based Coherent Control System

This chapter describes the waveform generator system based on a Xilinx RFSoC FPGA that was produced by partners at Sandia National Laboratories, and how it is used to generate gate waveforms in real time. We present the trade-offs between different options for waveform generators, and discuss the performance benefits in terms of experiment

[6] A soft CPU core means that the CPU is embedded in the FPGA logic fabric, and is not a single-function pre-designed block. Using a soft CPU core generally incurs a power and performance penalty.

cycle and data transfer that can be achieved by using arbitrary waveform generator (AWG) alternatives.

1.3.6 Chapter 6: PulseCompiler Waveform Synthesis & Specification

To put the RFSoC-based waveform generator to use in executing quantum circuits, users must write a transformation from a quantum circuit to the associated waveforms to execute that circuit. This chapter discusses the framework that we developed based on waveform features of IBM's Qiskit software[26, 27], and the translation layers to convert between a Qiskit circuit to a Qiskit schedule and then finally to the programming data that is sent to an RFSoC for execution.

1.3.7 Chapter 7: Experiments with Ion Trap Control System

Given the infrastructure to program and execute circuits on a quantum computer, the obvious next step is to use the quantum computer to demonstrate certain theoretical applications. This chapter discusses several applications that we have explored, how our technical development work has enabled them, and some of the results that we have collected.

1.3.8 Chapter 8: Advanced Ion Trap Operations

Several advanced operations on a trapped ion quantum computer require tight integration between different parts of the control stack, which are either entirely novel or rarely demonstrated in ions. Here we present our efforts to develop several capabilities for

trapped ion quantum computers:

- *Sorting a chain of ions* when there are several species (i.e. isotopes/elements) of ions present.
- *Mid-Circuit Qubit Measurement*: selectively reading the state of a subset of all qubits without nullifying the quantum information on other qubits, and then continuing with quantum operations. This is a fundamental building block of error-corrected logical qubits.
- *Rapidly Generating Ion-Photon Entanglement* using hardware-accelerated entangling loops.

1.3.9 Appendices

While the appendices generally fall into the category of software engineering, the trapped ion quantum computer that we have developed (EURIQA Breadboard[7]) is at a relatively unique scale in academia where software engineering best practices and discipline are extremely useful, yet too small for an average physicist developer to acquire expertise in their operation. As such, these appendices are meant to bridge the gap and acquire a basic familiarity with topics that are somewhat outside a typical physicist's education.

In the appendices, we discuss the programming language Python [28] (Appendix A), and some tips on its use in experimental setups such as software development best practices and tools for optimizing software. We then discuss Git [29] (Appendix B), *version control* software that can be of a great benefit for parallelizing software develop-

[7]EURIQA is the abbreviation of our collaboration's name: Error-corrected Universal Reconfigurable Ion-trap Quantum Archetype.

ment for experiments and allowing remote development. Finally, we introduce Nix [30] (Appendix C), a functional packaging language that is useful for synchronizing software across multiple laboratory computers.

Chapter 2: Quantum Computing Basics

2.1 Quantum Computing Overview

Quantum computing is an exciting field at the intersection of quantum physics, computer science, and engineering. It is fundamentally different from standard computers in that the operations that it makes are not based on digital logic [11], but instead use quantum physics.

2.1.1 Motivation for Quantum Computing

We live in an quantum world. At its base, all the atoms and matter around us interact and operate based on quantum physics. But when the number of degrees of freedom becomes large[1], that "quantumness" tends to morph into "classical" approximations that generally work well enough, depending on the problem. However, some problems are inherently quantum, and cannot be efficiently represented in digital computers. Some examples of these include simulating quantum physics/systems [19], chemical interactions between atoms/molecules [20, 21], and nuclear physics and material simulations, as summarized in [22].

The fundamental problem with trying to simulate problems of this nature is that

[1] Effectively, "large" means long length scales and large numbers of atoms.

the devices used to perform simulations are not themselves quantum. This leads to enormous overhead in the computing resources needed to simulate the quantum state of e.g. a molecule, and thus it becomes infeasible to simulate on today's available digital computers, or the ones expected to become available in the foreseeable future. [2] For example, in a recent experiment claiming "quantum supremacy" [31], they estimate that simulating $200\,\mathrm{s}$ of experiment runtime would take millions of years on current supercomputers. [3]

However, the fundamental disconnect between classical and quantum computing only arises when translating between the two types of computing. If a *quantum* computer could be created, with enough resources and enough quality to emulate the quantum physics of the inherently quantum system, e.g. a molecule, then these problems would not incur the same overhead, and we could efficiently run calculations that were infeasible before.

2.2 Quantum vs Classical Computing

To understand the differences between classical computing and quantum computing, it is helpful to contrast some of the names that both use for similar concepts, shown in Table 2.1. Here, we focus on the model of quantum computation called "gate-based quantum computing", as it is currently one of the leading models and has the added benefit of being most accessible to a general engineering audience. It should be noted that other

[2] This overhead is due to entanglement. Describing the state of an entangled quantum system scales exponentially with the number of degrees of freedom, which dictates the number of entangled qubits required. The overhead causes many computations to generally take extremely long amounts of compute times, as well as extremely large amounts of memory to store the quantum state while it is being calculated.

[3] The claim about the amount of supercomputer time to simulate this specific experiment has been disputed in [32], though the concept still applies to general quantum computations.

Concept	Digital Logic	Quantum Logic
Fundamental State	Bit (0/1)	Qubit ($\|\Psi\rangle$)
String of States	Register	Quantum Register
Operation	Gate	Quantum Gate
Set of Operations	Circuit	Quantum Circuit or Hamiltonian
Single-State Operation	NOT, Identity	Single-Qubit Rotation
Multi-State Operation	AND, OR	N-Qubit Gate (e.g. Two-Qubit Gate)

Table 2.1: **Comparison of fundamental concepts of Classical Digital Logic vs Quantum Information.** More information on all of these can be found in [18]. Note that $|\Psi\rangle$ means a general quantum state.

models exist (see Section 2.7), which tend to be focused on specific domains of problems.

Many of the terms for describing a quantum algorithm obviously borrow heavily from classical digital logic, such as calling an individual operation a "gate", and a set of operations a "circuit." With these basic definitions and context, we can begin considering how a quantum computer works, and what it needs to implement the operations described in Table 2.1.

2.2.1 Criteria for a Quantum Computer

There are several generally accepted criteria required to operate as a functional quantum computer, which are generally called the DiVincenzo criteria [33]. Throughout this thesis, we will explain our progress in implementing many of these requirements to demonstrate a fully-functional quantum computer.

1. *A scalable physical system with well characterized qubits*: The technology must be able to eventually support enough qubits to perform meaningful algorithms. How to achieve this number is outside the scope of this thesis. However, a relevant corollary

to this requirement is that the control over the qubits must also be scalable. Our work towards implementing scalable control will be discussed in Chapter 5.

2. *The ability to initialize the state of the qubits to a simple fiducial state, such as* $|000\ldots\rangle$: This is the ability to consistently prepare a high-quality known state, which represents the starting point for all subsequent calculations. State initialization will be covered in Section 3.6.2.2.

3. *Long relevant decoherence times, much longer than the gate operation time*: Must be able to perform many quantum gates without losing the quantum information to the environment. [4]

4. *A "universal" set of quantum gates*: Must be able to execute any desired unitary[5], which can be represented ("decomposed") using a standard set of supported gates available on a particular quantum computer system. Addressed in Sections 2.4.2, 2.4.3, 3.7.2 and 3.7.3.

5. *A qubit-specific measurement capability*: Must be able to project an individual qubit's quantum state to the measurement basis, without adding noise beyond the standard "projection noise" of a quantum measurement. Addressed in Section 2.4.4.

[4] In practice, this number will never be zero, so instead effort is generally directed towards *minimizing* the effects of decoherence instead of completely eliminating it.

[5] A unitary matrix, often called a unitary U, is one in which its conjugate (Hermitian) transpose will yield 1 when multiplied by U: $U^{\dagger}U = 1$.

2.3 Quantum States

A *quantum state* represents the quantum-physical state of the quantum computer at any given point in time. Quantum states thus lie within and obey the rules of quantum physics and quantum information, and differ in significant ways from digital states.

2.3.1 Digital States

Digital computers typically encode their information (states) into digital logic, which uses binary states for each "bit" of information. That is, one bit in a digital computer is either 0_2 or 1_2. A set of these bits is called a register (similar to a string of numbers/characters), which collectively represent a state, or a number. For example, the state $1000_2 \equiv 8_{10}$. Note that there are two formats of specifying binary numbers: big-endian [11], where the **most**-significant bit is on the right (e.g. 4 in 1234), and little-endian where the **least**-significant bit is on the right (e.g. 4 in 1234). In other words $\text{string}_{\text{little endian}} \equiv reverse(string)_{\text{big endian}}$. [6] For the remainder of this thesis, numbers missing a subscript are assumed to be in base-10 (decimal), and all registers will be specified in little-endian format.

2.3.2 Qubit

Quantum computers are governed by the laws of quantum physics, which means that their fundamental units must also be "quantum" and obey those laws. The equivalent of a

[6] Standard decimal (base-10) math uses little-endian notation, though most are unfamiliar with the term "little-endian". To reflect this expectation, all binary strings here are assumed to be written in little-endian format unless otherwise specified.

digital bit is a "qubit", or "quantum bit" [18]. A qubit is defined in Eqs. (2.1) to (2.3).

$$|\Psi\rangle = \alpha\,|0\rangle\ + \beta\,|1\rangle \tag{2.1}$$

$$|\alpha|^2 + |\beta|^2 == 1 \tag{2.2}$$

$$\{\alpha, \beta\} \in \mathbb{C} \tag{2.3}$$

While Eq. (2.2) might appear to imply that a qubit is simply in the range $[-1, 1]$, in reality the quantum state $|\Psi\rangle$ is a superposition, having qualities of both basis states $|0\rangle$ & $|1\rangle$ while allowing for complex interference between the basis states. This can be noted by Eq. (2.3), where $\{\alpha, \beta\}$ are defined to be complex numbers, i.e. of the form $a + b * j$ (where $j \equiv i \equiv \sqrt{-1}$).

This definition of a qubit is purely mathematical, and obviously idealistic. Further, it does not define the physical implementation of a qubit, just as digital computers do not specify whether their logic is performed on vacuum tubes, CMOS, or other semiconductor processes.

2.3.2.1 Alternative Qubit Representations

Due to this abstract mathematical nature of a qubit, there can be multiple conventions to represent a qubit. Nominally, any symbol can represent $|0\rangle$ & $|1\rangle$. In a quantum information context, where the standard two-state representation is used, $|0\rangle$ & $|1\rangle$ are the *de facto* standards. In many physics contexts, specifically for spin-1/2 systems[7], these map

[7] Spin-1/2 systems are systems which require two rotations to return to its original configuration. A spin-1/2 system is functionally equivalent to a qubit, as it has only two levels.

as $|0\rangle \rightarrow |\downarrow\rangle, |1\rangle \rightarrow |\uparrow\rangle$. [8] This convention is to more closely match the fundamental physics of quantum spin states. There is also a class of quantum states with infinitely many equally-spaced levels that represent a harmonic oscillator (e.g. mechanical or photon states in a single mode). These are denoted by $|n\rangle$, where the number of vibrational quanta (phonons) or photons is represented by the non-negative integer n ($n \geq 0$). For consistency, the rest of this thesis will primarily use the computational states $\{|0\rangle, |1\rangle\}$ for quantum information, with a few uses of $\{|\downarrow\rangle, |\uparrow\rangle\}$ when discussing ion physics. The numbered states $|n\rangle$ are very important to trapped ion quantum computing (described later in Chapter 3), especially as a "communication bus" of sorts for inter-qubit operations (the concept will be discussed in Section 2.4.3, and then applied to trapped ions in Section 3.7.3). [9]

There are two key alternative qubit representations that are important to follow the rest of this thesis: matrix representation, and the Bloch sphere representation [18].

The matrix representation of a qubit is straightforward: the complex amplitudes $\{\alpha, \beta\}$ are mapped to elements of a column vector. The general single-qubit state $|\Psi\rangle$ is defined as

[8] It is equally valid to reverse these mappings, and map $|0\rangle \rightarrow |\uparrow\rangle$. In the general quantum information context, $|0\rangle$ is more of a label than a fundamental physical quantity.

[9] Quantum states can fall anywhere in between these two extremes of just two states and infinitely many states. There is restriction on how many *basis* states that can be represented on a quantum system, as long as they are coherent and distinguishable. Other common numbers of basis states include 3 (called a *qutrit*) or 10 (called a *qudit*). For now, the majority of the quantum field is working with two basis states (qubits) by convention, so that is the primary consideration we will use here. Some reasons for choosing to consider > 2 states include academic curiosity, or better mapping to physical hardware.

$$|0\rangle = \begin{bmatrix} 1 \\ \\ 0 \end{bmatrix} \tag{2.4}$$

$$|1\rangle = \begin{bmatrix} 0 \\ \\ 1 \end{bmatrix} \tag{2.5}$$

$$|\Psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix} \alpha & \beta \end{bmatrix} \begin{bmatrix} 1 \\ \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha \\ \\ \beta \end{bmatrix} \tag{2.6}$$

Another representation of a quantum state is called the Bloch sphere, named after quantum pioneer Felix Bloch. This representation maps the basis states $|0\rangle$ & $|1\rangle$ to the south & north pole of a sphere, and then uses the complex amplitudes $\{\alpha, \beta\}$ to represent the position on the surface of the Bloch sphere. [10] The equation for the polar coordinates θ, ϕ of a quantum state is:

$$|\Psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\sin\left(\frac{\theta}{2}\right)|1\rangle \tag{2.7}$$

Viewing a state on the Bloch sphere is useful for gaining intuition about several features of quantum computing:

- *Probabilistic Quantum Readout*: discussed in Section 2.4.4. Measurements are the projection of a quantum state onto a vector, commonly the Z axis. Assuming

[10] A single quantum state has 4 unknowns: the real and imaginary parts of both α and β in $|\Psi\rangle$ (Eq. (2.1)). Representing qubits in polar space (Bloch sphere) reduces these unknowns to 2: the angles θ & ϕ. This simplification is valid because of known constraints about quantum states: insensitivity to global phase (multiplying $|\Psi\rangle$ by $e^{i\varphi}$), and $|\alpha|^2 + |\beta|^2 = 1$.

perfect measurement operations and measuring along the Z axis, the probability of measuring in $|1\rangle$ is proportional to the square of the Z component of the Bloch vector. The probabilistic nature of this operation is because any vector that is not exactly $|1\rangle$, such as $(\theta, \phi) = (0.999\pi, 0)$, will have some small component of the $|0\rangle$ vector, resulting in a small probability of measuring $|0\rangle$.

- *Rotations*: changing from one quantum state to another (for a single qubit) can be viewed as a rotation of the Bloch sphere. This will be discussed later in Section 2.4.2.

One primary drawback of the Bloch sphere representation is that the extension to multiple entangled qubits is not very obvious; multiple qubits which are unentangled and separable can be represented by separate, individual Bloch spheres.

It is important to remember that all these representations of qubit states are equally valid, and can be used interchangeably.

2.3.2.2 Common Single-Qubit States

There are several key quantum states that are important to understand. These states are often called basis states, because any other single-qubit state is composed of a linear combination of any of these states. In linear algebra notation, the Pauli basis states are equivalent to the eigenvectors of the Pauli operators [18]. A common task of quantum computers is preparing a qubit in one of these basis states, as they are commonly used as the starting value for a qubit in a quantum algorithm. Measurement also relies on these basis states, as projecting (measuring) an arbitrary state's alignment to one of these basis

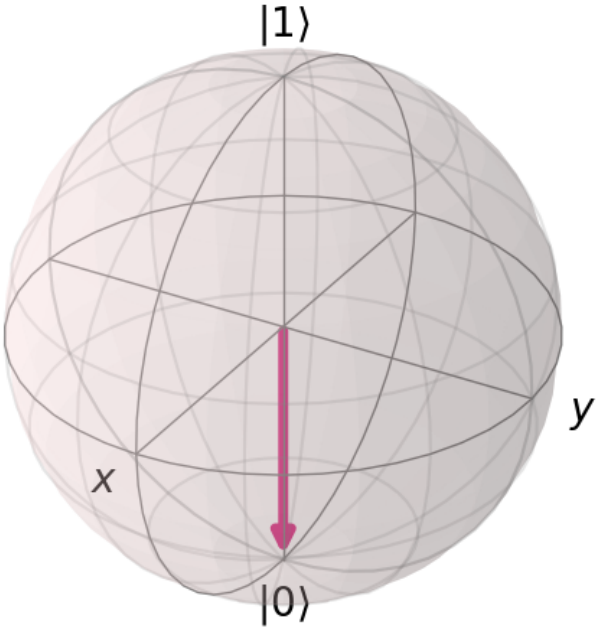


Figure 2.1: **Bloch sphere representation of $|0\rangle$.** In the Bloch sphere representation, any point on the surface of the sphere is a valid state. Some conventions will invert the Z ($|0\rangle$, $|1\rangle$) axis, but this thesis will not use that convention to align more closely with the ($|\downarrow\rangle$, $|\uparrow\rangle$) physics convention. The only difference between these two conventions is an inversion of the sphere across the $x - y$ plane.

states gives an idea of how similar they are. By preparing the same state many times, and then projecting onto each of the basis states, you can estimate what the original state was by collecting statistics. In other words, this state measurement in this manner is measuring the overlap of a state with each of the Pauli basis states.

2.3.3 Multi-Qubit States

To perform useful operations with a quantum computer, it is necessary to add many qubits. The exact number of required qubits depends on the problem that you are trying to solve, but will generally include ≥ 2 qubits, and usually significantly more. Thus, we need to expand our understanding of quantum information to systems with many qubits.

To leverage the power of quantum computing, there must not only be many qubits but there must also be a way to entangle the qubits, which will be described in Section 2.3.4.4. For every extra qubit that is added to a quantum state, it increases the size of the state matrix dimensions by a factor of 2. [11] Adding a single extra qubit to the quantum state increases the size of the unitary matrix by a factor of 2 for each dimension. Intuitively, each qubit can be thought of as exponentially expanding the variables that can be used to solve problems. [12] By adding more qubits to the computation, more complex problems can be solved.

[11] For example, just storing the state of 2 qubits (the statevector) requires a 1×4 matrix. Performing an operation on that system requires a unitary with dimensions 4×4. Adding a single extra qubit to the system requires a 1×8 matrix for the statvector, and a 8×8 matrix for any unitary operation.

[12] Explicitly, adding an extra N-th qubit increases the number of variables that can be solved for by 2^{N-1}.

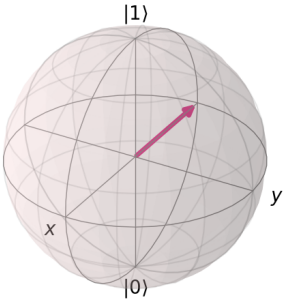
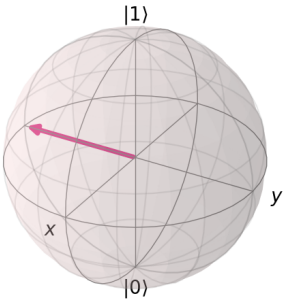
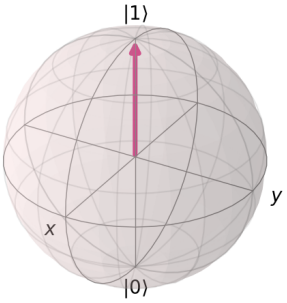
State Name	Rotation Matrix	Eigenvector(s)	Bloch Sphere Form
Identity	$\sigma_0 = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$		N/A
Pauli $\pm X$	$\sigma_1 = \sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ \pm 1 \end{bmatrix}$	
Pauli $\pm Y$	$\sigma_2 = \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	$\frac{1}{\sqrt{2}} \begin{bmatrix} \pm i \\ 1 \end{bmatrix}$	
Pauli $\pm Z$	$\sigma_3 = \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$	

Table 2.2: **Common Quantum Operators and their different representations.** These rotation matrices are more precisely called unitary matrices. Any qubit operation can be denoted by an equivalent unitary matrix. Note that the eigenvectors of the Pauli Z (σ_3) $\begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ are the qubit basis states $|0\rangle$ & $|1\rangle$, respectively. We call this the "Z-basis", or the computational basis. If direct measurement in only the Z-basis is possible, the other bases can still be measured by rotating from e.g. the Y-basis to the Z-basis using only single-qubit rotations. For each specified state, the Bloch sphere image shown denotes the positive direction (e.g. the +Z basis); the negative direction is along the same axis, opposite the origin point.

2.3.3.1 Quantum Registers

A quantum register is a set of individual quantum systems, which are combined to form a larger quantum state. The classical equivalent of this is a digital register, also called a bit string, such as 100101_2. By continuing this analogy to digital computers, each bit in the bit string is composed of one wire/register or (approximately) a few transistors, but their collective whole allows them to represent more information, such as the decimal number 37_{10} via the binary bit string 100101_2. It is simply impossible to represent 37_{10} using a single binary digit $\{0, 1\}$.

In digital computers, there is a distinction between a bus and a register. A bus has no storage, and can be considered as a set of wires. On the other hand, a register will store the last value set into it, and continuously output that value until it is updated (typically via a clock signal) to the newest value.

The terminology for quantum computers is not as concretely defined, because the line between a bus and a register is blurred due to the no-cloning theorem (see Section 2.3.4.3). In a quantum computer, a qubit can be considered memory (a register) because it stores a value, and will continually keep that value until it is intentionally modified. Unintentional modifications including dephasing, state decay, or imperfect operations such as over/under-rotation will break this model of a qubit keeping its value. Nevertheless, a qubit still meets a qualitative definition of a memory in the ideal, error-free case It does not meet the definition of a wire because accessing or moving the data requires an active operation to move it between two different "locations." A quantum memory does not fit neatly into the classical computing load-store architecture model,

because quantum computations are executed "in-memory" via control signals manipulating qubits.

2.3.4 Key Quantum Physics Principles

To this point, the full capabilities of a quantum computer over a digital computer have not been fully discussed. There are several key properties of quantum physics that must be understood to comprehend how a quantum computer can perform calculations more efficiently than a digital computer. These properties are: superposition, measurement, entanglement, and the "No-Cloning Theorem".

2.3.4.1 Superposition

The first property of quantum physics that quantum computers leverage is superposition [18]. Superposition is the concept that a quantum object can exist in multiple states at the same time. In a quantum computing context, this means that the current state of a qubit is in both $|0\rangle$ & $|1\rangle$. Visually, an equal superposition between $|0\rangle$ and $|1\rangle$ is any state on the equator of the Bloch sphere, e.g. aligned with the X or Y axis. One example of an equal superposition is the state $|\Psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. [13]

This property has entered the popular imagination through the thought experiment of Schrödinger's cat [34]. In this thought experiment, a cat is placed in a box with a vial of cyanide, and a triggered hammer that will break the bottle of cyanide. The hammer

[13] It is possible to have unequal superpositions, where the state is not halfway between $|0\rangle$ & $|1\rangle$. For example, $|\Psi\rangle = \frac{1}{3}|0\rangle + \frac{2\sqrt{2}}{3}|1\rangle$ is a valid superposition (according to the definition of a qubit from Eqs. (2.1) to (2.3)), which is weighted closer to $|1\rangle$ than $|0\rangle$ and is thus more likely to be measured (projected) onto $|1\rangle$.

is triggered by a known quantum system, such as the decay of a radioactive atom with a known half-life duration. After waiting for one half-life, the atom is in a superposition between decayed and not decayed, and thus the cat is also in a superposition between dead and alive. But when the box is opened, we can determine which of the two outcomes has occurred: the cat is either dead or alive. The tale of Schrödinger's cat is also informative because it introduces the concept of measurement in quantum mechanics, which will be discussed in the next section.

2.3.4.2 Measurement

Measuring a quantum state is inherently different than measuring a classical, digital state. On a digital computer, you can check the value of a bit at any point, e.g. represented by a voltage, and if the value is 0_2 then you can assume that it has always been 0_2 since it was last updated. [14] This assumption for digital computers does break down when the act of measuring a register sufficiently that voltage (e.g. DRAM), but the general concept of measurements not corrupting the initial value still holds.

In the realm of Quantum Mechanics, the act of measurement can alter or "collapse" the underlying state. Intuitively, the act of measurement can be thought of as causing the state to lose its superposition, thus "choosing" which state that it will hold. The closer that the arbitrary quantum state $|\Psi\rangle$ is to $|0\rangle$, the more likely that it will choose $|0\rangle$ instead of $|1\rangle$, and vice versa. The probability of measuring $|1\rangle$ is $|\beta|^2$ in the qubit representation of Eq. (2.1). This can be seen on the Bloch sphere: a state perfectly situated on the equator (e.g. $+X$) is equidistant from $|0\rangle$ and $|1\rangle$, so it is equally likely to be measured in either

[14] Ignoring noise that could flip the bit, which is rare.

state.

2.3.4.3 No-Cloning Theorem

In quantum mechanics (and quantum information), the *no-cloning theorem* shows that it is impossible to perfectly clone (copy) an arbitrary quantum state [35].

To prove that cloning is impossible: suppose that there exists a unitary cloning operator U_{clone} that can clone an arbitrary quantum state ($|\Psi\rangle$, Eq. (2.1)). This U_{clone} should operate to transform a single $|\Psi\rangle$ to $|\Psi\rangle|\Psi\rangle$, e.g. $U_{clone}\,|\Psi\rangle\,|0\rangle \;\rightarrow\; |\Psi\rangle\,|\Psi\rangle$. Further, suppose that we have two arbitrary quantum states that we would like to clone: $|\phi\rangle\,, |\psi\rangle$. If we apply the supposed cloning unitary U_{clone} to each of these states, we will yield:

$$|\phi\rangle \otimes |0\rangle \;\rightarrow |\phi\rangle \otimes |\phi\rangle \tag{2.8}$$

$$|\psi\rangle \otimes |0\rangle \;\rightarrow |\psi\rangle \otimes |\psi\rangle \tag{2.9}$$

If we consider the inner-product[15] of the arbitrary states, using the known inner-product $\langle 0|0\rangle = 1$, then

$$\langle\phi|\psi\rangle = \langle\phi|\psi\rangle\,\langle 0|0\rangle = \langle\phi|\,\langle 0|\,|\psi\rangle\,|0\rangle \tag{2.10}$$

Then applying the cloning operator U_{clone}, which is unitary and thus norm-preserving &

[15]The inner-product is a generalized form of the dot-product. Note that we drop the $\otimes$ notation because it is redundant and hinders readability.

will not change the inner-product, we obtain the inner-product:[16]

$$\left(\langle\phi|\langle 0|U_{clone}^{\dagger}\right)\left(U_{clone}|\psi\rangle|0\rangle\right) = \langle\phi|\langle\phi||\psi\rangle|\psi\rangle = |\langle\phi|\psi\rangle|^2 \tag{2.11}$$

$$\langle\phi|\psi\rangle = \langle\phi|\psi\rangle^2 \tag{2.12}$$

The result of this proof, shown in Eq. (2.12), is only satisfied for arbitrary states $|\psi\rangle, |\phi\rangle$ which produce an inner-product magnitude $|\langle\phi|\psi\rangle|$ of $0, 1$. Thus, cloning is **not** possible for arbitrary states, and is only possible for states that are either identical, anti-aligned, or orthogonal to begin with. Therefore U_{clone} cannot exist for cloning arbitrary states, i.e. ones that do not satisfy Eq. (2.12). Note that because it is possible to clone such special, non-arbitrary states, it is possible to devise a system to e.g. clone qubits that are only in $|0\rangle$ or $|1\rangle$.

The no-cloning theorem does not solely apply to qubit states: it is also not possible to clone an arbitrary mixed state [36]. However it is possible to *closely* clone some input states [37], with the condition that the cloning fidelity will depend on the input state.

The result of the no-cloning theorem is that many of the operations that are taken for granted in a digital computer are made significantly more difficult, or impossible. A simple example is backing up data on a computer. This operation is trivial on a classical PC: data is read from a storage drive, written to a second drive, and there now exists two copies of the same data. On a quantum computer, this operation can never be performed due to the no-cloning theorem. The best that can be done is *approximately* replicating the original datum. Worse, according to [37], even if an approximate copy is made, the

[16]Note that we apply the Hermitian conjugate of the cloning operator ($U_{clone}^{\dagger}$) because it is applying to the *bra* states $\langle\psi|\langle 0|$.

copy is entangled with the original, and then any operation or measurement performed on the copy can interfere with the state of the original, which is obviously not desired in a backup situati8on.

While this is a somewhat trivial example, and it could be argued that the need for backing up quantum information is relatively far off, it nevertheless illustrates a fundamental difference between classical and quantum computers. This fundamental difference will need to be accounted for at all levels of the computer architecture design, and have a net effect of complicating the design of a quantum computer by requiring high-fidelity long-term storage for future computations.

2.3.4.4 Entanglement

The principle of quantum entanglement is that there can be a correlation in the joint quantum system formed by multiple qubits. In a perfect world, every qubit begins as completely disentangled with every other qubit, i.e. independent. When a proper sequence of operations is applied to at least two qubits, the qubits can become entangled and begin sharing correlations. Once they are fully entangled, measuring one of the qubits' values will allow you to know the value of the other qubit by way of correlation.

For example, consider the following entangled state, which is a superposition between both qubits being in $|0\rangle$ and both qubits being in $|1\rangle$: $|\Psi\rangle = |00\rangle + |11\rangle$ (discarding normalization factors). If one of the two qubits was to be measured, and found in $|1\rangle$, then we would know that the other qubit was *also* in $|1\rangle$ (and vice versa for $|0\rangle$).

2.3.4.5 Reversibility

It is interesting to note that quantum computing is closely aligned with the concept of reversible logic. Nielsen and Chuang [18, Section 3.2.5] provides a good overview:

> First, reversibility stems from keeping track of every bit of information; irreversibility occurs only when information is lost or erased. Second, by doing computation reversibly, we obviate the need for energy expenditure during computation. All computations can be done, in principle, for zero cost in energy. Third, reversible computation can be done efficiently, without the production of garbage bits whose value depends on the input to the computation. That is, if there is an irreversible circuit computing a function f, then there is an efficient simulation of this circuit with action $(x, y) \rightarrow (x, y \oplus f(x))$. ([18], pg. 161)

An important implication of this is:

> To harness the full power of quantum computation, any classical subroutines in a quantum computation must be performed reversibly and without the production of garbage bits depending on the classical input. ([18], pg. 161)

In other words, performing any classical operation on a quantum state, such as adding 1 to a number stored in a quantum register, will require the use of reversible logic gates, such as the Toffoli gate [38]. We will discuss an improved Toffoli gate implementation in Section 7.2.

2.4 Quantum Operations

A quantum computer must be able to perform an operation to be useful. This section will discuss some of the fundamental operations that must be implemented for full control of a quantum computer, as laid out by Section 2.2.1.

2.4.1 State Preparation

The first building block of a quantum computer is implementing state preparation. This is the ability to reliably prepare a known quantum state. One of the most typical states to prepare is $|0\rangle$, or for multiple qubits $|00\ldots0\rangle$.

This prepared state then becomes the input state into a quantum circuit, or a set of ensuing operations. By convention, if an initial state other than $|0\rangle$ is desired, qubits are first prepared in $|0\rangle$, and then rotations (Section 2.4.2) are applied to achieve a different desired initial state, e.g. $|1\rangle$. [17]

It is important to note that multi-qubit entangled states will sometimes be required, such as a logical qubit basis state in [39]. In this case, this state can generally be achieved by preparing in $|0\rangle$, and then applying not just single-qubit operations, but a mix of single- and multi-qubit operations.

[17] In this scheme, because a subsequent operation is applied to enter $|1\rangle$ after starting in $|0\rangle$, preparing a qubit in $|1\rangle$ will have lower fidelity than $|0\rangle$.

2.4.2 Single-Qubit Operations

Single-qubit operations are operations that only address, or change the state of, a single qubit. The most basic of single-qubit operations is applying any of the Pauli rotation matrices, to rotate a qubit around a given axis (see Table 2.2).

By convention, a gate that applies a continuous version of a Pauli matrix will be denoted as (letting σ be an arbitrary Pauli matrix) [17]:

$$R_\sigma(\theta) \equiv \exp\left(\frac{-i\theta\sigma}{2}\right) \tag{2.13}$$

A common notation is to abbreviate $R_\sigma(\pi)$ as the equivalent Pauli matrix, e.g. $R_Z(\pi) = Z$, $R_X(\pi) = X$, $R_Y(\pi) = Y$.

All single-qubit operations can also be expressed as 2×2 matrices (unitaries). A single-qubit operation can be applied to any given qubit by taking the tensor product ($\otimes$). [18] For example, the single-qubit X gate can be applied to the second qubit (of two qubits) with:

$$I \otimes X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.14}$$

Another common single-qubit gate is the Hadamard gate. The Hadamard gate takes the computational states $|0\rangle$, $|1\rangle$ into a superposition: $H|0\rangle \rightarrow \frac{|0\rangle+|1\rangle}{\sqrt{2}}$, $H|1\rangle \rightarrow \frac{|0\rangle-|1\rangle}{\sqrt{2}}$, which are commonly abbreviated as $|+\rangle$ and $|-\rangle$, respectively. The Hadamard gate has the

[18] This tensor product operator is sometimes also called the Kronecker Product.

interesting property of being its own inverse ($HH^{-1} = HH = I$). It can be decomposed into Pauli rotations: $H = XR_Y(\pi/2)$. It is important to note that any single-qubit gate can be decomposed into a combination of continuous Pauli rotations.

2.4.3 Multi-Qubit Operations

Multi-qubit operations are operations that are applied to multiple qubits at the same time. Generally, applying a multiple qubit gate will produce entanglement, though it can also *dis*entangle qubits. [19] With a combination of single-qubit and two-qubit operations, an arbitrary multi-qubit gate can be realized [40].

The basic two-qubit gate is the CNOT gate, which stands for Controlled-NOT. This gate applies a NOT (X) gate to the *target* qubit if the *control* qubit is in $|1\rangle$. See an example CNOT gate in Fig. 2.3; the filled circle is the *control* qubit, and the qubit with the circle with the plus ($\oplus$) is the *target* qubit. The CNOT gate is represented in matrix form by:

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.15}$$

A two-qubit entangling gate such as a CNOT is required for universal quantum computation [18]. However, multi-qubit gates are not restricted to only operating on two qubits. Conceptually, if a quantum state will be shared between N qubits, it is more

[19] An example of a multi-qubit gate that produces *no* entanglement is the SWAP gate, where the quantum state of two qubits is swapped between the two.

efficient (in time and resources) to generate that entanglement using a N-qubit gate than using an equivalent sequence of 2-qubit gates (which produces the same unitary matrix).

Other typical quantum gates include:

- *Controlled-U gate*: applies a unitary U to the target qubit if the control qubit is in $|1\rangle$.
- *Toffoli Gate*: the basic 3-qubit version, briefly covered in Section 2.3.4.5, is effectively a CNOT with two controls instead of one (i.e. a Controlled-Controlled-NOT, or CCNOT), and flips the target qubit if and only if the two control bits are both in $|1\rangle$.

2.4.4 Quantum State Readout

Measurement is a fundamentally important concept in quantum computing, and was briefly discussed in Section 2.3.4.2. It deserves special care because quantum measurement is conceptually very different than in classical computing.

Measurement is the act of resolving the quantum state of a qubit, and the corresponding graphic signifying measurement is shown in Fig. 2.2. In contrast to classical computing, quantum measurement will generally irreversibly change the state of the qubit by collapsing any superpositions in the measured qubit(s). Measurement is sometimes referred to as *projective*, meaning that the precise state $|\Psi\rangle$ is not measured, but only its projection along a given axis (eigenvector).

It should be noted that measurement is a *statistical* process. One measurement along the Z-basis of a qubit will only determine if the qubit was in $|0\rangle$ or $|1\rangle$ for that

Figure 2.2: **Trivial measurement of a qubit.** This figure shows the symbol for a measurement operation on a qubit. Measurement symbols that are *not* shown at the end of a circuit are typically implied.

particular experiment run. Repeated measurements are necessary to approximate the pre-measurement quantum state, which then requires balancing the number of repetitions against approximation accuracy.

There are certain cases where the act of measurement does not completely resolve the state of the entire system, but instead fully resolves the state of certain qubits in the system (i.e. $N_{measured} < N_{qubits}$), which we here refer to as "partial measurement". In certain applications, such as quantum teleportation and quantum error correction, measurements are performed on an ancilla qubit, which has certain information *about* the quantum state mapped to it, but not the identity of the quantum state itself. This "partial measurement" is in contrast to a "weak" or "imperfect" measurement, where the quantum state is minimally modified, and little information about the quantum state is obtained in turn.

2.5 Quantum Algorithm Abstractions

Generally, executing an algorithm on a quantum computer requires some form of abstraction to represent the sequence of operations that are being performed. This abstraction can be at varying levels depending on the overall goal. Just as computer scientists do not specify data structures and algorithms in terms of transistor voltages, it can be an inappropriate level of abstraction to specify a given quantum algorithm in terms of the RF

drive on the given quantum system needed to perform an operation. This tradeoff, as well as some benefits that come from breaking these abstractions, will be discussed further in Chapter 3 and Chapter 4.

In the context of this thesis, there are a few levels of abstraction that can describe a sequence of operations on a quantum computer. In order from higher level of abstraction (higher abstraction here implies least knowledge of the quantum computing hardware and the quantum physics Hamiltonian that describes it) to lower level of abstraction: Quantum Algorithm, Quantum Circuit, Unitary, Pulse Schedules and Hamiltonian. We will not fully treat each of these abstractions, but it is important to understand that each of these methods are fully valid and equivalent methods for describing the same sequence of operations. However, the highest levels of abstraction are also the least precise, as they tend to make simplifying assumptions about the underlying system. The most-precise level is the lowest level, but this precision includes a higher level of complexity, which is especially true as the scope of the system that is being considered increases.

For example, if you want the most accurate simulation of a quantum computer, you will try to incorporate as many physical and environmental parameters into your simulation as possible, but this will add complexity to the simulation. Eventually, as more and more factors are incorporated, or the size of the quantum system grows enough, the simulation will become infeasible (for a reasonable amount of computing power). On the other hand, a highly abstract quantum algorithm specification (such as [41, 42]) will tend to ignore the implementation details of how a particular operation is implemented, and just specify what the end result is. This allows the algorithm to be generally system-independent, to scale to any given system size, as well as be more easily simulated be-

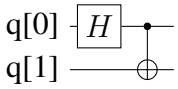
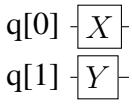
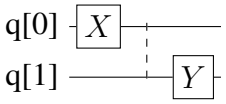


Figure 2.3: **Quantum circuit diagram of a Bell circuit.** A Bell circuit is a simple circuit that generates a *Bell state* [18], which are maximally entangled two-qubit states. The first gate on $q[0]$ is called a Hadamard gate (Section 2.4.2), and effectively puts the qubit in superposition between $|0\rangle$ & $|1\rangle$. The second gate is a CNOT (Controlled-NOT or Controlled-X) gate, and entangles the two qubits.



(a) Circuit with implied parallel operations



(b) Circuit with guaranteed sequential operations

Figure 2.4: **Quantum circuit diagram with parallel operations.** Fig. 2.4a demonstrates parallel single-qubit operations on two qubits. The quantum circuit does not necessarily *guarantee* that these operations will happen simultaneously, it just allows that possibility. In other words, it would be equally valid to perform the X gate followed by the Y gate, or the Y gate followed by the X gate, or both simultaneously. Alternately, sequential operations can be enforced using devices such as a "barrier", shown in Fig. 2.4b,

cause it has removed complexity.

2.5.1 Quantum Circuit Model

The quantum circuit model lies between the two ends of the spectrum of quantum abstractions: partially between high-level quantum algorithms (a computer science approach), and a low-level Hamiltonian specification (a quantum physics approach).

The concept of a quantum circuit is most analogous to a digital circuit diagram. For reference, an example quantum circuit diagram is shown in Fig. 2.3. Because these circuits will be used in several places throughout this thesis, we shall provide a brief overview of their operation and usage.

A quantum circuit is composed of *qubits*, *operations*, and *wires*. Viewing it as a se-

quence of inputs & outputs, a quantum circuit will generally begin with a qubit (typically with a label), which is prepared into a given state (typically $|0\rangle$, for reasons discussed in Section 2.4.1 and Section 3.6.2.2). Every qubit has a *wire* connected to it. A *wire* is a virtual operation, which conceptually connects the output of one circuit element to the next circuit element. Wires apply the *Identity* operation to a qubit, i.e. effectively no operation ("no-op"). [20] A wire then typically connects two operations. We will primarily consider two types of operations: gates, and measurement, which retain their meaning as previously discussed in Section 2.4.

This representation clearly shows the operations that will be executed on a quantum system. The quantum circuit is also useful for intuitively understanding the complexity of a given quantum algorithm: more qubits will require an exponentially larger state space, which is more difficult to simulate using classical computers, and physically requires control of many qubits. Further, adding more gates will add approximately linearly to the execution time of the circuit, whether in simulation or physical execution. Larger circuits, in terms of either number of qubits or number of gates, are thus more difficult to simulate classically and show the value of a physical quantum computer.

There are several additional features that add complexity to the basic quantum circuit, which are necessary for more advanced quantum applications:

- *Digital Bits*: These can represent either an arbitrary input classical register, or store the result of a measurement (and potentially use it later on in the quantum circuit). These are represented by two lines for a wire, instead of a single line for a quantum

[20] Some compilers will insert "echo" operations during wires/identity operations, to make the qubits more-insensitive to certain classes of noise sources.

wire.

- *Mid-Circuit Measurement*: This is the act of performing a measurement in the middle of a quantum circuit. In other words, it is performing a measurement while there are still subsequent operations on a qubit that will be executed after the measurement.

- *Conditional Operations*: These operations enable executing any other operation based on either a *classical* or *quantum* state, while maintaining the quantum state. The condition[21] is based on a set of digital bits, which can be sourced from a classical digital source (such as a random number generator), or as a direct result of a mid-circuit measurement. Together with mid-circuit measurement, these operations are necessary for critical quantum computing algorithms such as quantum error correction, or for studying quantum information concepts such as interactive protocols [43].

- *Mid-Circuit Reset*: This resets a qubit into a known state, effectively re-applying a preparation operation in the middle of a circuit. This can either be destructive (i.e. erasing the quantum state), or reversible by undoing any operations applied to the qubit.

- *Barriers*: These enforce serial operations, by defining certain points across which operations cannot be moved. Barriers exist because quantum circuits do not specify an order of operations, as they assume that all operations can happen simultaneously, as in an electrical circuit. This assumption does not always hold true in

[21] i.e. Whether or not the operation should be executed

practical quantum computers, typically due to either physics or control limitations.

2.5.1.1 Shortcomings of Quantum Circuits

Though quantum circuits may appear superficially similar to circuit diagrams, there are several important caveats to consider with their usage, which are rarely discussed.

- *Legibility at scale*: It is difficult to comprehend a large-scale quantum circuit, e.g. one comprising hundreds of operations on dozens of qubits. This is not a unique problem to quantum circuits: a digital scale with a similar number of operations is just as difficult to visualize and comprehend. In this case, I suggest that the quantum community borrow from the experience of electrical engineering, which is accustomed to similarly large-scale systems. Electrical engineering has developed hierarchical schematics, where an overall circuit schematic is broken down into sub-blocks, each of which are logically grouped and somewhat independent. For example, a logical qubit circuit could have one block for entangling qubits, another for operating on the qubits, and then another for ancilla measurement & correction. Each of these blocks would have a circuit illustrating it, and then a high-level block diagram showing the ordering of the logical blocks.

- *Error Diagnosis*: Related to the previous point, there is no simple way to easily identify a simple transcription error when defining a circuit. For example, consider a human copying a textbook circuit, who mistakenly swaps an X gate for a Y gate on a single qubit, but has the remaining qubits correct. Other than painstakingly manually double-checking the circuit, or debugging it, there currently exists no sort

of automated tool for checking that the circuit matches expectations.

- *Parallelism*: As previously described, quantum circuits assume that every operation can be parallelized (executed simultaneously). This assumption does not always hold. This conceptual mismatch is due to the difference in nature between electrical circuits and a quantum circuit: an electrical (logic) gate such as AND is effectively a passive device, in that once properly designed and set up it will continue to perform its operation with no external control. The same is not true of quantum computers, which tend to require a drive signal to perform an operation (discussed later in Section 3.7).

- *Entanglement Visualization*: As previously discussed, quantum entanglement is generally a desired quality in a circuit, but it is also reversible (akin to how entanglement is generated in a circuit). Certain classes of circuits will entangle qubits, perform some operation, and then disentangle the qubits. However, in a long quantum circuit, the entangling operation might occur at the beginning of the circuit, but the entangled qubits might not be used until the end. This problem is unique to quantum circuits, as opposed to a digital circuit where correlations only last for the duration of a gate, primarily due to the no-cloning theorem (Section 2.3.4.3), so to track the entire state of the qubit, you must track the history of all operations on that qubit back to the initial state. Currently, to track the state of entanglement between multiple qubits, you must be able to perform linear algebra calculations in your head (or use some level of intuition), but entirely without any guide markings in the quantum circuit as to the creator's intentions.

- *In-Circuit Commentary*: Currently there is no method for annotating a quantum circuit without accompanying external text. By analogy to a musical score, a composer annotates information such as volume outside of the sequence of notes themselves. However, none of this is available in standard quantum circuits. They do offer the concept of a barrier, which can logically divide parts of a circuit in time, but it is difficult to ascertain the intention behind the circuit without access to either accompanying text or the software that generated the quantum circuit (which is presumably annotated/well-named). Again by analogy, the equivalent would be an electrical circuit schematic that did not allow including any text comments or references to outside information.

2.6 Breaking Abstractions: Hardware Details

So far, this chapter has addressed the topic of quantum information very generally, with very little discussion of physical quantum hardware. We will continue to leave the majority of that discussion for later chapters, but there are a few high-level considerations that are relevant in a purely quantum information context. These are slight permutations to the concepts already introduced: circuits and gates.

2.6.1 Qubit labels: Physical vs Virtual

In a typical quantum circuit abstraction, every qubit is treated identically. This is a reasonable abstraction; but to actually execute the circuit, the information associated with every qubit in the circuit must be mapped to some physical entity. A useful way of thinking

about this mapping is in terms of *physical* and *virtual* qubits.

A *virtual* qubit is simply a label when designing a quantum circuit. At the most basic level, it is simply an index that denotes which qubit that you are performing operations on. The assumption that most quantum circuit designers work with is that every qubit in their circuit operates identically. In this case, "identical" means that they perform similarly in terms of quality (usually denoted by fidelity in quantum systems), and connectivity (discussed in the next section, Section 2.6.2).

By contrast, a *physical* qubit corresponds to a physical qubit implementation, two of which will be discussed in Section 4.3. The number of physical qubits in a system is typically a design parameter when the quantum computer is designed, so it is relatively constant (this is not quite true of trapped ion systems, as will be discussed in Chapter 3). Suppose that you have five physical qubits in a system. Each of those qubits inhabits some physical space that is inherently, even if minutely, different from the other qubits. Because qubits, despite our best efforts, are coupled to (interact with) their environment, this means that each qubit is effectively unique. If we were operating in a digital realm, this inconvenient fact could be somewhat abstracted away with thresholding, but every minute change matters in today's quantum computer. [22]

Thus, we have two different conventions for considering a qubit: *virtual*, which

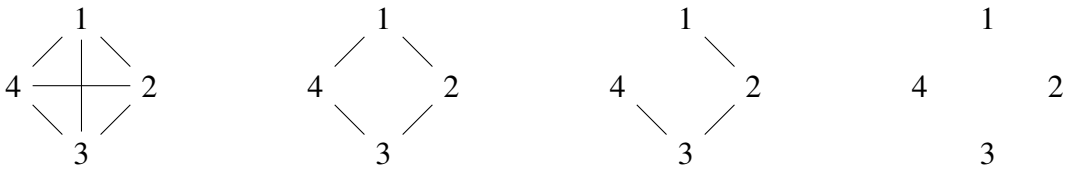
[22] For an example of digital thresholding, consider a transistor that switches at $\geq 2.7\,\mathrm{V}$, but accepts input voltages in the range $2.7\,\mathrm{V}$ to $3.3\,\mathrm{V}$. Any voltage in the range $2.7\,\mathrm{V}$ to $3.3\,\mathrm{V}$ will be treated as 1_2, while any voltage in the range $0.0\,\mathrm{V}$ to $0.6\,\mathrm{V}$ will be treated as 0_2, and behavior in between is undefined.

It is interesting to consider that thresholding will likely apply to quantum computers as they reach the scale of logical qubit error correction. Error correcting codes have a fidelity threshold, below which they do not improve on the underlying qubits' native fidelity. If the logical qubit operations are of sufficient quality, i.e. above the threshold, then error-corrected qubits will have similar effects to digital. The precise threshold is very dependent on system and error correcting code specifics, and will not be discussed further here.

treats every qubit identically, and *physical* which must consider the complexities of a device in the real world. For productivity on the side of both the circuit designer and the system designer, it is necessary to have these two abstractions. By analogy, most traditional software programmers do not typically concern themselves with exactly what model of CPU that their software will run on, or which transistors a particular block of code operates on. Those details can be handled by either the CPU itself or the compiler. Requiring every programmer to understand them would simply be inefficient. Thus, a method of mapping between the abstract and concrete representations is required.

This process for quantum computers is typically called *qubit mapping*, or *qubit layout*. Qubit mapping is the process of converting between a virtual qubit representation of a circuit to a physical qubit representation.

To explain this process, consider the example quantum circuit in Fig. 2.3. This circuit is specified for 2 qubits, but it could be run on a quantum computer with ≥ 2 qubits. Suppose that the physical qubits have indices $\{0, 1, ..., N\}$. The simplest mapping would simply be to assign the two qubits in order to the first two indices: so $q_0 \rightarrow 0$, $q_1 \rightarrow 1$. While that mapping might be technically correct, it fails to take into account any information about the *physical* qubits. For example, suppose that two-qubit gates were not possible between qubits 0 and 1, or that the two-qubit gate between them was of comparatively low quality. Both of these scenarios would make a different qubit mapping optimal. It should be clear that it is an optimization problem to assign the virtual to physical mapping in the most efficient manner.



(a) Complete graph. Also called fully-connected.

(b) Cycle graph.

(c) Linear graph.

(d) Disconnected graph.

Figure 2.5: **Example Qubit Connectivity Graphs for $N_{qubits} = 4$.** There are a limited amount of graphs that can be demonstrated with only 4 nodes, but this illustrates a few of the major classes.
Fig. 2.5a is the closest to the connectivity of an ion trap quantum computer. Qubits with a physical 2D (planar) qubit layout have difficulty replicating this connectivity because of the crossing wires, e.g. overlapping edges between $(2, 4)$ & $(1, 3)$. Difficulty can also arise if control signals need to be routed to the inner connections ($(2, 4)$ & $(1, 3)$), as in superconducting qubits, because a common way of performing multi-qubit gates is using a tunable coupler which must be enabled to perform a gate [44].[23]
Fig. 2.5b will easily map to a planar geometry, but at the cost of not being able to perform gates between distant nodes, which forces information to be moved around for interactions to occur.
Fig. 2.5c is very simple to implement, but may have very limited practical capability due to the limited connectivity.[24]
Fig. 2.5d is almost entirely useless for quantum circuits because it does not support multi-qubit gates, but might still be useful for test and characterization purposes, assuming that single-qubit operations can still be performed.
Many other connectivity graphs are considered in quantum computing, including Chimera [45], Heavy-Hexagon [46, 47], as well as hypercubes and cyclic butterfly layouts [48].

2.6.2 Qubit Connectivity

In the previous section, we considered that every *physical* qubit is not identical. One of the common ways that physical qubits are not identical is in their connectivity.

Every quantum computer has a connectivity graph. In the connectivity graph, the *nodes/vertices* are the qubits, and an *edge* is drawn between any qubits that can perform

[24] For this particular number of nodes, this graph could still be mapped to a planar architecture by routing one of the edges around the outside, but that becomes either more difficult or more complex at higher numbers of qubits.

[24] Practically, this type of architecture tends to require many SWAP gates to move information to the appropriate location to perform a gate on it.

a direct two-qubit gate (e.g. a CNOT gate) between the two qubits. Single-qubit gate connectivity[25] is typically not drawn, because it is assumed that a single-qubit gate can be performed on any qubit. [26] Thus, the connectivity of a quantum computer is effectively the set of two-qubit gates that can be performed. Some example connectivity graphs for small numbers of qubits are demonstrated in Fig. 2.5.

Another layer of complexity is the quality of the quantum gate. Quantum computers today are operating at the limits of their performance, and as such any tiny variation (in manufacturing, control signals, environment, etc.) can cause performance fluctuations. Thus, the fidelity (quality) of a gate between e.g. Qubits $(0, 1)$ might be better than between Qubits $(1, 2)$ at the moment, but that might change in a few hours. So optimal performance when executing a circuit relies on assigning qubits with knowledge of connectivity, as well as weighting connectivity by the quality of the gates. This is an active area of research, and there are many approaches to solving this problem [49, 50, 51, 52, 53, 54, 55, 56].

2.6.3 Native Gate Set

For ease of communication, many quantum algorithms are designed assuming a standard library of quantum operations, such as the Clifford group (consisting of CNOT, Hadamard, Pauli matrices, and the S (Phase) gate) [57] plus the T ($\frac{\pi}{8}$) gate. Using the Solovay-Kitaev Theorem [58, 59], this set of "universal" operations can be used to construct any arbitrary quantum state efficiently.

[25] i.e. a looping edge from a node to the same node

[26] This assumption does not always hold true when some qubits become unaddressable for various reasons, such as the unusable superconducting qubits in Google's Sycamore processor [31, Fig. 2].

However, no currently functioning quantum computer natively implements all of these operations, due to details in the Hamiltonian that each quantum computer implements. Instead, each quantum computing technology implements some other set of gates, which we call "native" to that technology family. A properly-chosen set of native gates will be universal, and thus able to implement every possible quantum operation.

The disconnect between native operation implementation and high-level abstractions of those operations is actually not unique to quantum computers. Modern digital computers are typically implemented using CMOS transistors, whose native gate is a NAND (AND $\rightarrow$ NOT) operation. Fortunately, the NAND gate is a universal gate, and so any logic circuit that is specified using e.g. AND or OR gates can be mapped to native NAND circuitry.

This similarity between native quantum and digital gates even carries over to the need for post-mapping optimization once a standard set of operations is mapped to the native gate set. The issue is that once the mapping to the native gate set is complete, there will typically be redundant native gate operations that either perform no operation, or are cancelled by some other operation.

2.7 Alternative Quantum Computing Paradigms

This chapter has primarily treated quantum information in terms of gate-model quantum computing. Gate-model quantum computing is just one of several quantum computing models. Some other models include Quantum Simulation [22, 60], and Adiabatic Quantum Computing (AQC) [61, 62, 63, 64]. Both of these are rich fields that will not be fully

treated here, though a general overview of each model is useful for the sake of comparison. Before covering each sub-field individually, it is important to note that theorists have determined that each quantum model (gate-model, quantum simulation, and adiabatic quantum simulation) are equivalent, in that theoretically they can all produce the same results [62]. However, just as processors vary by application[27], each computational model has its own tradeoffs. A specific quantum computing system will be normally optimized for one application (model) or another.

2.7.1 Quantum Simulation vs Gate-Model

The goal of quantum simulation is to model the behavior of some physical phenomenon using the quantum behavior of a quantum computer. Quantum simulation aims to broadly simulate the dynamics of a system. This should allow simulating either dynamics or systems that cannot currently be directly observed, such as high-temperature superconductors [22]. This model differs from gate-model quantum computing because it generally is not aiming to precisely reproduce **every** Hamiltonian or Unitary, only a particular Hamiltonian, and generally can accept some level of error. Gate model quantum computing typically expects that every operation is as precise as possible, and any errors can be catastrophic to the result of the current circuit execution. The reliance on precision makes gate-model quantum computing extremely difficult. By analogy to electrical engineering, imagine if every transistor in a computer would only work as expected if it received a precise voltage signal (i.e. if it was operated in the linear regime instead of at saturation).

[27] For example, the processor which controls a microwave is not the same processor that controls a personal computer.

In this scenario, if the voltage was off by even 0.1% (due to cosmic rays [65, 66], poor voltage regulation, external temperature drifts, etc.), the transistor would give an inconsistent result. A system with this fatal flaw would obviously have difficulty scaling to include the billions of transistors that today's modern CPUs have.

Quantum simulators are making progress in simulating interesting physics. More information on quantum simulation can be found in [67, 22, 68, 69, 70, 60, 71, 72, 73, 74, 75, 76, 77].

2.7.2 Adiabatic Quantum Computing vs Gate-Model

Adiabatic Quantum Computing (AQC) is yet another model of quantum computing, which is primarily considered for solving optimization problems [78, 79]. An optimization problem is a class of problem where some target function is maximized (or alternately, minimized), with optional constraints on the input values. For example, consider an example function shown in Fig. 2.6. The goal of an *optimizer* is to find the best solution, typically by finding the minimum of a *cost function*. In other words, the best solution to an optimization problem is the solution with minimal cost. If the cost is plotted as a function of the input variables, the result is a plot somewhat like Fig. 2.6, but potentially with more dimensions.

However, it is very computationally expensive to solve optimization problems [78], and there can be many local minima that hinder finding the globally optimal solution. For example, consider trivial optimization solver that starts at $x = 0.0$, and continues searching along the line as long as the slope is negative. For the example of Fig. 2.6, this

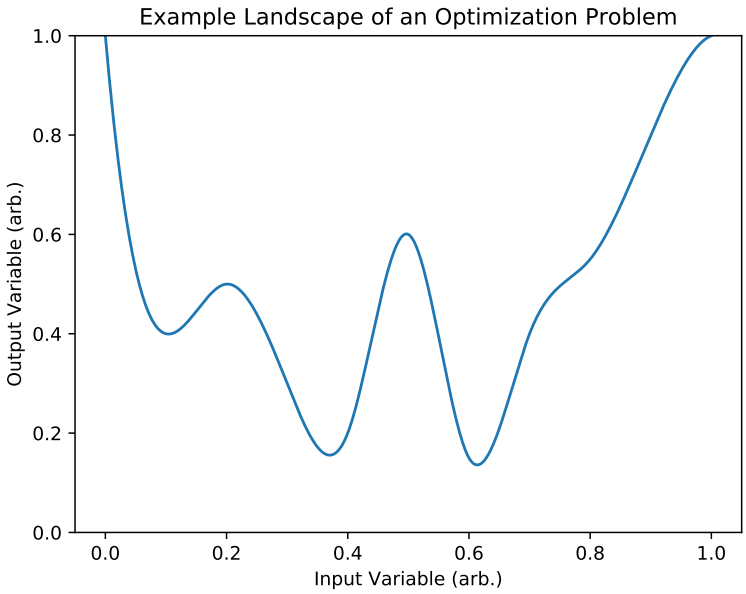


Figure 2.6: **Example Landscape of an Optimization Problem.** In this simple example, there is only one independent variable (variable that can be changed), and there are many local minima. The global minimum is $x \approx 0.6$.

would stop around $x = 0.1$, and never find the globally optimal solution around $x = 0.6$, let alone a near-optimal solution around $x = 0.4$.

The concept behind AQC as applied to optimization problems is that it can take advantage of quantum effects to "tunnel" through large barriers in the cost function (such as the one around $x = 0.5$ in Fig. 2.6) to find the optimal solution. Conceptually, AQC achieves this by initializing a quantum bit in the ground state of that system (e.g. $|0000\rangle$). Then, it slowly (*adiabatically*) applies a Hamiltonian corresponding to the problem that you would like to solve. If the transition is applied slowly enough, then the quantum register should remain in the ground state of the currently-applied Hamiltonian for the entire duration. Thus, the ground state at the end of the transition should be the ground state of the problem Hamiltonian, and the optimal solution has been found.

While this approach might be useful for certain types of problems, it has a few main

drawbacks:

1. *Classical Competition*: For a quantum optimizer using adiabatic computing to be useful, it must compete against the best algorithms and processors that can be designed for classical computation. Classical optimizers are currently improving by using techniques learned from quantum algorithms [80, 81, 82, 83].

2. *Near-Optimal Classical Solutions*: Further, classical optimizers are generally very good at getting close to the optimal solution, if not the exactly correct solution. Thus a quantum optimizer can only win if its solution is better (which has relatively small margins against classical optimizers), or if its solution is faster to compute.

3. *Problem size*: Generally, state-of-the-art quantum computing systems are very small today. Thus, the problems that they can solve are also small. This restricts a quantum optimizer to only solving small-scale problems (for the near-term), which are more difficult to out-compete against classical optimizer algorithms. However, this problem is not unique to AQC, as current gate-based quantum computers encounter a similar problem.

4. *Language/Education*: Most people who are attempting to solve optimization problems do not have a background in quantum mechanics, or even know how to begin describing their optimization problem for an Adiabatic Quantum Computer.

5. *Problem Mapping*: Even after identifying a suitable problem for optimization using quantum annealing, the problem still needs to be mapped to a QUBO (Quadratic Unconstrained Binary Optimization), and then to the available hardware [84]. [28]

[28] This process of mapping to hardware, called *embedding*, is an optimization problem in itself.

Chapter 3: Ion Trap Quantum Computing

3.1 What is a Trapped Ion Quantum Computer?

A trapped ion quantum computer is one leading technology that implements the concept of a qubit as a physical quantum device. In a trapped ion quantum computer, each qubit is represented by the energy levels of a single charged atom (i.e. ion). One or more ions are then trapped via an RF potential in an RF Paul trap [85]. The energy levels of the atoms are manipulated using atomic physics principles to perform the various operations necessary for a quantum computer, ranging from initializing in the ground state to performing coherent quantum operations.

The above is generically true of most trapped ion quantum computers. The following sections will delve into the specifics of our experimental setup.

3.2 Ytterbium Ions

While every element from the periodic table can become charged, there are certain atoms that are particularly advantageous for quantum computing. Generally, this class of atoms has the property that when it is charged by removing an electron from its outermost orbital (i.e. ionizing it), it will have low-lying energy levels similar to that of Hydrogen. This

class of Hydrogen-like ions is useful because these ions have a cycling optical transition, which aids in state preparation and detection. [1]

We have chosen to do the majority of our research work with an isotope of the element Ytterbium (abbreviated as Yb), with atomic mass (combined number of protons and neutrons) of 171. This isotope, when positively charged, will henceforth be shown as $^{171}Yb^{+}$. [2] Specifically, $^{171}Yb^{+}$ has the following desirable properties:

- *Hydrogen-Like*: When the atom is ionized, $^{171}Yb^{+}$ will behave similarly to Hydrogen because it only has a single valence[3] electron.

- *Clock States*: $^{171}Yb^{+}$'s nucleus has nuclear spin of spin-1/2. For our purposes, that means that $^{171}Yb^{+}$ has easily accessible "clock states" which are first-order magnetic field-insensitive. [4] This reduces their sensitivity to noise from fluctuating magnetic fields.

- *Reasonable Transition Wavelengths*: Performing quantum operations on trapped ion qubits typically requires using lasers, though research is being performed to minimize the need for these [86, 87, 88]. The goal of performing quantum operations quickly & at high fidelity requires precise [5] & high-power lasers at the desired transition frequencies[6]. For a variety of reasons, the desired quality of lasers are not

[1] Cycling transitions are useful for emitting many photons (light particles/waves) without decaying to undesired states. This allows high fidelity state detection even when collection optics only collect a small fraction of the scattered photons. Measuring the qubit state (e.g. $|0\rangle$ or $|1\rangle$) via a cycling transition allows determining the quantum state (state detection) with high-fidelity by the presence or absence of photons during a collection window.

[2] Sometimes this isotope of neutral Ytterbium is abbreviated as Yb-171.

[3] Outermost-shell

[4] When a field of a few Gauss is applied, a small linear field shift can be observed, so $^{171}Yb^{+}$ is not perfectly magnetic field-insensitive.

[5] e.g. Narrow linewidth.

[6] Equivalently, wavelengths.

available at every optical wavelength. However, $^{171}Yb^{+}$ has an advantageous set of wavelengths (described in Section 3.4) with lasers & optics generally available.

- *Hyperfine Clock Transition*: This indicates that the $^{171}Yb^{+}$ qubit is generally insensitive to its environment. In technical language, it has a very stable oscillation rate that can be first-order magnetic field-insensitive, so any quantum information that is stored in the qubit will remain there for long periods of time [89, 90].

- *Easy to source*: ^{171}Yb, though not the most-common naturally-occurring isotope of Ytterbium, can be easily sourced from various laboratories. Through these sources, we have procured isotopically-enriched samples of ^{171}Yb, which contains $> 95\,\%$ neutral ^{171}Yb, with the majority of the remainder being various other isotopes of Ytterbium.

- *Non-radioactive*: Isotopes of certain elements can be radioactive, which complicates their handling and increases the dangers of a laboratory environment. Certain other leading trapped ion systems are using radioactive ^{133}Ba [91]. [7]

3.3 Ion Trap Physical Hardware

In this section, we will provide an overview of the *EURIQA Breadboard Ion Trap Quantum Computer*. This is a single trapped ion quantum computing system that was designed, constructed, and operated by our academic laboratory.

Ion trap quantum computers are different from the transistor-based computers that

[7] Several Barium isotopes are radioactive. While ^{133}Ba is not used for medical imaging due to its long half-life, various other isotopes of Barium are used for medical radiological imaging [92].

Hardware Class	Device Specifics
Ion Trap	Sandia HOA 2.1.1
Atomic Sources	^{171}Yb & ^{138}Ba
Vacuum Chamber	Custom design [93, Sec. 4.1]
RF Resonator	Custom design [93, Sec. 4.2.2]
Global AOM Cell	L3Harris Single-Channel AOM
Individual AOM Cell	L3Harris 32-Channel AOM
Cooling/Pump Laser	Nichia $370\,\mathrm{nm}$ laser diode, various AOM/EOMs
Coherent Qubit Laser	Coherent $4\,\mathrm{W}$ Paladin Laser
Readout PMTs	Hamamatsu individual PMTs
High-NA Imaging Lens	Custom PhotonGear High-NA lens
Imaging Fiber	32-core custom fiber
Trap DAC	Custom SNL-designed 112-output DAC board
Trap DAC DC Filters	5th-order RC Lowpass Filter, $-3\,\mathrm{dB@} \approx 1\,\mathrm{kHz}$
Qubit Gate Drive	Xilinx ZCU111 RFSoC with custom software (SNL Octet)

Table 3.1: **Key Hardware Components of the EURIQA Breadboard Ion Trap System.** This is not a full bill of materials (BOM), but it includes many of the unique system components. More details can be found in [93, Ch. 4].

most people are familiar with. First, the "quantum computer" itself requires initialization other than providing power: an ion trap only becomes a quantum computer when it holds at least one ion, and when operations are performed on it (e.g. preparing the qubit into $|0\rangle$).

Second, the ion trap quantum computer physically looks very different than a traditional computer. Today, a standard processor core (CPU) is a processed semiconductor bonded to a package, connected to a motherboard that provides I/O, e.g. communication with a storage device. For ion trap quantum computers, the *core* of the system is the trapped ions themselves (I/O operations will be described in Section 3.7). These ions are suspended in space, held in place using a combination of static (DC) and oscillating electromagnetic (EM) fields at RF frequencies (usually called RF fields for simplicity). The RF signal is generated by an external RF source, and then routed to a device called the

ion trap via a resonator. [8]

3.3.1 Ion Trap

The ion trap is responsible for ensuring that the individual ions, once ionized and cooled, remain approximately in the correct position in space. There are many different configurations of these ion traps [95, 96]. The main types of RF Paul traps are blade (or rod) traps, and surface electrode traps. Both types of traps (see Fig. 3.1) need RF electrodes where the RF trapping fields are emitted. In a rod (blade) trap, the trapping electrodes are macroscopic rods (blades) typically arranged in approximately a rectangular configuration (Fig. 3.1a). The trapping region is then parallel to the rods (blades), along the line equidistant from all of the rods (blades). In a surface electrode trap (Fig. 3.1b), a similar trapping field is emitted from electrodes that are along a flat surface (like a PCB). The ion(s) are then pinned (trapped) floating above the flat surface, parallel to the RF electrodes/pads.

[8] Note that the exact amplitude and frequency of the RF signal depends on the mass of the ion, and the ion trap itself. It is also important that the RF voltage be held relatively constant, and treated with care. The RF source is driving a large amount of power to the ion trap, so catastrophic failures can be caused by either quick changes of this power/voltage or driving the voltage of an ion trap too high [94].

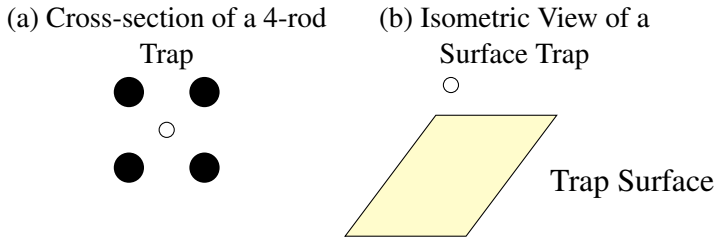
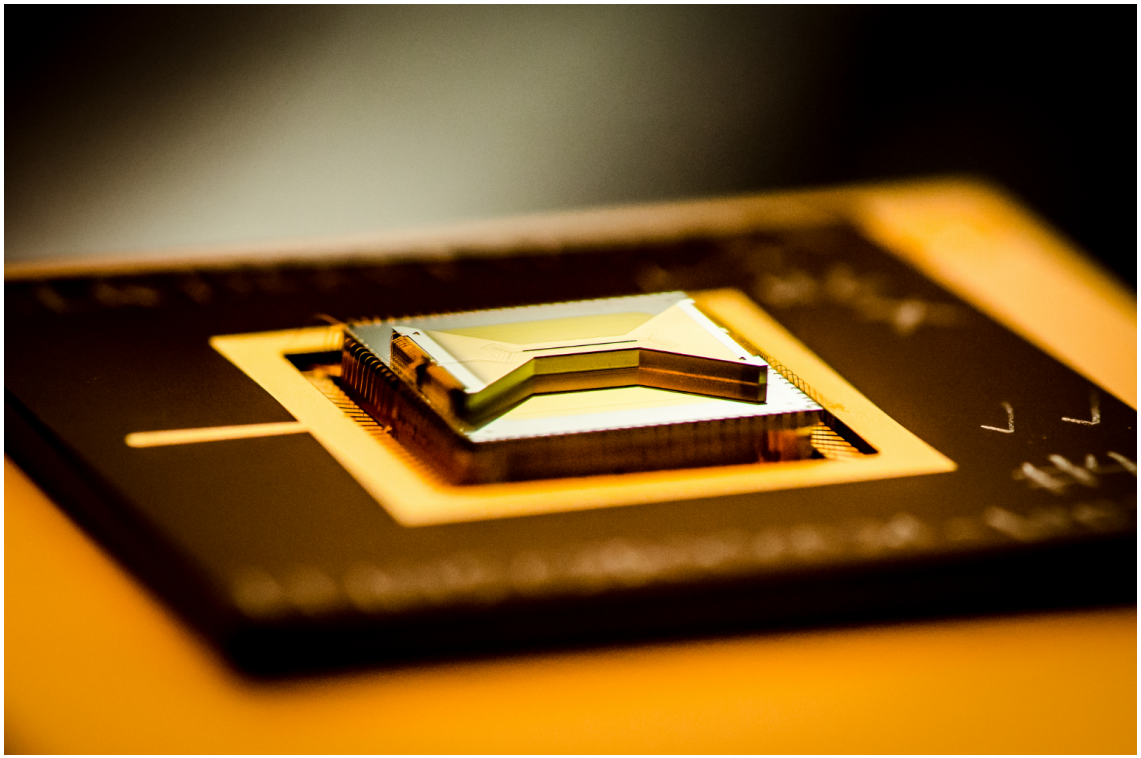


Figure 3.1: **Diagram of common ion trap types.** The unfilled circle denotes an ion. On a blade (rod) trap (Fig. 3.1a), the trapping RF potentials are provided by the blades (rods). Rods and blades can be used interchangeably. Blades are sometimes used instead of rods to provide better optical access, but the physics will be similar regardless of the physical shape of the trapping electrodes on a blade/rod trap. If multiple ions are trapped simultaneously in a blade trap, they will extend in a line above/below the pictured ion, perpendicular to the page. Surface Traps (Fig. 3.1b) are commonly microfabricated, and consist of many electrodes on the surface of the trap (not shown) that together create the trapping potential. A Surface Trap is effectively an "unwrapped" blade trap, where all electrodes are on the same plane.

(a) Photograph of an HOA 2 chip trap



(b) Diagram of chip trap, ion, and lasers

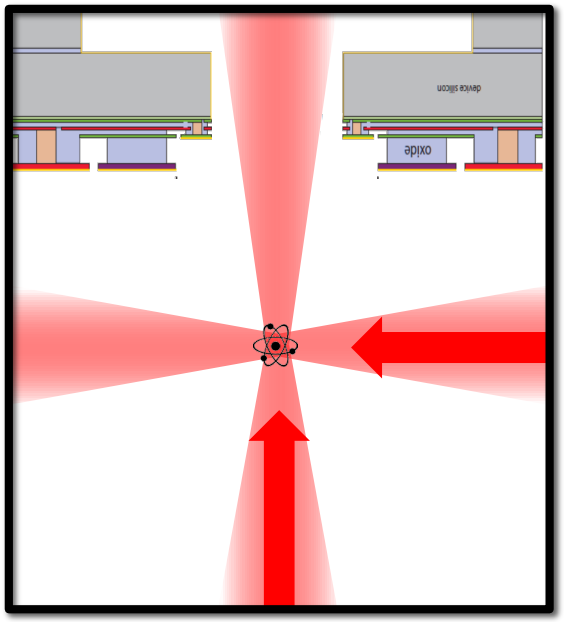


Figure 3.2: **Sandia HOA 2 surface electrode ion trap.** The HOA shown in Fig. 3.2a is an RF Paul trap. The line in the center of the trap is the "slot", which allows directing laser beams perpendicular to the surface of the trap through the slot. The trap is in a bowtie shape to allow for tighter optical focusing at the ion location; tapering the width of the trap towards the center allows for a wider range of access angles compared to a rectangular-shaped trap. The ions are moved from loading regions near the ends of the bowtie to the center of the slot, where they are manipulated using laser beams.
Fig. 3.2b shows the HOA, laser beams (red shading), and ions (atom symbol) that we use to perform qubit operations. In our system, the HOA is suspended upside-down (i.e. inverted relative to Fig. 3.2a). The chain of ions comes out of the page. The ions are trapped about $70\,\mu\mathrm{m}$ above the surface of the trap (i.e. the electrodes). The vertical laser beam is one of the "individual" addressing qubit laser beams, which is tightly focused on a single ion, while the horizontal laser beam is a broad "global" beam that addresses all ions at the same time.
Image Credits: UMD JQI/Monroe Group.

Our ion trap is a surface electrode trap, specifically the Sandia High Optical Access (HOA) 2.1.1 [97, 98, 99, 100, 101, 102, 103], henceforth shortened to the *Sandia HOA*. This *ion trap* is an electronically-passive device, which acts as somewhat of an antenna to emit the RF fields in a specific pattern, effectively creating a semi-stable RF null where the ion(s) will rest [85]. Surface electrode traps are microfabricated ion traps that are functionally similar to a Printed Circuit Board (PCB), but smaller and typically made out of silicon. [9] Just as a PCB might have exposed copper traces that can inadvertently emit EMI (Electromagnetic Interference), a similar process can emit the (intentional) electric fields that trap an ion.

Let us consider for a moment these EM fields. While it is appropriate to consider the EM field at the point of the ion, for simplicity in this context we will only consider the voltages at the ion trap that *produce* the EM fields that the ion experiences. As previously mentioned, there are two types of these EM fields: static (DC) and oscillating (RF). The static field can be thought of as controlling the *axial*[10] confinement, and the oscillating field can be thought of as providing the *radial*[11] confinement of the trap. We will label these DC and RF respectively, and use the terms electrode, electrical net, and pad interchangeably. The Sandia HOA has one exposed RF electrode, and over one hundred DC electrodes. Some of the DC electrodes share a source (the same input pin is connected to multiple electrodes), so there are actually only 94 DC control signals to the Sandia HOA [97]. The control of these Trap DAC signals will be discussed further in Section 4.5.

[9] The primary requirement is that the surface layer be made of an insulator with conductive pads/traces for producing EM fields.

[10] i.e. along the length of the trap. In Fig. 3.2b, this is normal (perpendicular) to the plane of the page.

[11] i.e. radial when looking directly into the "chain" of ions. In Fig. 3.2b, this is the plane of the page.

3.3.2 Out-of-Vacuum Components

Another important conceptual difference between the physical hardware for classical & quantum computers is that a quantum computer requires high isolation from its environment. For a highly-sensitive quantum computer, every interaction with its environment will decohere (degrade) its "quantumness" (more precisely, the quantum information stored in a qubit), so interactions between the quantum part and its surroundings (*environment*) should be minimized as much as possible. This is primarily realized through reducing the temperature & increasing physical isolation.

Most quantum computers will typically operate at extremely low temperatures & pressures. [12] For superconducting qubits, these goals are simultaneously achieved by using a cryostat (dilution refrigerator) which will cool the superconducting qubits & surrounding equipment to $\approx 20\,\mathrm{mK}$. In trapped ions, these two goals tend to be implemented separately: temperature reduction of an ion is executed by various methods of laser cooling [104], while physical isolation is primarily provided by encapsulating the ion trap in a vacuum chamber, which can be evacuated to reduce the background gas molecules in the system. This extremely low pressure allows ion traps, including EURIQA Breadboard, to operate at room temperature, meaning everything is at room temperature ($\approx 25\,^{\circ}\mathrm{C}$) *except* the ions. [13]

[12] Realistically, temperature & physical isolation are related. The primary way that heat transfer (whether heating or cooling) occurs is via collisions between two particles of different energy, which initiates an energy transfer between the two. With enough collisions and constantly cooling one of the two substances (particles), average temperature will decrease. As the temperature decreases, collisions will become less frequent, which increases physical isolation.

[13] Many trapped ion quantum computers are now using cryogenic coolants to reduce the overall temperature of their vacuum chamber system [67, 70, 105, 106]. While this may seem to implement temperature reduction, the primary benefit that this provides is lowering the background gas temperature, which reduces the rate of collisions between ion(s) & the background gas. This then lowers the frequency of ion chain loss

Temperature management of the ions via laser cooling will be discussed later in Section 3.6.2.2, but it is important to understand that there are many different laser beams that need to be projected at the ion chain in order to operate the trapped ion quantum computer (see Table 3.2 for a summary of these lasers). Each of these laser beams must:

- *Project to a small location*: to be focused on individual ions to allow for individually-addressed single- and multi-qubit gates. [14]
- *Be mechanically robust*: minimal susceptibility to drift or mechanical vibrations.
- *Sufficient power*: Higher laser power tends to mean that the gates can be executed faster, which minimizes overall circuit runtime to reduce the effects of decoherence.
- *Stable frequency*: some of the atomic transitions in $^{171}Yb^{+}$ have relatively narrow linewidths, such as the $435\,\mathrm{nm}$ transition (see Fig. 3.3). Even if the transition is not narrow, a drifting laser frequency can negatively impact reliability and uptime.

To produce these laser beams, there must be nearby space dedicated to producing & refining the laser beams to the desired optical characteristics (i.e. position, shape, intensity, etc). For our system, we chose to separately generate the laser beams, and then fiber-couple[15] them to beam positioning boxes. These boxes are directly mounted to the vacuum chamber[16], and can be minutely adjusted via motors.

(Section 3.6.4).

[14] The waist (width) of the laser beam at the ion should be much less than the spacing between the ions.

[15] i.e. direct the laser beam into an optical fiber, route an optical fiber near the destination, and then modify the output of the fiber to produce the final intensity/shape.

[16] i.e. mechanically referenced to the vacuum chamber

3.4 $^{171}Yb^{+}$ Atomic Physics

The energy levels and structure of an ion are important to understanding how operations on the atom are performed. In the ideal case, we want to consider a trapped ion as a system that can only be in one of two states (e.g. $\{|0\rangle, |1\rangle\}$). In reality, most modern qubits are encoded in systems which have more than two levels, and two levels are selected for the qubit. As such, we need to consider the atomic states (levels) of a given ion.

3.4.1 What are atomic levels?

Atomic levels are the different states that an atom/ion can occupy. These energy levels correspond to different configurations of the electron shell around a nuclear core. When a certain amount of energy is added to/subtracted from the system, in our context typically by lasers, the atom can transition between different energy levels.

The difference between two atomic levels is typically described by its transition wavelength (or equivalently the transition frequency, via $\nu = \frac{c}{\lambda}$). [17] The atom can naturally change (decay) from one higher-energy state to a lower-energy state. [18] When this happens, it will emit a photon of a certain frequency, which is at the transition wavelength of the difference between the state that it started in and the state that it decayed into. There are many different transition wavelengths in a given atom, which collectively generate a *spectrum* of wavelengths. This is called the *decay rate*. The *lifetime* of an atomic state is the average time [19] that an atom will remain in a given state before decaying into a

[17] In this equation, λ is the wavelength, c is the speed of light, and ν is the frequency.

[18] The decay rates for certain atomic states are extremely long (e.g. 5.4 years for $^{171}Yb^{+}$'s $^{2}F_{7/2}$ state), while others are extremely short (e.g. $\approx 8\,\mathrm{ns}$ for the $^{171}Yb^{+}$ $^{2}P_{1/2}$ state). See Fig. 3.3.

[19] Specifically, the time after which $\frac{1}{e} \approx 37\,\%$ of the original state remains.

different state, typically abbreviated as τ.

A few other key concepts that are important to understand about atomic physics and energy diagrams:

- *Selection rules*: These determine which states (levels) are coupled to each other, i.e. if it is possible to transition between any pair of energy levels $A \rightarrow B$.

- *Hyperfine states*: These are adjacent substates to a general electron orbital shell configuration that arise when an atom/ion has nuclear spin, and correspond to different spin configurations of the electrons relative to the nuclear spin of the atom-/ion. The spin states will align to some degree with the background magnetic field, which creates different sub-states of the same energy level, just offset by a (typically small) amount. These different states can be typically addressed by modifying the polarization of the driving laser beams, or by tuning the laser frequency to be the same as that particular state.

- *Ground State*: The ground state is a special state of the atom & its level diagram. This is the lowest energy state that the atom can be in. In other words, any operations on its electron shell will *add* energy to the atom. In $^{171}\mathrm{Yb}^{+}$, the ground state is $^{2}S_{1/2}$.

- *State Transitions*: A typical way of transitioning between different atomic states is by directing a laser beam at the ion. The laser beam's wavelength (frequency) must be at (near) the transition frequency between the two levels.

- *Linewidth*: This is a measure of the allowed frequency ranges that can drive a tran-

sition, which typically follow a Lorentzian profile. The linewidth of a state is inversely proportional to the state lifetime. Generally, transition linewidths that are wide ($\approx$ MHz) are easy to drive, while narrow $\approx$ Hz linewidths will be more difficult. The primary difficulty in driving transitions with narrow linewidths comes from stabilizing the laser frequency to the very narrow range of the linewidth.

3.4.2 $^{171}Yb^{+}$ Atomic Levels

$^{171}Yb^{+}$ is a Hydrogen-like ion with a hyperfine clock state transition, as shown in Fig. 3.3. For our setup, we have chosen to define our qubit levels $|0\rangle$ & $|1\rangle$ as sub-levels separated by a hyperfine splitting of $\approx 12.6\,\mathrm{GHz}$ (see Section 3.7.1.4 for the exact number). [20]

[20] Hyperfine splittings arise due to interactions between an atom's electrons and its nucleus.

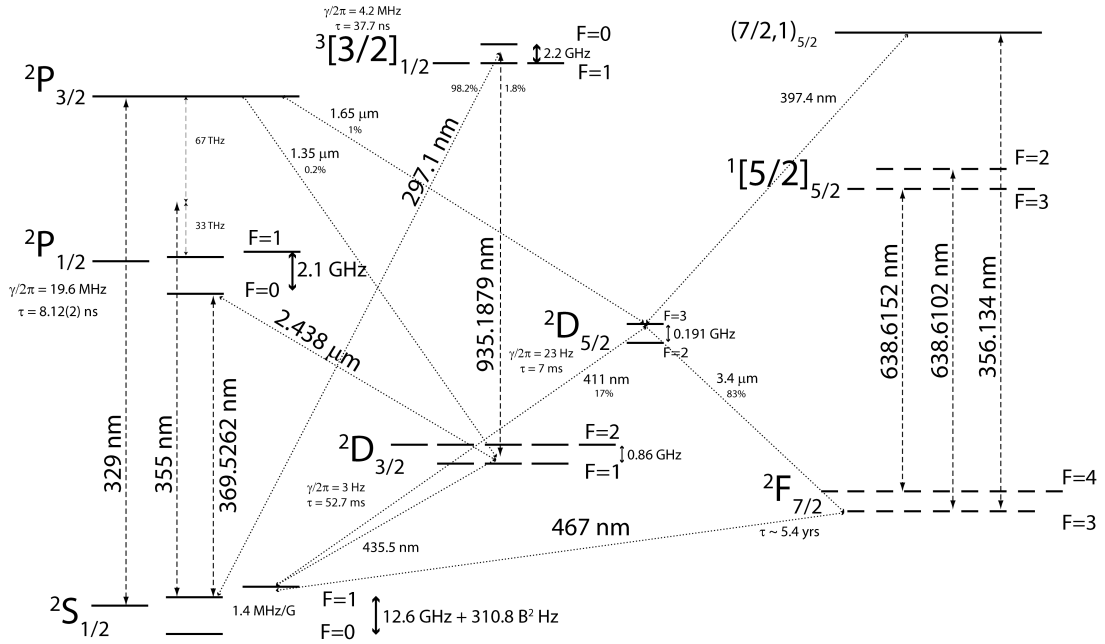


Figure 3.3: **Diagram of the atomic energy levels of $^{171}Yb^{+}$ ions.** The horizontal lines in this diagram denote different energy levels of $^{171}Yb^{+}$. Within a given shell configuration (e.g. $^2S_{1/2}$), there are different sub-levels that correspond to different electron- & nucleus-spin configurations. We chose to define our qubit states as sub-levels of the $^2S_{1/2}$ manifold: $|0\rangle \equiv |F=0, m_F=0\rangle$, $|1\rangle \equiv |F=1, m_F=0\rangle$. To rotate between $|0\rangle \leftrightarrow |1\rangle$, we use a pulsed $355\,\mathrm{nm}$ laser to off-resonantly couple to the $^2P_{3/2}$ & $^2P_{3/2}$ levels, which can yield a transition between $|0\rangle \leftrightarrow |1\rangle$ if the difference between two paths of this laser are equal to the frequency difference between the $|F=0, m_F=0\rangle$ & $|F=1, m_F=0\rangle$ states. This is called a stimulated two-photon Raman transition. Note that this frequency difference can change depending on e.g. the magnetic field. The dashed lines denote transition wavelengths between two energy levels. Dotted lines indicate spontaneous decay transitions from a higher-level energy state to a lower-level energy state. The $\frac{\gamma}{2\pi}$ denotes the transition wavelength of this state, and τ denotes the lifetime of that state. For quantum computation, the primary states in this diagram that concern us are $^2S_{1/2}$, & $^2P_{1/2}$. For state preparation, we also need to consider $^2D_{3/2}$, as discussed in Section 3.6.2.2. Finally, note that this is a partial state diagram, and not every possible state of $^{171}Yb^{+}$ is shown here. Image courtesy of George Toh.

3.5 Operations

A quantum computer must be able to perform operations on qubits in order to be useful. Generally, operations fall into several classes:

- *Ion (Non-Qubit) Operations* (Section 3.6): These operations can be thought of as "bookend" operations, in that they happen before or after the main sequence to prepare or measure the ions in some way, but are generally agnostic of what state that the qubit is in.

- *Qubit Operations* (Section 3.7): These operations are roughly unitary, implement the general concepts from Section 2.4, and comprise both single-qubit, two-qubit, & multi-qubit operations.

3.6 Ion (Non-Qubit) Operations

In this classification, ion operations are purely atomic physics-oriented, and have no direct relation to any coherent quantum information for ions (other than the act of measurement). That should not be mistaken as having no relation to quantum computation. Ion operations are extremely important for preparing quantum states. However, these operations are logically distinct from quantum operations, in that they are fundamental to quantum operations, and are required to prepare an ion to be used as a qubit, or to move classical information into or out of an ion (i.e. state preparation & measurement).

Following sections will discuss how each of these operations are implemented on our ion trap quantum computer. The majority of these operations are summarized in

Purpose	State Sequence	Transition Wavelength (Approx., nm)
Ionizing neutral Yb	$^{171}\text{Yb} \rightarrow {}^{171}\text{Yb}^{+}$	399
Doppler Cooling & State Readout	$^2S_{1/2} \rightarrow {}^2P_{1/2} \rightsquigarrow {}^2S_{1/2}$	369.5
"Repump" to computational subspace	$^2P_{1/2} \rightsquigarrow {}^2D_{3/2} \rightarrow {}^3[3/2]_{1/2} \rightsquigarrow {}^2S_{1/2}$	935
Pump into $\|0\rangle$	$^2S_{1/2} \rightarrow {}^2P_{1/2} \rightsquigarrow \|0\rangle$ or $\rightsquigarrow$ ($\|F=1\rangle$ manifold $\rightarrow {}^2P_{1/2} \rightsquigarrow \|0\rangle$)	369.5
Raman operations	$^2S_{1/2} \rightharpoonup$ (between $^2P_{1/2}$ & $^2P_{3/2}$)	355

Table 3.2: **Main transition wavelengths in an $^{171}\text{Yb}^{+}$ qubit.** This is a non-exhaustive list of the main wavelengths that are used when performing quantum operations on $^{171}\text{Yb}^{+}$. In the above table, $\rightarrow$ denotes a stimulated transition driven by a laser, $\rightsquigarrow$ denotes a spontaneous decay transition (i.e. not laser-driven), and $\rightharpoonup$ denotes an off-resonant transition (i.e. a transition that is not directly driven). These operations are all described in Section 3.6.

Table 3.2, which summarizes the different lasers & atomic levels needed to perform a given operation.

3.6.1 Ion Loading Operations

The first step in operating on a trapped ion qubit is to procure the trapped ion itself. This process has two steps: moving the atom to the proper location, and removing electron(s) from the atom to produce a positively-charged atom. These two steps can occur in any order, as long as at the end of the process an ion is injected into the "trapping region" where the RF fields of the ion trap are suitably configured.

To consider how atoms are moved from their storage location to the trapping region, it is important to recall the system constraints. Ion trapping typically occurs in a sealed vacuum chamber (Section 3.3.2). Thus, ions cannot be directly injected into the system,

and must be produced from components which are already under vacuum (i.e. in the vacuum chamber). The community has settled on two methods for this process:

- *Thermal Oven*: This method involves heating a chunk of target material (in our case, primarily neutral ^{171}Yb) at high temperature to sublimate it[21], which causes atoms to emit in arbitrary directions. The spray can be directed using a tube, which focuses the spray towards the target trapping zone. A small percentage of these ions will be moving at the correct velocity (energy) to be ionized & trapped by the oscillating RF fields (Section 3.3.1). These ovens typically have a warm-up time, which is necessary to produce a sufficient beam of atoms (flux) to ionize atoms. [22]

- *Laser Ablation*: This method involves firing a high-energy laser at a small chunk of target material (^{171}Yb), which will spray in all directions and some of which will be small & slow enough to be trapped. This method typically does not use an ionizing laser, because the high energy of the ablation will typically produce some ions.

Atoms are generally uncharged (neutral) when in storage, and must be charged to be used as an ion, typically by removing an electron. [23] For our system, we chose to use an ionizing $399\,\mathrm{nm}$ laser to excite an ^{171}Yb neutral atom from $^1S_0 \rightarrow {}^1P_1$, and then a $393\,\mathrm{nm}$ laser to remove an electron [93, Sec. 2.2].

[21] We heat this using an electric current, in a process similar to that of turning on a tungsten electric lightbulb.

[22] Our oven is actually positioned on the back (non-electrode) side of the HOA. Atomized Yb will be directed towards the rear of the loading zone via the tube of the oven, and then will be reduced to a smaller fraction by use of a slit in the HOA. Because only a small fraction of all atoms ejected from the oven will be ionized, it is preferable for the many spare atoms to be in a position where they will have minimal effect on normal qubit operations.

[23] It is possible to doubly-ionize an atom, producing e.g. $^{171}\mathrm{Yb}^{++}$. This is uncommon, but has been observed.

3.6.2 State Preparation Operations

Once an atom has been ionized, and then the resulting ion is trapped, a few steps still remain to be able to perform coherent quantum operations on the ion. These steps are cooling, and then state preparation (preparing the qubit in $|0\rangle$).

3.6.2.1 Ion Ground State Cooling

To cool an ion to the ground state, the first step is typically Doppler cooling [104]. Doppler cooling is a process by which an ion absorbs a photon from a laser beam, and then emits a photon in a random direction. The key to Doppler cooling is that the laser is tuned to the red (lower frequency) of the transition resonance frequency, which causes the ion to preferentially absorb the photon when the ion is blue-shifted due to its velocity relative to the laser beam. Thus, the ion will mostly absorb energy when moving towards the origin of the laser photons, providing a net cooling/slowing force on the ion [93, 104, 107].

The purpose of Doppler cooling in our sequence is to reduce the motional energy, or temperature, of an ion. Motional energy of an ion is measured in units of motional quanta, called *phonons* and which are represented as $|n\rangle$. The motional energy is tallied for each *mode* and *axis* of the harmonic oscillator of the ion separately. For a chain of N ions in 3-dimensional space, there are $3N$ phonon modes in the chain. [24]

It is important to note that measuring the number of phonons (a quantum property) is a statistical process, because the value must be calculated from an average of several

[24] This is important for the discussion in Section 3.7.1.1.

rounds of quantum measurement. Even with $\bar{n} \leq 1.0\,\text{quanta}$, for a single experiment it is still possible to have relatively high n, e.g. $n = 3\,\text{quanta}$. Finally, remember that $\bar{n}$ is implicitly a timestamped value, which is measured at a certain point in the experimental sequence; measuring earlier or later could yield different $\bar{n}$. Generally, an ion that is not continuously cooled will heat some amount [108], which is measured in quanta per second. Thus, if a system has $\bar{n} = 0.1\,\text{quanta}$ immediately after cooling, and a $\dot{\bar{n}} = 10\,\text{quanta/s}$, after $100\,\text{ms}$ of no further cooling the $\bar{n}$ will be $\bar{n} = 1.1\,\text{quanta}$, assuming linear heating rates. [25]

Doppler cooling is highly effective at cooling from high-energy motional states. However, Doppler cooling has a lower limit to which it can cool. Since many multi-qubit gate schemes for trapped ions rely on motional state coherence (e.g. the Mølmer-Sørensen gate (MS) [1], or an N-qubit gate as described in Section 7.2), to obtain the highest fidelity it is desirable to have the lowest achievable motional quanta. The Doppler limit for $^{171}\text{Yb}^{+}$ on the $369\,\text{nm}$ cycling transition is $\approx 5\,\text{quanta}$ [109], significantly higher than desired for high-quality gates (targeting $\ll 1\,\text{quanta}$).

To calculate the Doppler cooling limit for $^{171}\text{Yb}^{+}$, we use the equation for the Doppler laser cooling temperature for an atomic transition in a harmonic ion trap, defined as [110]:

$$E_i^{\infty} = \frac{\hbar\gamma}{4}\left(1 + \frac{1}{3\cos^2(\theta)}\right)\left[\frac{\Delta}{\gamma} + \frac{\gamma(1+s)}{4\Delta}\right] \tag{3.1}$$

where E_i^{∞} is the energy along the ith axis after infinite cooling duration, $\hbar$ is Planck's constant, γ is the radiative linewidth of the atomic transition that is being cooled, θ is

[25] The heating rate $\dot{\bar{n}}$ is the change in phonon population $\bar{n}$ over time (the derivative of $\bar{n}$). Refer to [108] for a thorough treatment.

the angle of a single laser beam relative to the ith axis, Δ is the red detuning of the laser beam frequency ω from the transition frequency[26] ω_L ($\Delta = \omega - \omega_L$), and s is a saturation parameter denoting the laser beam intensity I relative to the saturation intensity I_s, where $s = I/I_s$. This expression is minimized with parameters $s \ll 1$, $\theta = 0$, & $\Delta = \gamma/2$, resulting in the minimum Doppler temperature of

$$E^{\infty}_{i,min} = \frac{\hbar\gamma}{3} \tag{3.2}$$

Given that $\gamma = 19.7\,\mathrm{MHz}$ for the $370\,\mathrm{nm}$ ${}^2S_{1/2}$ to ${}^2P_{1/2}$ ${}^{171}\mathrm{Yb}^+$ transition, and using the Boltzmann's constant k_B, the Doppler cooling limit in our experiment is:

$$E^{\infty}_{i,min} = \frac{\hbar 19.7\,\mathrm{MHz}}{3} = 6.925 \times 10^{-28}\,\mathrm{J} \tag{3.3}$$

$$E^{\infty}_{i,min} = k_B * T \tag{3.4}$$

$$T \approx 50\,\mu\mathrm{K} \tag{3.5}$$

This can be equated to the number of phonons in a vibrational mode ω_m by using [67, Eq 2.2]:

$$E^{\infty}_{i,min} = \hbar\omega_m \left(\bar{n} + \frac{1}{2}\right) \tag{3.6}$$

Assuming that the lowest-frequency axial mode of an ion chain is $\approx 200\,\mathrm{kHz}$[27], then the

[26] Atomic transitions are typically denoted by their wavelength, but can be equivalently described as a frequency using $\nu = \frac{c}{\lambda}$

[27] This value of $200\,\mathrm{kHz}$ is representative for our work with 15-ion chains. It is primarily dependent on the confining DC voltage potential; the axial COM frequency of a single ion will typically be higher.

minimum number of phonons after Doppler cooling will be:

$$\frac{19.7\,\mathrm{MHz}}{3} = 200\,\mathrm{kHz}\left(\bar{n} + \frac{1}{2}\right) \tag{3.7}$$

$$\bar{n}_{min} \approx 32.33\,\mathrm{quanta} \tag{3.8}$$

Thus our cooling has a second stage to go beyond the Doppler cooling limit. There are several schemes that would theoretically work, but we have chosen to use sideband cooling [111], specifically parallel sideband cooling [109]. With SBC, we are able to achieve cooling to $\bar{n} < 0.5\,\mathrm{quanta}$ for a 15-ion chain ([93, Fig 5.2]).

3.6.2.2 Ion State Preparation

Once the ion(s) are sufficiently cooled near their ground motional state(s), the next step in an experiment cycle is to prepare them into the qubit basis, typically $|0\rangle$, via a process called optical pumping [112], which is illustrated in Fig. 3.4. The goal of this step is to reliably move an ion into the $|0\rangle$ atomic level by forcing the ion to progress through a sequence of atomic level transitions that "trap" it in the desired state.

Effectively, this process works by:

1. Turning on the $370\,\mathrm{nm}$ laser (this includes properly aiming the laser beam at the ion(s) to prepare).

2. Adding sidebands to the laser using AOMs to only excite the ${}^2S_{1/2}\,|F=1\rangle$ manifold states ($|m_F = \{-1, 0, 1\}\rangle$) to the ${}^2P_{1/2}\,|F=1, m_F = \{-1, 0, 1\}\rangle$ states (see [113, Fig. 2.2] for reference).

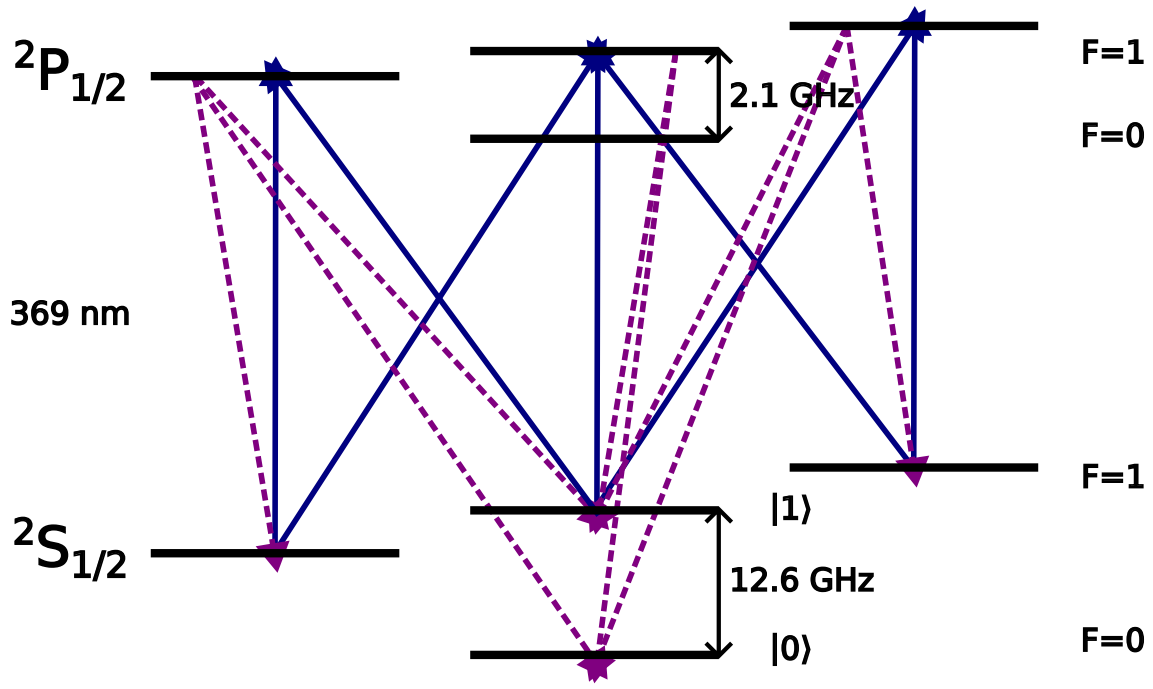


Figure 3.4: **$^{171}Yb^+$ Optical Pumping State Diagram.** This diagram illustrates the transitions that are needed to pump into the state $|0\rangle$ of $^{171}\mathrm{Yb}^+$ ($^2S_{1/2}$, $|F=0, m_F=0\rangle$). Blue solid arrows represent driven transitions ($\approx 370\,\mathrm{nm}$) via photon absorption. Dashed purple arrows represent spontaneous photon emission. $369\,\mathrm{nm}$ and $370\,\mathrm{nm}$ are used interchangeably because the transition is near $369.5\,\mathrm{nm}$.

3. Allow the level to decay into $|F = 0, m_F = 0\rangle$. The lasers are not tuned to excite the ion out of this atomic state.

4. If the ion decayed into a state other than $|F = 0, m_F = 0\rangle$, i.e. the ${}^2S_{1/2}\,|F = 1, m_F = \{-1, 0, 1\}\rangle$ states, those states are then excited to the ${}^2P_{1/2}\,|F = 1\rangle$ manifold, where it has another chance to decay to the $|F = 0, m_F = 0\rangle$ state.

5. This process continues long enough that enough transitions have happened that it is vanishingly unlikely for the ${}^{171}\mathrm{Yb}^+$ ion to be in any state other than $|F = 0, m_F = 0\rangle$ ($|0\rangle$).

3.6.3 State Measurement Operations

After an experiment has completed, the state of the ion must be determined. Because this information is stored in a physical qubit, this is effectively an "analog" operation, meaning that this is a sampling operation that has some implicit error in approximating the actual state. [28] Because this is an "analog"-style operation, it is useful to think of this operation in terms of signal & noise. Many classical analog measurements are most effective[29] if they maximize the Signal-to-Noise Ratio (SNR) for a given setup. A similar concept applies to measuring ion state. There are certain characteristics that indicate the state of the ion, and care was taken to maximize the difference between $|0\rangle$ & $|1\rangle$, maximizing the probability of a correct state measurement.

To measure the state of an ${}^{171}\mathrm{Yb}^+$ ion, the ion must simply be illuminated with

[28] As an example, assume that the ion is definitively prepared in $|0\rangle$. If your detection fidelity is $99\,\%$, then $99\,\%$ of the time this ion will register as $|0\rangle$, and $1\,\%$ of the time this ion will register as $|1\rangle$.

[29] In terms of integration time, accuracy, precision, etc.

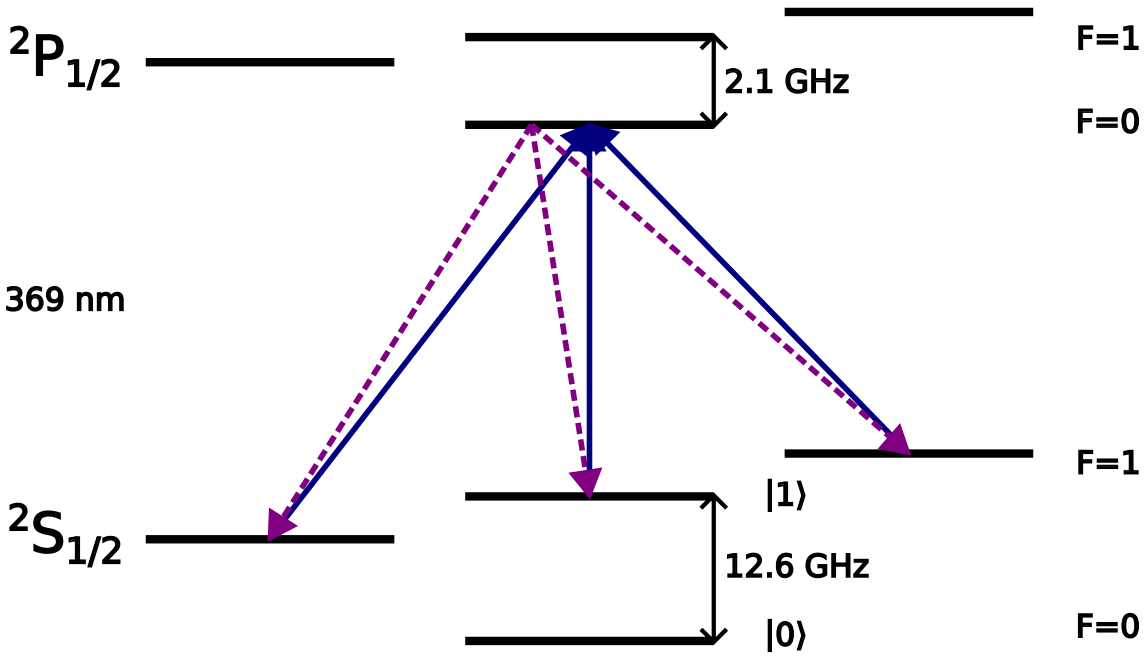


Figure 3.5: **$^{171}Yb^+$ Readout State Diagram.** Only qubits that are in $|1\rangle$ will be excited to emit light. Blue solid arrows represent driven transitions ($\approx 370\,\mathrm{nm}$) via photon absorption. Dashed purple arrows represent spontaneous photon emission. $935\,\mathrm{nm}$ light is not shown here, though it is important to the readout process. The role $935\,\mathrm{nm}$ light is to "re-pump" from the $^2D_{3/2}$ state (which can be entered spontaneously when decaying from $^2P_{1/2}$) to the $^2S_{1/2}$ $|F=1\rangle$ manifold. This ensures that the cycling transition will continue to emit light if the qubit is in $|1\rangle$, instead of stopping whenever the ion enters the $^2D_{3/2}$ state.

$369\,\mathrm{nm}$ and $935\,\mathrm{nm}$ light of the correct wavelengths, as shown in Fig. 3.5, and then scattered $369\,\mathrm{nm}$ fluorescence from the ion should be collected. This process is similar to the cooling cycle described in Section 3.6.2.1. We call the duration of this entire period the "detection window", and will assume that the illumination drive and signal acquisition window will last for the entire "detection window". To be more specific, the following sequence happens from the perspective of the ion [114, 115]:

1. *Illumination* with $369\,\mathrm{nm}$ light of the appropriate frequency to perform excitation (next step).

2. *Excitation* from ${}^2S_{1/2}$ to the ${}^2P_{1/2}$, $|F=0\rangle$ manifold. The frequency is chosen so that only ions which are in $|1\rangle$ (specifically, the $F=1$ manifold) will be excited, but not ions that are in $|0\rangle$ ($|F=0, m_F=0\rangle$). Thus, we call ions in $|0\rangle$ "dark", and ions in $|1\rangle$ "bright", because only ions in $|1\rangle$ can undergo this transition that results in emitting photons. This is a convention choice for our particular ion; other ions such as ${}^{40}\mathrm{Ca}^+$ tend to use $|0\rangle$ as "bright". [30]

3. *Spontaneous emission* of a $369\,\mathrm{nm}$ photon. Alternatively, decay ($\approx 0.5\%$) to the ${}^2D_{3/2}$ state, which is repumped to ${}^2S_{1/2}$ $|F=1, m_F=0\rangle$ via ${}^3[3/2]_{1/2}$ using $935\,\mathrm{nm}$ light.

4. *Repeat* 1-3 until illumination ends.

[30] It is important to note that the ONLY states that will fluoresce in this scheme are ${}^2S_{1/2}$ $|F=1\rangle$ manifold (at readout time during an experimental sequence, practically the entire population will be in $|F=1, m_F=0\rangle$ and not the $m_F=\{-1,+1\}$ states). The converse is that atomic states that are **not** $|1\rangle$ (more precisely, $|F=1, m_F=\{-1,0,1\}\rangle$) will be indistinguishable. So this scheme cannot tell the difference between an ion in $|0\rangle$ and one that is in a stray atomic level that is not in the computational subspace. For instance, a collision can force an ion to the ${}^2F_{7/2}$ state, which will also appear dark and thus indistinguishable from $|0\rangle$.

Once the photon is emitted from the ion, it will scatter in a random direction in 3D space. This photon will be collected with some probability via a collection of optics, and then routed to a detector. In our system, the primary loss in this chain is due to the first lens, which although being relatively large[31] ($NA = 0.63$) still only collects $\approx 7.4\,\%$ of the photons spontaneously emitted by the ion ([93, Eq. 4.2]).

Specifically, in order to be detected, this entire sequence of events must occur:

1. $369\,\mathrm{nm}$ *Photon Emission* from the ion.

2. *Photon Collection* by high-NA lens.

3. *Photon Transmission* by fiber (32-channel fiber, roughly one per ion that the hardware supports).

4. *Photon Amplification* by Photo-Multiplier Tube (PMT). This generates an electrical signal when a photon is collected.

5. *Signal Acquisition*: the electrical signal is conveyed over electrical transmission lines and registered by a digital input device (described in Chapter 4).

6. *Event Counting*: each input event during the readout window is recorded in a buffer, and then counted by the digital control device. [32]

[31] In terms of overall angle covered.

[32] We have chosen to count the number of input photons, discarding the event timestamps. If pursuing high-fidelity state readout, this information could be incorporated for slightly higher readout fidelity [116, 117]. The general idea behind this scheme is that the $^{171}\mathrm{Yb}^{+}$ readout scheme has a time-dependence on when photon counts occur, and the time of photon arrival can indicate state transitions in the ion. For example, over a detection window the ion could initially be "dark", and then later be "bright". This is called *dark-to-bright pumping*. *Bright-to-dark pumping* is also a potential error. These errors are both more likely to occur at later times in the detection sequence, so signals earlier in the detection window should be given more weight.

Operation Type	Example Operation
Whole-Chain	Ion Chain Movement, Motional Mode Adjustments
Sub-Chain	Chain sorting, Split/Merge Chain
Individual Ion	Ion Loading (Merge to Chain)

Table 3.3: **Example of Ion Chain Operations.** These are just a subset of all possible operations, and will be summarily covered in Section 3.6.4. Sub-chain operations will be discussed in Chapter 8.

Once the approximate number of photon events has been counted, discrimination must still be applied to determine if the measured qubit is in $|0\rangle$ or $|1\rangle$. In our system, we have decided to use simple thresholding to determine which state that our qubit is in. We have previously collected data (e.g. [93, Fig. 2.9]) indicating the number of photons collected when the ion is in $|1\rangle$ vs in $|0\rangle$, and thus calculated the State Preparation & Measurement errors (SPAM) ([93, Table 5.1]). Based on these histograms, we have decided that the optimal threshold is one (1) count. In other words, if $\geq 1\,\mathrm{count}$ is recorded, we record $|1\rangle$, otherwise we record $|0\rangle$. This is admittedly a simple method for state discrimination, but even with this crude method our average single-qubit SPAM error is still $\approx 0.46(2)\,\%$. Other methods for state discrimination exist for ions [116, 118, 117], but we have not yet seen the need to push our measurement fidelities to this level.

3.6.4 Ion Chain Operations

Finally, there are certain operations that are performed on a chain of ions as a whole. These generally fall into the categories of whole-chain operations, sub-chain operations, and individual ion operations.

The whole-chain operations are gross changes that tend to affect the state of all ions

together. This is the regime where all ions are together in a single potential well, and are all affected equally. Moving the ion chain is simply moving the entire set of ions together. Adjusting the motional modes is described thoroughly in [119, Chapter 3].

At the other extreme is the individual-ion operation. In normal operation, the smallest unit that we care about is either a sub-chain or the entire chain. However, to get to the point where an entire chain of ions can be used, we must first create a chain of ions.[33] Ion chains are typically built one ion at a time by loading an ion (described in Section 3.6.1), and then merging it into the potential well where the remainder of the chain is stored. For loading the initial ion, the initial condition is simply that the well that would contain the chain does not hold a single ion. This process is then repeated until the desired number of ions are loaded. In our case, we do not have ion detection in the loading region (to see if the ion has been successfully loaded during one pulse of the ionization lasers), so we repeatedly attempt the sequence of:

1. *Attempt to load ion*: Described in Section 3.6.1.

2. *Shuttle ion to chain*. At this stage we don't actually know if there is an ion, but we still move the well that *would* contain the ion near the chain, which is approximately in the center of the trap (maybe offset by a few hundred micrometers).

3. *Check (Inspect) newly-loaded ion presence*. This step confirms that an ion is present after loading but before the merge into the chain. This allows issues with loading to be isolated from the more voltage potential-sensitive issues of chain merging.

[33] An ion chain is sometimes called an ion crystal because it is a repeating sequence which are treated as a single unit, with lattice sites operated by individual ions.

4. *Merge ion to the chain*. This is effectively the same as the previous step, but this step is somewhat delicate so it is important to consider it separately.

5. *Check if chain size increased*. Increment a counter accordingly. [34]

6. *Repeat* steps 1-5 until desired chain size (length) is reached.

Once these steps are completed, then the number of ions in the chain is checked using a *center-scan*. A center scan consists of shifting the potential well of the ions along the length of the trap (perpendicular to the laser beam), while performing a short Rabi pulse (described in Section 3.7.2). A Rabi pulse is one of the simplest quantum pulses, consisting of a coherent transfer ($|0\rangle \leftrightarrow |1\rangle$), in this case immediately followed by a measurement of the qubit state. If all of the ions are present and equally spaced, then we should see a Gaussian curve on each PMT channel around the same point.

Once an ion chain is loaded, it is important to remember that it is not a static object. Due to events such as collisions with the background gas, the ions can spontaneously reorder. When all of the ions are of the same species (i.e. the same isotope & charge), then

[34] In our setup, we detect the number of ions using the fluorescence collected on each PMT for a chain of ions during cooling. When an ion is added, the fluorescence will exhibit roughly a stair-step behavior, illustrated in Fig. 3.6 (see also [93, Fig. 5.1b] for an image of all PMT channels when loading). The "stair-step" is due to the ions aligning with and then anti-aligning with the PMTs that collect their fluorescence. In our system, our fibers are imaged onto a chain spacing of $\approx 4.45\,\mu\mathrm{m}$. However, the voltage potentials need to be properly adjusted to align the ions to the PMT imaging, but a chain is needed in order to perform this calibration. To bypass this race condition, we create a good-enough potential well based on the rough number of ions (e.g. one for the first 8 ions, one for the next 8 ions, etc.), and assume that enough total fluorescence will be collected. Then, when an ion is added to the well, the ions will shift roughly from aligned with the PMT to anti-aligned, which will cause the total fluorescence to drop. Thus we can use a simple dot-product between the previous fluorescence (number of PMT counts) and the current fluorescence. If the dot-product is greater than a threshold, we assume that an ion has been added (incrementing a counter), otherwise we assume that the load attempt failed and another attempt should be made. This counting method is a rough approximation, and it would obviously be more optimal to create and calibrate a different potential well for every number of ions up to the desired number of ions. However, we decided that the complexity of that approach was not necessary since our simpler approximation is reliable enough.

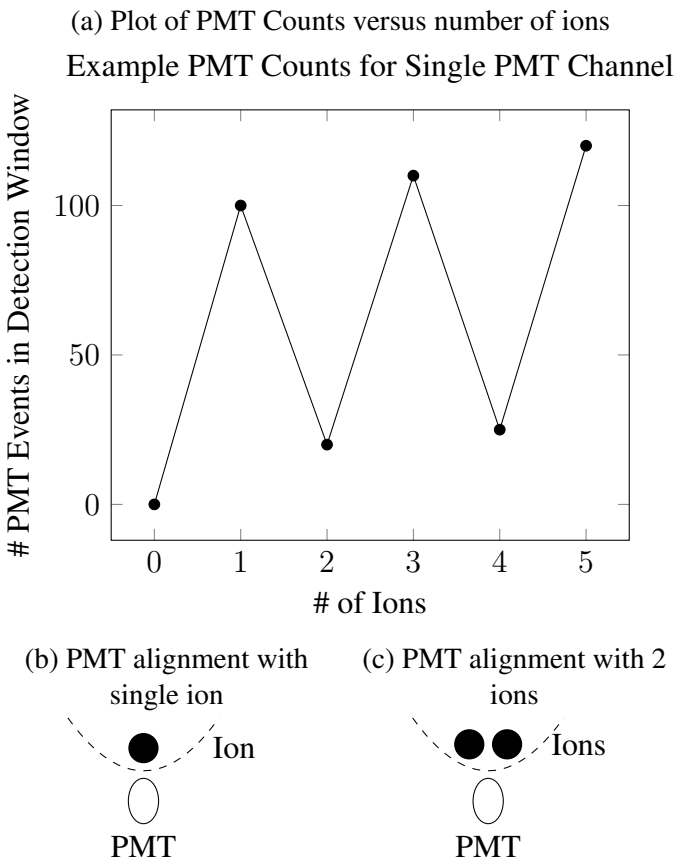


Figure 3.6: **Example of light collection aligning then anti-aligning with ion(s).** As the number of ions in the well changes from odd to even, the number of photons collected will dramatically drop because the ion chain is no longer aligned with the collection PMT. To visualize this effect, imagine a single PMT imaging the bottom of a parabola (simulating the potential well). With one ion, the bottom of the parabola is aligned with the PMT. With two ions, the ions will repel each other, and now the center of the well will be halfway between the two ions, causing the fluorescence collected by the PMT to drop dramatically. This effect could be mitigated by shifting the center of the well to align to the PMTs as the number of ions changes. However, we take advantage of this effect during loading to produce a strong signal indicating that an ion has been added to the well, indicating success in a loading attempt.

this is not an issue because all ions of the same isotope are identical. [35] By contrast, chains of mixed ion species, such as those considered in [120], or currently in use elsewhere [105, 121], will reorder into different sequences following a collision. In other words, following a non-destructive collision event the ion chain has a tendency to reorder into a different sequence of ions (typically not changing the total number of ions). The different options for handling this situation will be presented in Section 8.3, but assume for now that the optimal solution is reordering the scrambled chain into the desired configuration. Reordering an ion chain into the desired configuration is similar to a sorting algorithm from Computer Science, such as a simple inefficient Bubble Sort algorithm [122] or the more-efficient Merge-Sort algorithm [123]. Broadly, these algorithms are sequences of comparison ($a < b$) & swap ($(a, b) \rightarrow (b, a)$) operations between the elements. Thus, an essential operation on the ion chain to enable performing a chain sorting is the ability to do an ion swap on an arbitrary set of ions in the chain.

In other words, to physically perform a chain sorting (reordering) operation, the following physical actions must be performed for each step of the sorting algorithm:

1. *Determine ions to swap*

2. *Isolate ions to swap*

[35] This argument only holds when NOT operating on the ions as qubits (i.e. when performing gates on the qubits). During qubit operations, unintended reordering of the ion chain will cause gates to be applied to the incorrect ion(s)/qubit(s). In our scheme, we can check for ions using Doppler cooling on the $369\,\mathrm{nm}$ transition. However, this is the same transition that we use to measure the qubits, so performing this operation in the middle of qubit operations would collapse the state, defeating the purpose of any remaining quantum operations. Thus, we have chosen to implement reordering detection *between* "shots" of a quantum circuit. If reordering is detected, the previous shot could be discarded. We have determined that this an acceptably small error for the experiment repetition rate that we currently work with ($\approx 60\,\mathrm{Hz}$). We observe reordering chain collisions approximately every $7.5\,\mathrm{min}$. Thus, we approximate that $\frac{7.5\,\mathrm{min}}{reordering\ event} * \frac{60\,\mathrm{s}}{1\,\mathrm{min}} * \frac{60\ shots}{1\,\mathrm{s}} = 27000 \frac{shots}{reordering\ event} \rightarrow \approx 3.7 \times 10^{-3}\,\%$ error.

3. *Swap ions*

4. *Reconstruct chain*

5. *Check swap success*

This sort operation will be addressed in further detail in Section 8.3.

Once all of these cooling and state preparation operations are performed, and the desired chain configuration is obtained, then the set of ions can be considered a set of qubits.

3.7 Qubit Operations

After being initialized into $|0\rangle$, we can consider an $^{171}Yb^{+}$ ion as a qubit. The next step is to be able to apply single- and multi-qubit operations, which were broadly described in Section 2.4. Here we will overview how these operations are implemented on a trapped ion quantum computer.

3.7.1 Raman Operations

Before considering the specifics of gates, it is important to consider that there are two coupled quantum systems in a trapped ion quantum computer. One of these is a qubit mapped to the spin of the trapped ion atomic levels. The other is a motional harmonic oscillator, derived from the collective motion of the ion(s) in the harmonic potential of the ion trap.

Together, the Hamiltonian of a single trapped ion in a harmonic potential is [124,

Eq. 1][93, Eq. 3.5 & 3.15]:[36]

$$H = \hbar\Omega\sigma_+ \exp\left(-i\left(\mu t - \phi\right)\right) * \exp\left(i\eta\left[a\exp\left(-i\omega_t t\right) + a^\dagger \exp\left(i\omega_t t\right)\right]\right) + h.c. \quad (3.9)$$

where

- Ω is the strength of the laser field, observed as the Rabi frequency of the qubit.
- $\sigma_\pm$ is the atomic raising/lower operator.
- $a^\dagger$ & a are the creation and annihilation operators for a motional quantum (phonon), respectively.
- ω_t is the trap frequency of the ion trap.
- $\eta = k_x x_0$ is known as the Lamb-Dicke parameter, where k_x is the projection of the laser's EM wavevector along the trap axis, and $x_0 = \sqrt{\frac{\hbar}{2M\omega_t}}$ is the spread of the ion's wavefunction in the oscillator, and M is the mass of the ion. The Lamb-Dicke parameter is effectively the coupling strength between an ion's atomic states and its motional state.
- μ is the detuning of the laser beam from the atomic transition wavelength.
- ϕ is the phase of the field relative to the atomic polarization.
- $h.c.$ is an abbreviation for the Hermitian Conjugate of the entire sequence.

[36] After applying the rotating wave approximation (RWA)

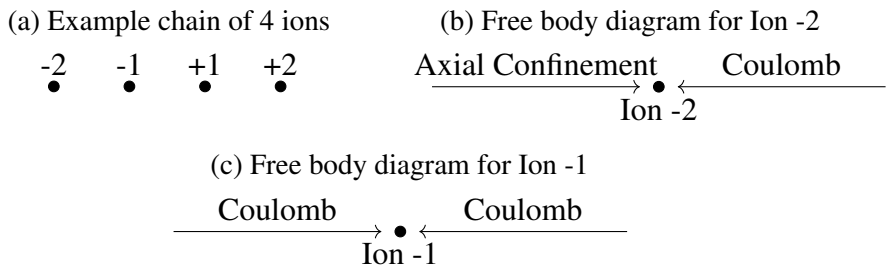


Figure 3.7: **Diagram of forces on 4 ions in a chain.** The horizontal direction is along $\hat{X}$, i.e. along the axis of the ion trap. The forces on Ions +1 and +2 are symmetric to the forces on Ions -1 and -2, respectively. These free body diagrams omit gravity, the RF confinement of the ion trap, and micromotion.

3.7.1.1 Trapped Ion Motion

A chain of ions, resting at the electromagnetic RF null point of the ion trap and properly cooled into a crystal, essentially is only subject to two forces: Coulomb repulsion (due to the positively-charged ions repelling each other), and the static DC potential of the ion trap (axial confining potential), as shown in Fig. 3.7. The residual motion of the ions is modeled as a quantum harmonic oscillator. Each oscillation axis has an oscillator motional mode, which is quantized as phonons. Based on the position of the ions and other static parameters, the equilibrium ion positions and harmonic motional modes (frequencies) can be calculated.

Along the trap axis, due to the tight confining axial field, the ions can be thought of as primarily stationary with some small harmonic oscillations. So we can then consider the instantaneous position of the i-th ion in a chain of ions as:

$$\hat{X}_i = \bar{X}_i + \hat{x_{0i}} \tag{3.10}$$

where $\bar{X}_i$ is the average, static position of the ion, and the operator $\hat{x_{0i}}$ is the i-th ion's harmonic oscillations. To obtain the position of the ions, we can calculate the mode participation coefficient of each ion b_{im} (collectively, the *participation matrix*), where i is the i-th ion, and m is the mode index $m = 1 \dots N$, so $\hat{x_{0i}} = \sum_{m=1}^{N} b_{im}\hat{\xi}_m$. Note that $\hat{\xi}_m$ denotes a phonon mode m, oscillating at frequency ω_m. Thus, the position of the i-th ion is: [37]

$$\hat{X}_i = \bar{X}_i + \sum_{m=1}^{N} b_{im}\xi_m^{(0)} \left(a_m^\dagger e^{i\omega_m t} + a_m e^{-i\omega_m t}\right) \tag{3.11}$$

3.7.1.2 Ion Optical Spin Qubit

We will summarily derive the Hamiltonian for a trapped ion in a well (Eq. (3.9)). A full derivation can be found in [93, Sec. 3.1.1-2]

The Hamiltonian for a non-interacting trapped ion is:

$$H_0 = H_{spin} + H_{motion} = \frac{1}{2}\hbar\omega_0\hat{\sigma}_z + \hbar\omega_m(a^\dagger a + \frac{1}{2}) \tag{3.12}$$

When a laser beam addresses the ion near-resonant with ω_0, it applies the Hamiltonian $H_{laser} = -\hat{d} \cdot \mathbf{E}$, where $\mathbf{E}$ is the applied electric field at the ion (location $\hat{x}$), and $\hat{d}$ is the dipole operator. For our $^{171}\mathrm{Yb}^+$ system, we couple $|F = 0, m_F = 0\rangle$ ($|0\rangle$) and $|F = 1, m_F = 0\rangle$ ($|1\rangle$) using correct polarization, so H_{laser} becomes $\Omega(\sigma_+ + \sigma_-)$.

$$H = H_0 + H_{laser} = \frac{1}{2}\hbar\omega_0\hat{\sigma}_z + \hbar\omega_m(a^\dagger a + \frac{1}{2}) + \Omega(\sigma_+ + \sigma_-) \tag{3.13}$$

[37] Note that the commutation of the phonon mode raising/lowering operators is $\left[a_m^\dagger, a_n\right] = \delta_{nm}$, i.e. the difference in frequency between phonon modes n, m.

When we transform into the interaction frame and apply the Rotating Wave Approximation (RWA), we obtain the interaction Hamiltonian H_I in Eq. (3.9).

When we substitute Eq. (3.11) into Eq. (3.9), perform a Taylor expansion and drop all except the 0th and 1st order terms, the Hamiltonian simplifies to:

$$H = \sum_{i=1}^{N} \frac{\Omega_i}{2} \sigma_+^i e^{i\left(\delta k \bar{X}_i - \mu_i t - \delta\phi_i\right)} \left(1 + i \sum_{m=1}^{N} \eta_{im} \left(\hat{a}_m^\dagger e^{i\omega_m t} + \hat{a}_m e^{-i\omega_m t}\right)\right) \tag{3.14}$$

If we set the detuning μ_i to particular values where $\mu_i = \sum_m l_m \omega_m$, we obtain the Hamiltonian for applying sidebands. There are a few important cases of these sidebands that we will consider: $\mu = 0, \pm\omega_m$

Carrier: $\mu = 0$. In this case, the Hamiltonian reduces to

$$H_{carrier} = \frac{\hbar\Omega_{nn}}{2} \left(\hat{\sigma}_+ e^{i\phi} + \hat{\sigma}_- e^{-i\phi}\right) \tag{3.15}$$

which is a **spin-only** transition, i.e. it does not change the phonon state at all, and is called the *carrier* transition. This rotates between the states $|\downarrow\rangle\ |n\rangle \leftrightarrow |\uparrow\rangle\ |n\rangle$, for some phonon state n. This transition is used for the $|0\rangle \leftrightarrow |1\rangle$ transition.

Blue/Upper Sideband: $\mu = +\omega_m$. This Hamiltonian simplifies to:

$$H_{BSB} = \frac{\hbar\Omega_{n+1,n}}{2} \left(\hat{a}^\dagger \hat{\sigma}_+ e^{i\phi} + \hat{a}\hat{\sigma}_- e^{-i\phi}\right) \tag{3.16}$$

This transition *adds* a single motional quanta when flipping the spin, transitioning (flopping) between $|\downarrow\rangle\ |n\rangle \leftrightarrow |\uparrow\rangle\ |n+1\rangle$.

Red/Lower Sideband: $\mu = +\omega_m$. This Hamiltonian simplifies to:

$$H_{RSB} = \frac{\hbar\Omega_{n-1,n}}{2}\left(\hat{a}\hat{\sigma}_+ e^{i\phi} + \hat{a}^\dagger\hat{\sigma}_- e^{-i\phi}\right) \tag{3.17}$$

This transition *removes* a single motional quanta when flipping the spin, transitioning (flopping) between $|\downarrow\rangle\ |n\rangle \leftrightarrow |\uparrow\rangle\ |n-1\rangle$. [38]

3.7.1.3 Ion Spin-Dependent Force (Multi-Qubit Gates)

If both the red Eq. (3.17) and blue Eq. (3.16) Hamiltonians are applied to an ion chain simultaneously, via bichromatic (two-frequency) beatnotes at frequencies $\omega_0 \pm \mu_i$ symmetrically detuned from the carrier, it will apply a spin-dependent force that will excite motion in the ion chain depending on the state of qubit(s). If the beatnotes are instead *asymmetrically* detuned from the carrier (at frequencies $\omega_0 + \mu_{i+}, \omega_0 - \mu_{i-}$), then the effective spin-dependent force frequency is $\mu_i = \frac{\mu_{i+}+\mu_{i-}}{2}$, which also creates a Stark shift equivalent to $\mu_{i+} - \mu_{i-}$. These two sidebands together create the following Hamiltonian:

$$H(t) = \frac{1}{2}\sum_{i,m}\eta_{i,m}\Omega_i\sigma^i_{\theta_i}\left[\hat{a}^\dagger_m e^{-i(\delta_{im}t+\psi_i)} + \hat{a}_m e^{i(\delta_{im}t+\psi_i)}\right] \tag{3.18}$$

This introduces two phases: a spin phase θ_i that determines the angle (phase) of the operator applied, and a motional phase ψ_i that only influences the phase of the optical forces, not the spin-spin (inter-qubit) coupling.

For our system, we primarily operate using a "phase-sensitive" configuration that is

[38] This red transition is the one that is used when performing sub-Doppler cooling (i.e. sideband cooling), because it is possible to selectively *remove* motional energy from the ion(s) by exciting only the red sideband transition(s).

sensitive to both movement of the ion(s) and optical path length changes. In this phase-sensitive scheme, the red and blue sidebands are applied through a beam with the same k vector for both sidebands ("co-propagating", i.e. the same laser beam). By contrast, the phase-insensitive scheme, which is not sensitive to ion movement or optical path length changes, requires that the sidebands propagate in different directions. [39]

3.7.1.4 Driving Raman Gates

Referring to Fig. 3.3, note that the difference in energy between the $|0\rangle$ level $|F = 0, m_F = 0\rangle$ and the $|1\rangle$ level $|F = 1, m_F = 0\rangle$ is $12.642\,812\,118\,466\,\mathrm{GHz}$ (hereafter called $f_{carrier}$). Thus, by driving the qubit with microwave RF signals at frequency $f_{carrier}$, transitions between $|0\rangle$ and $|1\rangle$ can occur, as described in Section 3.7.1.2 and Eq. (3.15). The downside of these microwave gates is that it is difficult to individually address ions. One factor is that the Lamb-Dicke parameter for microwave gates is very low, because the gradient of the microwave field (e^{ikx}) is only the small value ikx. Another factor is that the wavelength of this microwave frequency is relatively long ($\lambda = \frac{c}{f} = \frac{299\,792\,458\,\mathrm{m\,s^{-1}}}{f_{carrier}} = 0.023\,712\,482\,254\,02\,\mathrm{m} \approx 24\,\mathrm{mm}$) compared to a typical ion chain length of $< 70\,\mu\mathrm{m}$ for a 15-ion chain.

An alternative solution is to use an off-resonant stimulated Raman transition [113], which works by exciting the qubit using a pulsed $355\,\mathrm{nm}$ laser to off-resonantly couple to both the $^2P_{1/2}$ and $^2P_{3/2}$ manifold states. [40] If this laser is being operated as a frequency comb [125] with a difference between frequencies equal to the difference between the $|0\rangle$

[39] The phase-insensitive configuration effectively moves the need for phase stability to the RF waveform source. Having the ability to more-easily perform phase-insensitive gates was one reason why we were interested in implementing the RFSoC-based RF waveform system, described in Chapter 5.

[40] This means that the excitation is partway between both of these atomic manifold states.

& $|1\rangle$ (i.e. $f_{carrier}$), then the qubit will transition between the two states. This solution has the benefit that laser beams can be designed to optically address very small areas with high precision, which then allows addressing (i.e. targeting) individual ions, and thus performing operations only on the ions which are being illuminated by the laser(s).

We have both microwave and Raman methods available, but have decided to only use the microwave gate method for calibration and debugging purposes, where individual ion addressing is less important. We cannot use microwave gates for gates that rely on motional sidebands (e.g. the Mølmer-Sørensen gate in Section 3.7.3), because microwaves do not create enough force to adequately drive motional sidebands in our experiment.

3.7.2 Single-Qubit Operations

In the EURIQA Breadboard trapped ion quantum computer, our system is configured to execute non-co-propagating Raman transitions. If the Raman transition is tuned to the carrier frequency $f_{carrier}$ (Eq. (3.15)), a direct single-qubit gate can be executed.

The general gate that the Raman transition can apply is a $R(\theta, \phi)$ gate, which uses a detuning $\delta = 0$ for a carrier transition. The $R(\theta, \phi)$ gate is defined by the following unitary/rotation matrix:

$$R(\theta, \phi) = \begin{bmatrix} \cos\frac{\theta}{2} & -ie^{-i\theta}\sin\frac{\theta}{2} \\ -ie^{i\phi}\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{3.19}$$

This equation can be reduced to rotations about the X and Y axes:

$$R_x(\theta) = R(\theta, \phi = 0) = \begin{bmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{3.20}$$

$$R_y(\theta) = R(\theta, \phi = \frac{\pi}{2}) = \begin{bmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \tag{3.21}$$

Executing these gates functionally looks like playing sine waves at the correct frequencies to drive a carrier Raman transition, by generating a frequency difference of $f_{carrier}$ between the global and individual Raman laser beams. We call the continuous version of the $R_x(\theta)$ gate a *Rabi* pulse.

If the envelope of the sine waves is a square[41], we can calculate the rotation as a function of time t (i.e. the pulse duration), where t_π is the observed time that it takes to complete a $R_x(\pi)$ rotation (i.e. start in $|0\rangle$ and transition to $|1\rangle$). Note that we assume that t_π is measured experimentally. We distinguish between t_π and $t_{2\pi}$ (i.e. the time to make a full $|0\rangle \rightarrow |1\rangle \rightarrow |0\rangle$ transition): typically, $2t_\pi$ should be equivalent $t_{2\pi}$, though this might not always be true due to experimental details.

To demonstrate how t_π influences gate rotation for a time-variable Rabi pulse, we calculate the rotation angle as a function of t:

$$\theta(t) = \frac{\pi t}{t_\pi}, \tag{3.22}$$

[41] In reality, the envelopes for the RF drive for the individual and global beams will not precisely overlap, due to differing delays on the different beam paths. For example, on our system we have observed that the laser beam along the global path is delayed by $1.02\,\mu s$. This means that that RF drive for the global AOM cell must be turned on $1.02\,\mu s$ before the individual AOM drive to produce a square-envelope Rabi pulse.

then we substitute that into the R_x equation Eq. (3.20):

$$R_x(\theta) = R_x\left(\frac{\pi t}{t_\pi}\right) = \begin{bmatrix} \cos\frac{\pi t}{2t_\pi} & -i\sin\frac{\pi t}{2t_\pi} \\ -i\sin\frac{\pi t}{2t_\pi} & \cos\frac{\pi t}{2t_\pi} \end{bmatrix} = R_x(t, t_\pi) \tag{3.23}$$

These equations assume fixed amplitudes for the Raman drive (thus electric field), which is calibrated to obtain the t_π.

Other pulse shapes[42] than square are possible in this scheme, but require a calibration factor $\frac{t}{t_{\pi_{square}}}$ to equate their duration (or enclosed phase) to the equivalent square pulse. Other pulse shapes can be useful to avoid higher-frequency harmonic tones that are unintentionally driven with certain pulse shapes [126]. For example, a square pulse has many higher-order harmonics, which can be demonstrated by transforming a step function into the Fourier domain. These higher-order harmonics can excite unintended motional modes of the motional harmonic oscillator, causing a loss in fidelity.

We can calculate the duration calibration factor by realizing that the square Rabi pulse (or single-qubit $R_x(\theta)$ gate) actually accumulates θ as an integral of the accumulated Rabi drive $\Omega(t)$. For a square pulse at a constant Rabi rate ω (i.e. constant amplitude), we can calculate

$$\theta(t) = \int_0^t \omega\, dt = \omega t \tag{3.24}$$

If we use t_π as the desired rotation angle, then the above becomes $\theta(t_\pi) = \omega t_\pi$

[42] E.g. DRAG [126, 127, 128], Gaussian, or Hyperbolic-Tangent (tanh).

This can be generalized to an arbitrary function:

$$\theta(t) = \int_0^t \Omega(t)\, dt \tag{3.25}$$

Using these equations, it is relatively simple to calculate a normalization factor for an arbitrary pulse shape $\Omega(t)$ that you would like to execute:

$$\theta_{square}(t_{\pi_{square}}) = \theta_{arb}(t) \tag{3.26}$$

$$\omega t_{\pi_{square}} = \int_0^t \Omega(t)\, dt \tag{3.27}$$

and then solving for $\frac{t}{t_{\pi_{square}}}$.

Finally, while the above $R(\theta, \phi)$ is the basic gate that we can execute, we have found that it is generally advantageous to use composite pulse sequences instead of direct R gates. Composite pulse sequences, such as the SK1 sequence [129] or the BB1 sequence [130], make a quantum gate somewhat insensitive to common errors or mis-calibrations, at the cost of increased gate time. [43] We have demonstrated high-fidelity gates using SK1 pulses, with average *in*fidelity [44] of $3.4(8) \times 10^{-4}$ per Clifford, equivalent to an infidelity of $1.8(3) \times 10^{-4}$ per native single-qubit gate ([93, Section 5.3.3]).

So far we have addressed $R(\theta, \phi)$ gates, which would be sufficient to produce arbitrary-angle quantum states and (the single-qubit portion of) universal quantum com-

[43] The long coherence times of trapped ions compared to the duration of a gate such as $R_x(\pi)$ make composite pulse sequences very attractive. In this regime, the primary source of error is from executing a gate, and not from the qubit just "idling". Hence it makes sense to spend extra time to perform a higher fidelity gate.

[44] Infidelity is defined as $1 - F$, where F is the fidelity.

putation. However, trapped ions can also implement a native virtual R_Z gate by advancing the phase of the drive frequency (i.e. applying a phase offset to a sine wave) [128]. The advantage of a virtual R_Z is that it does not require any time to execute and can be implemented entirely in software, so it should be of extremely high fidelity. Therefore using a virtual R_Z gate is highly desirable for improving circuit fidelity. [45]

3.7.3 Multi-Qubit Operations

Single-qubit operations alone are not sufficient for a useful universal quantum computer. Multi-qubit gates must also be available. In a trapped ion quantum computer with long chains, the primary way of performing a multi-qubit gate is:

1. *Entangle* qubits' spin with the motion of the collective ion chain.
2. *Accumulate phase* in the motional modes of the ion chain.
3. *Disentangle* the qubits' spin from the motion.

There are many ways of performing each of these steps, especially the accumulation step, but the most common trapped ion multi-qubit gate schemes use some variant of the above steps.

The *de facto* standard quantum gate today, and the primary one used on this experiment, is the Mølmer-Sørensen gate [2, 1, 3, 131]. This gate is designed to perform well even when the harmonic oscillator of the trap is noisy. [46]

[45] Arbitrary-angle R_Z gates also have the advantage that they can be combined with an $R_x(\frac{\pi}{2})$ to produce an arbitrary-angle gate (any $SU(2)$ gate) [128].

[46] i.e. When there are high & unknown amounts of phonons in each of the motional modes of the trap.

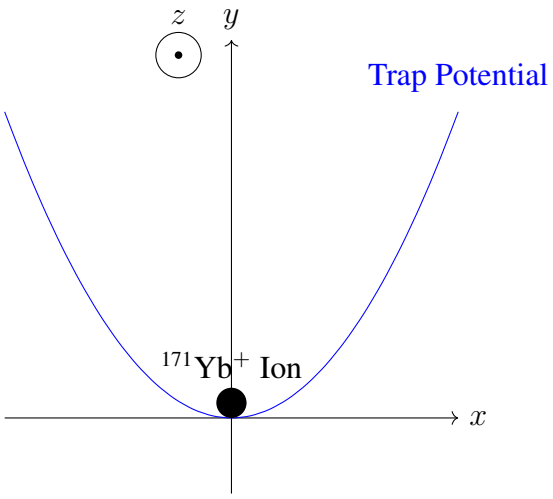


Figure 3.8: **Axes of an ion in a harmonic trap.** The x axis is the axis of the trap. Vibration along the y (vertical) and z (out of page) axes is called the *radial* (*transverse*) modes. A single ion trapped in this potential will have **3** motional modes, each with their own energy and number of phonons.

Consider a single ion trapped in a harmonic trap, as in Fig. 3.8. This ion has three principle axes that it can vibrate in: x, y, and z. If we extend this concept to a chain of N qubits (Fig. 3.9), the overall chain will have $3N$ motional modes (one for each of the N normal modes in each of the 3 dimensions).

We elect to perform gates using the radial modes [132, 68] (i.e. the modes that are not along the trap axis), because:

- *Higher Frequency Modes* when compared to axial modes.
- *Less subject to* $\frac{1}{f}$ *noise* by nature of being higher frequency. One effect of being less subject to $\frac{1}{f}$ noise is that the mode will heat (i.e. increase in average phonon population) more slowly from sources with a $\frac{1}{f}$ distribution, which are most background noise sources.
- *Addressing*: Allows individual addressing using perpendicular laser beams, with

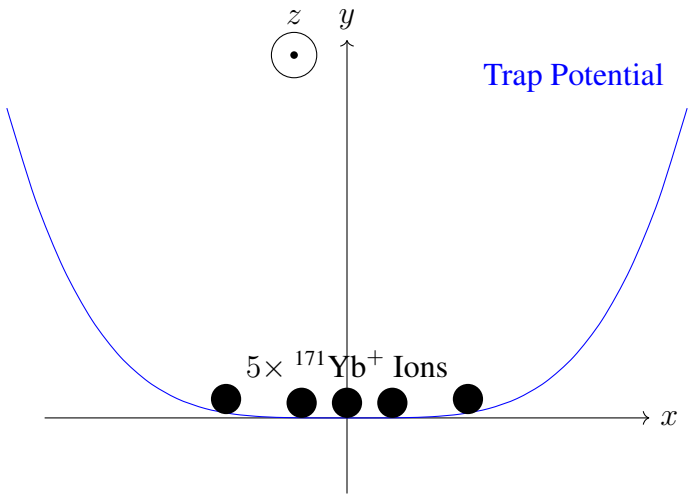


Figure 3.9: **Five ions trapped in a quartic potential.** Relative to Fig. 3.8, this image has a total of five ions. With several ions, we elect to use a quartic (not harmonic) trapping potential to approximately equally space the ions in the chain from each other. We also use the term "chain" to describe a crystal of ≥ 2 ions that are trapped in a well. Having a chain will break symmetry in an axis, so it now makes sense to add the concept of the "trap axis". In this frame, vibration along the x axis is said to be *axial* motion, and the vibrational modes in that direction are called the *axial* modes. With multiple ions, each additional ion will add one motional mode in each axis. Thus, a two-ion chain will have a total of 6 modes; five ions will have a total of 15 modes. For a chain of ≥ 2 ions, the axial mode will be the lowest-frequency mode, and generally subject to the highest heating/number of phonons.

minimal optical crosstalk between ions.

- *Cooling*: these modes can be very effectively cooled using sideband cooling.

The Mølmer-Sørensen gate two-qubit gate unitary is sometimes called an XX, or a R_{XX} gate. This gate is defined as

$$XX(\chi) = \begin{bmatrix} \cos(\chi) & 0 & 0 & -i\sin(\chi) \\ 0 & \cos(\chi) & -i\sin(\chi) & 0 \\ 0 & -i\sin(\chi) & \cos(\chi) & 0 \\ -i\sin(\chi) & 0 & 0 & \cos(\chi) \end{bmatrix}, \tag{3.28}$$

where the parameter χ is the angle of the XX gate rotation. [47] We can scale the gate angle (to first-order) by varying the power (waveform amplitudes), which allows calibrating the exact angle applied. Typical two-qubit gate times are $\leq 250\,\mu s$ for fully-entangling gates on a chain of 15 ions.

The gate time is a direct consequence of the ion mode frequencies used in the gate. Intuitively, two-qubit Mølmer-Sørensen gates are driving all the ion chain modes simultaneously, which is causing each of them to make loops in phase space. [48] We can ignore the modes that are far-detuned (not strongly driven) because they are making small loops so any errors will be small, and choose a duration where the primary drive modes will complete a full 2π rotation and return to zero.

The problem of designing gates to satisfy the scheme of an Mølmer-Sørensen gate

[47] χ is typically in the range $-\frac{\pi}{4} \leq \chi \leq \frac{\pi}{4}$, because the gate unitary is fully-entangling at $|\chi| = \frac{\pi}{4}$.

[48] Phase space is effectively the axes of position (x) and momentum (p) of an ion for each of the harmonic oscillator modes.

has been treated elsewhere, such as [133, Sec. 4.1] and [93]. With properly-designed & -calibrated two-qubit gate waveforms, two-qubit Mølmer-Sørensen gate fidelity $\geq 98.5\,\%$ has been demonstrated on our experiment.

3.8 Experiment Cycle

Stepping back from details of how operations are performed on a quantum computer, it is helpful to consider the overall sequence how a quantum circuit is executed on a trapped ion quantum computer, shown in Fig. 3.10.

This sequence can be broken down into the following categories:

1. *Pre-Experiment*
2. *State Preparation*
3. *Execution*
4. *Measurement*

3.8.1 Pre-Experiment

To perform quantum operations an ion trap quantum computer, one must first ensure that it does in fact contain trapped ions. On our system, where the vacuum chamber (and thus the surrounding environment) is at room temperature ($\approx 70\,^{\circ}\mathrm{F}$, or $\approx 21\,^{\circ}\mathrm{C}$), a single ion will typically be trapped for a period of several days, while a chain of about 15 ions will be trapped for a period of a little less than an hour. [49]

[49] The lifetime of an ion chain is dependent on many different factors, including:

- *Background pressure* of the vacuum chamber. Higher pressure means more gas atoms are present, and thus a higher incidence of collisions between the ion chain and the background gas atoms. Assuming the other requirements below are met, the pressure will have the most direct impact on chain lifetimes.
- *Temperature* of the background gas. Lower temperature is better because collisions are less energetic

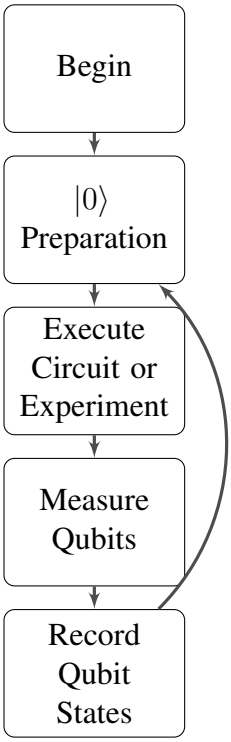


Figure 3.10: **Diagram of a standard experiment cycle using trapped ion qubits.** This diagram is only the part of the cycle that interfaces with the ions directly. Other preparation, calibration, and analysis steps will need to be performed external to the experiment runtime on the ions. Each loop iteration here is commonly referred to as a *shot*.

Once the correct number of ions are available, and the appropriate calibrations have been performed [119], then the quantum computer is ready to execute a circuit. Once the desired circuit or experiment is selected, the circuit must be processed to turn it into a sequence of actions that can be executed on the quantum computer. This topic will be discussed in more detail in Chapters 4 and 6.

3.8.2 State Preparation

After all of the sequence preparation has been performed, a quantum experiment begins by preparing the qubit(s) in $|0\rangle$. This was discussed in detail in Sections 3.6.2.1 and 3.6.2.2.

The following steps are executed in sequence (Fig. 3.11), which together take a large percentage of the overall duration of a given shot.

Once the qubit(s) have been prepared in $|0\rangle$, then it is time to perform operations on them.

3.8.3 Experiment Execution

Executing a quantum circuit is conceptually very simple for most quantum computers. Most quantum computers use a waveform system that pre-computes the actions that they will execute during the *pre-experiment* phase (Section 3.8.1). Then the circuit execution

---

and thus give the chain less energy.

- *RF Confining Fields*: An RF potential is applied to trap the ions. Changes in this potential can cause ion ejection.
- *DC Voltages*: The potential well that the ions are in can be reduced in height depending on the voltage settings. A shallow trap depth can put the ions closer to being removed from the trap.
- *Laser stability*: To remain cold (i.e. low-energy), the ions must be continually cooled. If the cooling laser frequency shifts so that it is no longer cooling, then it will actually *heat* the ions out of the trap.

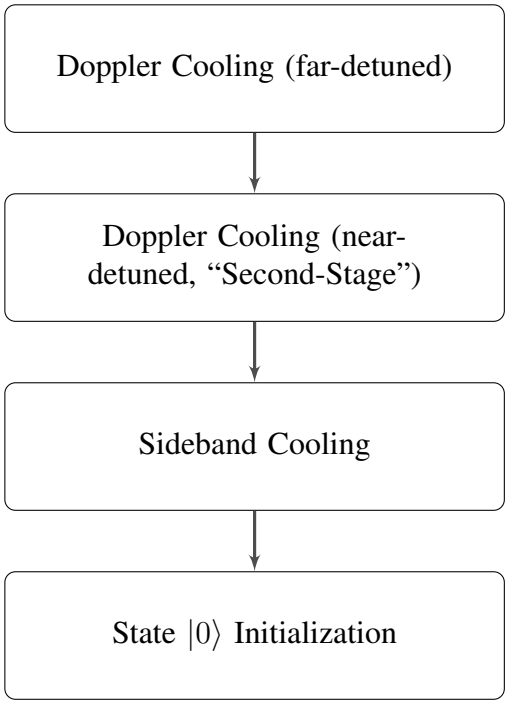


Figure 3.11: **Sequence of Qubit State Initialization.** These are the steps from Section 3.6.2.2.

is just a matter of triggering the appropriate waveform/execution sequence to begin.

This model works well for the majority of algorithms that are being executed, but it does not hold for sequences containing conditional gates, such as quantum error correction algorithms. To handle conditional execution of this form, tight coupling must exist between collecting the quantum state and using that measurement to select the appropriate gate(s). In our system, that requires sending the measurement result to the waveform generator in real time as soon as possible after collecting the measurement result (see Section 5.5.5).

3.8.4 State Measurement/Readout

Whether a quantum circuit includes conditional quantum gates or not, the state of all qubits is measured at the end of every experiment. This was covered in more detail in Section 3.6.3, and summarized in Fig. 3.12. In short, the qubits are illuminated with resonant $369\,\mathrm{nm}$ light for the ${}^2S_{1/2}\ |F=1, m_F=0\rangle$ to ${}^2P_{1/2}$ transition, which causes the ions to fluoresce and emit $369\,\mathrm{nm}$ photons only if the qubit is in $|1\rangle$. Those photons are collected with a high-NA lens, and routed to individual PMTs which output digital signals when they detect a $369\,\mathrm{nm}$ photon. The total quantity of registered photons for each ion is recorded, and a simple discriminator is applied to determine if the qubit is in $|0\rangle$ or $|1\rangle$ based on the number of photon counts collected. This information is then saved for later processing.

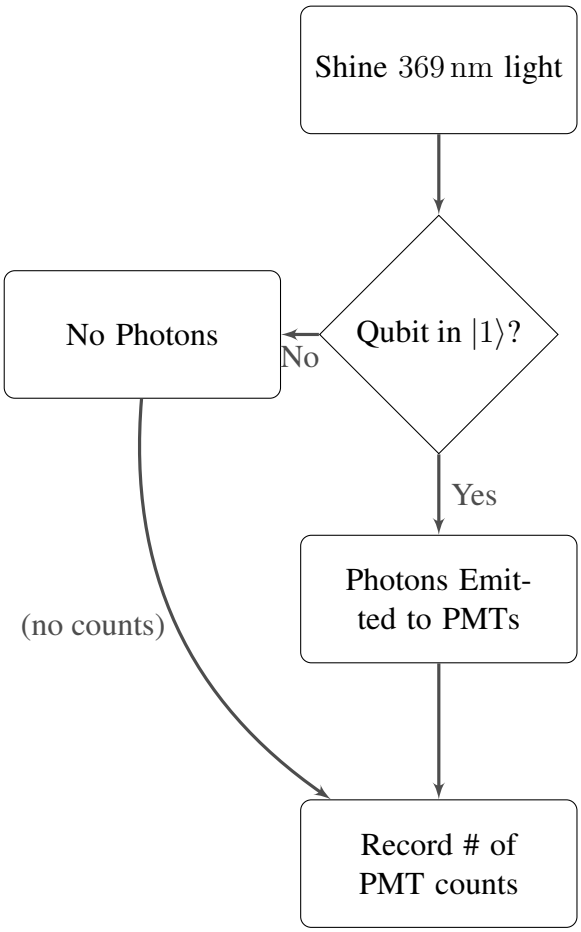


Figure 3.12: **Sequence of Qubit State Readout.**

Chapter 4: Control System Design

4.1 Motivation

In order to turn a set of quantum computing hardware into a functioning quantum computer, all of the devices that are used in an experimental shot must be synchronized and controlled together. This complexity generally varies depending on the type of qubit used. Superconducting qubits will generally have simpler control systems in terms of number of devices, because their qubits are persistent, so most of the control complexity becomes driving the qubits using RF signals. Trapped ion control systems tend to be more complex because there are more steps to prepare, execute, and measure qubits during an experimental sequence.

Generally, the control system is responsible for (assuming a single linear sequence with no partial measurement):

1. Ensuring that the qubits are in the system
2. Preparing the initial quantum state of the qubits
3. Performing operations on that quantum state
4. Measuring the state of all of the qubits

5. Recording measurements

6. Preparing for the next repetition (typically called a shot).

4.2 Quantum Computers as Embedded Systems

In the engineering world, control systems are typically the domain of embedded systems & controls engineers. These engineers are responsible for sequencing many different real-world input & output devices together, in order to produce some real-world output. Examples of embedded systems can range from automated factory assembly lines to a laundry washer.

Control of a quantum computer requires essentially a very high-precision embedded system, which must track and control the state of both a distributed electronics control system, as well as control over the qubits themselves.

4.2.1 Challenges of Quantum Embedded Systems

Designing a control system for a quantum computer has several requirements and challenges that are relatively unique, which makes the process a challenging & research-worthy endeavor. Specifically, quantum computing control systems require:

- *Cross-domain knowledge*: typically electrical & mechanical engineering, physics, & software engineering.

- *Precision & High-frequency Analog signals*: Quantum computers are extremely sensitive to any Electromagnetic fields or noise, but especially to phase and ampli-

tude variations.

- *Precise timing*: $100\,\mathrm{ns}$ or finer precision, depending on the system. [1]
- *Synchronizing* many digital & analog inputs/outputs.
- *Repeatability*: Gathering state statistics typically requires collecting many "shots" of a single circuit, which assumes that the system has remained constant and drift-free for the duration of the data collection
- *Varying timescales*: each of the following requires operating timescales ranging from nanoseconds to hours or days: laser locks, optical coherence, lab HVAC drifts, optics burning.

Though this entire requirement set of quantum computers are unique, a quantum computer control system is qualitatively similar to radar systems, and digital CPUs. A radar system, especially Active Electronically-Scanned Arrays (AESAs), requires precise phase control across many different channels, as well as real-time control to avoid jammed frequencies and to interpret the radar results in real time. Radar systems and quantum computers share the similarity of being conceptually simple, but fiendishly complex in practice. The theory of both types of systems is well-established, and can be taught at the undergraduate level. However, bringing each system to a level of reliability & high-performance where it is truly useful requires a significant engineering effort. That is not to say that an academic quantum computer control system is necessarily simpler. Academic quantum computer control systems have the further challenges of:

[1] Superconductor quantum computers have typical gate times of $\approx 60\,\mathrm{ns}$[134], so timing resolution of $\leq 10\,\mathrm{ns}$, and typically $\leq 5\,\mathrm{ns}$, is highly desirable.

- *Constrained Manpower*: While well-funded commercial teams might have enough people that certain people or even sub-groups can specialize in a particular problem, academic groups are typically stretched thin, and each person must take responsibility for several subcomponents.

- *Complexity*: The publishing-first nature of academia tends to discourage long-term projects or usability improvements. Most academic projects tend to involve just a handful of graduate students at most, and are limited in scope to what is needed for the next incremental step or paper.

- *Extensibility*: Academic software must be able to be improved over time to adapt to new requirements, but that extensibility requires tradeoffs over more-specialized software, such as slower runtime, more development effort, etc.

- *Heterogeneity*: Academic labs typically consist of mix-and-match hardware, which was designed over time to a constrained budget, by people people with varying amounts of experience.

- *Tight development cycles*: Academic groups typically try to iterate quickly, and want to just to do proof-of-concept. It takes a considerable amount of work to bring a project from working in specific circumstances (e.g. $50\,\%$ reliable, undocumented, etc.) to working reliably $100\,\%$ of the time.

- *Communication*: Academic labs are primarily composed of graduate students, who will eventually leave after a period of $\approx 5\,\mathrm{years}$. The new tools tend to be developed by junior graduate students, who are less experienced and have little need for

documenting design decisions, as well as little institutional incentive for such documentation. Projects tend to primarily be documented in graduate theses, but the downside is that this is the only remaining reference once that person leaves, and can be left in varying states of detail.

4.3 Comparing Existing Qubit Support System Requirements

The table in Table 4.1 lays out the different steps of an experiment (laid out in Section 3.8), comparing the hardware that is required for superconducting vs trapped ion qubits.

Experiment Stage	Trapped Ion Qubits	Superconducting Qubits
Qubit "Loading"	Photoionization Laser Trap Voltages, Cooling Lasers Trap RF	Dilution Refrigerator Temperature Probe Resistance Meter
Qubit Initialization	Cooling Lasers RF Source (Sideband Cooling) Pumping AOM	RF Source DC Bias Lines
Quantum Operations	RF Source Cooling Lasers Trap Voltages	RF Source RF Switches
State Readout	Readout Lasers/AOMs	RF Source Cryo Amplifiers Mixers

Table 4.1: **Comparison of Required Control Hardware for Trapped Ion & Superconducting Quantum Computers.** This list is not meant to be exhaustive, but simply indicative of the differing requirements between a leading-edge superconducting quantum computer vs a similar quality trapped ion quantum computer.

4.4 Control System Realms

For the remainder of this chapter, we will primarily address control systems and the specific requirements for Trapped Ion Quantum Computers. When considering control of a trapped ion quantum computer, it is important to understand the different timescales that the control system needs to handle, which range in scale from order of days to order of nanoseconds. For the purposes of this discussion, one "experiment" denotes executing one quantum circuit with varying parameters, repeating the sequence for each parameter some number of times (typically called "shots" or "repetitions").

We can divide the control system roughly according to these categories, which are also shown in Fig. 4.1:

- *Daily Control*: These are items that have timescales of hours to days.
- *Experiment-Time Control*: These items need to be controlled on the scale of an "experiment", which currently typically takes place over a period of seconds to minutes.
- *Coherent Control*: These are the quantum operations that happen *during* a quantum "experiment". Because any error in these operations directly affects the unitary that is executed, this is typically the most strict and precise segment of the entire experimental sequence.

Note that this framework completely neglects the important discussion of designing the physical quantum system itself. While that is not a primary focus of this thesis, it is discussed more in Chapter 3, and more thoroughly in a wealth of other resources [93,

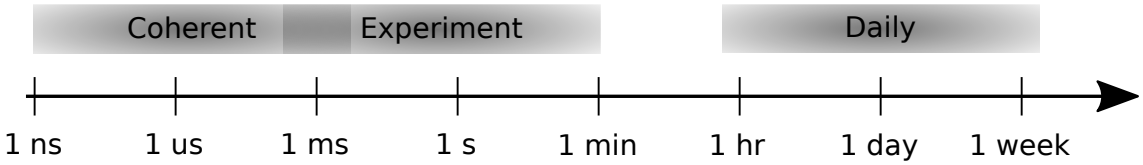


Figure 4.1: **Diagram of the various experimental timescales and their corresponding control realm.** The gap between experiment and daily is filled in our experiment via control software, which manages issues at this timescale via e.g. ensuring that the appropriate number of ions is present.

67, 113, 133, 135, 114]. For the remainder of this chapter, let us assume that there is a pre-built trapped ion quantum computer that is effectively a quantum black box, and a control system is needed to operate it.

4.5 Ion Trap DAC Control

One of the advantages of a surface electrode trap is the ability to have minute control over the trapping potential that a chain of ions collectively experience. With enough electrodes, you can have fine control over voltage potentials in most directions of the ion chain, and thus have fine control over the various vibrational frequencies (modes) of the ion chain. A surface electrode trap also generally has separate zones for loading and performing quantum operations.[2]

However, it is not enough to simply have a surface electrode ion trap (often shortened to *surface trap*) with many electrodes. There must also be a method for controlling those electrodes. This process is essentially applying DC voltages between the trap elec-

[2]Zone separation is primarily due to the need for high optical access & low noise where quantum operations are performed, which allows tight focusing of laser beams for individual control. This separation also allows for maintaining a mostly pristine trap surface near the quantum region. Loading ions is a dirty process in that it involves spewing a beam of atoms towards the surface of the trap, some of which will attach to the surface and can become charged, which applies undesired potentials to the ions. This effect can be minimized by spatially separating the "dirty" loading zone from the "clean" so-called *quantum zone*.

trode & the trap ground plane.[3]

This electrode control scheme must not just output static DC voltages, but it must be able to change in timescales ranging from real-time to every few minutes, to enable:

- *Voltage Calibration*: the DC voltages must be periodically updated to adjust for drifts in the ambient electric fields.[4]

- *Ion Shuttling, Merging & Swapping*: As will be described in Section 8.2, a common operation for an ion trap quantum computer is to move a chain of ions around as a single unit, or to split that chain into sub-sections and then move the smaller chain/ions independently. Performing any of these operations on an ion chain typically involves outputting a pre-computed sequence of voltages across all electrodes, and then stepping through that sequence to move (or reshape, depending on the operation) voltage potential wells from physical location (configuration) A to B. For simplicity, we will collectively call these *shuttling operations*, because the way that they are processed by stepping through voltage outputs is identical, only differing in the precise voltages needed to perform each type of operation.

As with the majority of the control system, there is a design trade-off in the controllability of the trap DAC. In this case, we must trade-off number of electrodes controlled, response time, real-time control, precise control and memory space. For our particular application, we need to drive the ≈ 100 DC electrodes of the Sandia HOA with $\leq |10\,\mathrm{V}|$, updating every few tens of $\mu\mathrm{s}$, and be able to store enough voltage outputs to perform all

[3] In the case of our Sandia HOA, the recommended voltage range to apply to DC electrodes is $\pm 10\,\mathrm{V}$. Applying voltages outside of this range risks damage to the ion trap.

[4] We believe these are primarily due to photoelectric effects from our high-powered $355\,\mathrm{nm}$ laser.

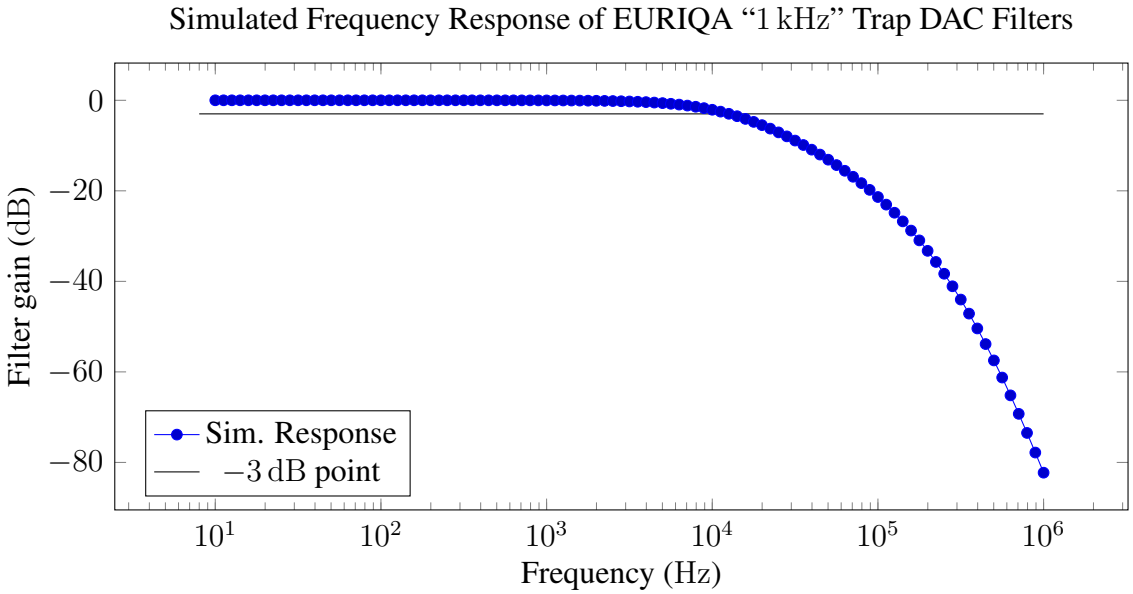


Figure 4.2: **Simulated Frequency Response of EURIQA $\approx$ 1 kHz Filter.** This is a straightforward 5th-order lowpass RC filter.

the desired shuttling operations ($\approx$ 4 MiB). We use a $\approx$ 1 kHz 5th-order low-pass RC filter on the output of each channel of our custom 112-channel DAC, which limits both the noise on the electrodes and the rate at which the electrode voltages can change. The simulated frequency response of this filter can be seen in Fig. 4.2.

4.6 Digital Control System

To execute the entire experiment cycle that was described in Section 3.8, we need a device to sequence all the output events, collect input events, and record of the information. We use the ARTIQ (Advanced Real-Time Infrastructure for Quantum physics) ecosystem [25, 136] for this function. ARTIQ is specifically designed for quantum computing applications, and combines the flexibility of Python with real-time control over hardware devices. ARTIQ is designed around executing *experiments*, which go through four stages

sequentially: `build()`, `prepare()`, `run()`, `analyze()`.

ARTIQ is effectively an embedded system network with a few primary components, which is described below and visualized in Fig. 4.3.:

- *"Host" or "Master" PC*: This is responsible for the majority of the computational load, and acts as a control server for the rest of the ARTIQ ecosystem. All experiments begin on this device.

- *Core device*: This is an FPGA (such as the Kasli FPGA carrier board [137]), which runs a real-time program. The ARTIQ FPGA is comprised of 3 key components: a real-time processor[5], a network processor (to offload non-real-time operations from the real-time core), and FPGA gateware blocks for communicating with external devices. [6] Example external devices include DDS's, digital I/O signals, ADC/-DACs, and any other device that has a digital interface. [7] This system allows queueing input/output events from the real-time CPU, which are then executed at

[5]The CPU is based on the OpenRISC design [138], though recent work has been put towards using built-in Arm processors on ZynQ FPGA boards.

[6] Many embedded systems do not operate natively in meaningful SI units (e.g. Hertz, Volts, etc). Instead, they operate in binary units which roughly correspond to a physical unit. Following the convention laid out in ARTIQ, we call these *machine units (mu)*. Machine units must be used because digital computers cannot represent arbitrarily-precise numbers, and instead must approximate it as close as possible. For example, it would take an infinite number of bits to represent π, so instead π will typically be rounded to $\approx 32\,\mathrm{bit}$ of precision, depending on the exact floating point scheme [139]. Floating-point calculations are typically slow and expensive to execute on a lightweight real-time embedded device, so these are generally converted into a fixed-point binary number. The range of this binary number should precisely match the capability of the hardware device. For example, consider a DDS IC such as the Analog Devices AD9910. The datasheet [140] gives the equation for the output frequency as $f_{out} = \left(\frac{FTW}{2^{32}}\right) f_{SYSCLK}$, where FTW (Frequency Tuning Word) is the *machine units* representation of the frequency (a $32\,\mathrm{bit}$ register), and f_{SYSCLK} is the input clock to the AD9910. When $FTW = 1$, the right side of the equation is equivalent to the frequency resolution. This equation can be slightly extended and generalized for an arbitrary output range for *mu* calculations:

$$out = \left(\frac{n_{mu}}{2^{n_{bits}}} * (out_{max} - out_{min})\right) + out_{min} \tag{4.1}$$

where n_mu is the number in machine units, n_{bits} is the number of bits of precision, and out_{min}, out_{max} are the minimum and maximum output values, respectively.

[7]If you are able to write drivers and gateware blocks for them.

the appropriate timestamp via the FPGA blocks.

- *Auxiliary Core device (optional)*: The Kasli system allows for digital I/O expansion over a custom ARTIQ protocol called DRTIO (Distributed Real-Time Input/Output).

- *Client PC*: The ARTIQ interface on the Host PC is available remotely over network protocols (IP). This lets multiple users at multiple PCs execute ARTIQ experiments interleaved.

- *Laboratory Instruments*: More complex quantum computers require knowledge of and control of more laboratory devices to achieve high fidelity and complex operations (such as shuttling). These instruments can be connected into an ARTIQ setup via any protocol available via Python (e.g. IP, VISA, USB, etc.) for software control, or real-time digital protocols such as SPI/I2C depending on the instrument and control needs.

4.7 ARTIQ Experiment Design

ARTIQ provides a very barebones interface out-of-the-box, leaving it to users to design and implement their own experiment framework. ARTIQ does provide some standard features, such as:

1. *Core Device interface*: ARTIQ can compile and run your Python programs on the FPGA core device.

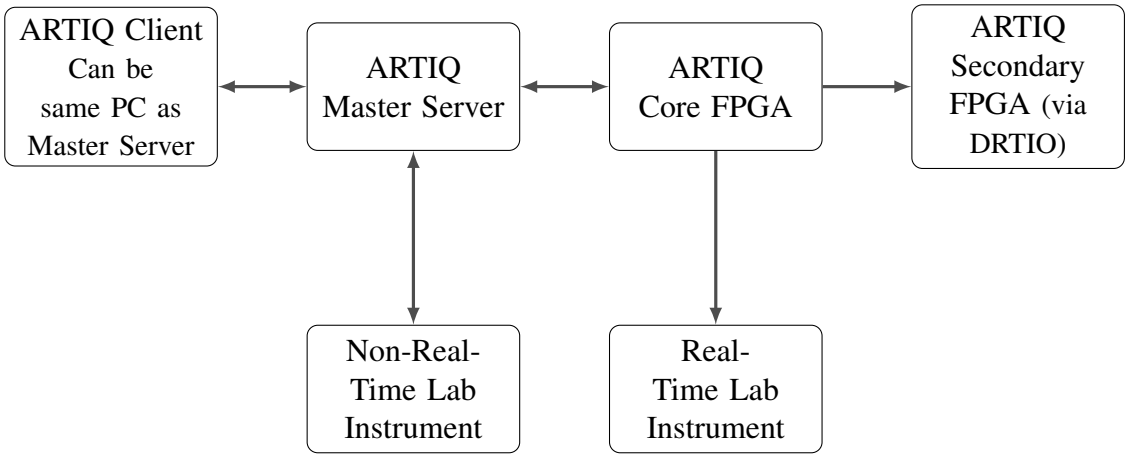


Figure 4.3: **Example ARTIQ Laboratory network.** Connections between devices indicate that direct communication can be made between the two devices over some protocol (e.g. Ethernet, or a real-time protocol like SPI) during the course of an experiment. Note the direction of (typical) flow is denoted by the arrowhead direction. Practically, the ARTIQ Core FPGA can communicate bidirectionally with any real-time device, but practically the Core FPGA will act as the primary controller/sequencer for any other devices during an experiment.

2. *Dataset values*: ARTIQ can save/load values from storage, which allows persistent values that are shared across experiments.

3. *Network interfaces*: The ability to interface with distributed Python programs, which are accessible via different IP addresses and ports.

4. *Device Driver library*: for some subset of hardware.

However, this capability falls far short of meeting every experiments' needs. Particularly, it is missing the crucial capabilities to interface with all the devices in a given hardware setup, as well as the higher-level software interfaces for commanding those devices in a logical way. For example, the DAC described in Section 4.5 can be commanded to output a specific voltage for each output channel. But that level of control is too fine-grained, and can hinder a conceptual understanding of what the overall purpose is. A

more useful command would be something of the sort "Shuttle ion from A to B", which is then decomposed into a sequence of voltage commands. However, even if a driver for the Sandia DAC was provided in ARTIQ, this high-level interface is left to users to define, architect, and implement.

Some research groups have developed frameworks such as DAX [141, 142], but these did not exist when this requirement was discovered. Our infrastructure ended up being developed somewhat ad-hoc[8], but included writing drivers for the wide range of hardware that we had available in the laboratory, including:

- Power supply for the magnetic field electromagnetic coils and the Yb thermal oven.
- Sandia 100-channel DAC. This includes both real-time control and high-level control for commanding shuttle operations between different voltage configurations.
- 8-channel DAC to control positioning piezos.
- Waveform generators: both DDSs, AWGs, and the RFSoC (Chapter 5).

Further, we also developed the higher-level interfaces to incorporate each of these devices into our experiment, and calibration value infrastructure that could be easily extended and incorporated in new routines. For the high-level interfaces, we generally organized devices by functionality instead of physical device. For instance, a high-level interface for performing Doppler cooling would only expose methods such as the ability to turn it on for a duration, or to change the laser frequencies used for the cooling. Un-

[8]The design and implementation of these drivers was further complicated by a set of concerns that included: physical location, timing concerns, software driver compatibility issues (e.g. Windows/Linux low-level drivers), limited development time, etc.

der the hood, it would then execute the necessary commands to the required devices to execute the high-level goal.

We were successfully able to define, implement, and demonstrate these interfaces and drivers working in our laboratory environment. Altogether, we were able to demonstrate a working quantum computer control stack that was able to convert quantum circuits and calibration routines into the control signals for running a quantum computer.

Chapter 5: RFSoC-based Coherent Control System

This chapter will describe the implementation of a waveform generator for ion trap quantum gates on a Xilinx RFSoC. An RFSoC is a system-on-a-chip (SOC) combining an Arm CPU, a Xilinx FPGA, and RF ADC/DACs for outputting and sampling radio-frequency signals.

5.1 Need for Flexible Waveform Control

As described in Section 3.7, an RF source is needed to perform qubit operations for trapped ion quantum computers. While there are a variety of waveform generation solutions, many of the Commercial Off-the-Shelf (COTS) solutions are not well-suited for use in quantum computing. Further, even the relevant COTS solutions will typically need some modifications or upgrades to meet the requirements of the range of quantum experiments that we are interested in performing.

The RF system to drive our quantum gates must be able to meet our research needs. These research needs change over time, especially because the expected lifespan of these systems is on the order of ten years. The RF system is being constructed for an academic setting where requirements are based on the scientific avenue that we are currently pursuing, which will tend to change on the order of one year. Thus, we need a system that

is flexible & future-proof, while achieving the high performance that we need to demonstrate many of these scientific goals.

Despite the rigorous quality & flexibility demands that academia is placing on this system, it must still be designed for end-users, so programmability and clear operation are important. For many of the quantum applications we are considering, the quantum computer must be capable of operating at high-fidelity ($>= 98\%$) for hundreds of circuits. It must not only be capable of handling this many circuits, but must also quickly generate the appropriate sequence of waveforms for each circuit & give useful feedback if there is something wrong.

5.2 Waveform Generation Trade-offs

When we were exploring some waveform generator alternatives, certain solutions would generate the waveforms in about 90 seconds, but the actual data collection would only take about 5 seconds, which is simply inefficient and frustrating to use. It should be obvious that a solution where it takes approximately $10\times$ longer to compute a sequence than it takes to execute it has plenty of room for optimization. This chapter will explore some of the design trade-offs behind different solutions to this problem, and the approach that we have taken.

5.2.1 Waveform Generation Requirements

At a technical level, designing an RF control system that works for quantum control systems has a few challenging, and sometimes conflicting, requirements:

- *Phase control*: phase errors on an output that drives the qubits will cause direct qubit decoherence, which leads to a decrease in fidelity.

- *Duration Range*: A wide range of experiment and waveform durations must be supported. For example, the pulses to execute a Ramsey sequence [18] to measure decoherence rates will last approximately the duration of the qubit's T_2^* time, which is greater than $1\,\mathrm{s}$ for $^{171}\mathrm{Yb}^+$ [90]. On the other hand, the RF system must also be capable of outputting pulses of short durations, for example less than $500\,\mathrm{ns}$ for scanning the duration of a full-power Rabi pulse. [1] Thus, pulses with durations ranging from approximately $[1\,\mathrm{ns}, 10\,\mathrm{s}]$ must be supported by the RF system.

- *Pulse Shaping*: For various reasons, primarily due to avoiding harmonics and unwanted frequency interference, it is generally preferable to avoid square-shaped pulses when driving qubits. There are a variety of pulse-shaping schemes, but in general high-fidelity quantum operations require shaping the up-/down-ramp of pulse output.

- *Fast Preparation*: The time to generate and upload the waveform sequence must be reasonably short. Ideally, this would be the same or less than other experimental setup delays. In practice, this means that the desired compile and upload time for an experiment's RF waveforms is on the order of $\leq 5\,\mathrm{s}$.

- *Repeatability*: In a typical experiment, the same sequence will be played many times, once per shot. This sequence should be the *same* every single time, and not

[1] At full output amplitude, we produce Rabi flops between $|0\rangle$ & $|1\rangle$ with a 2π-period of $\approx 600\,\mathrm{ns}$. Superconducting qubits typically see pi periods of $\approx 20\,\mathrm{ns}$ [143].

require re-generating the output sequence every single shot.

- *Waveform Feedback*: The waveforms that are actually output by the RF subsystem will need to vary over time. These reasons include but are not limited to crosstalk, laser frequency drift (technically repetition rate of the laser [144], which we correct in a process that we call "frequency feedforward"), amplitude (due to the chain of trapped ions heating up over time [120]), and other potential effects that have not been fully characterized.

- *Clear Programming Interface*: While not completely necessary, having a clear, well-documented, and easy-to-use programming interface for controlling the RF output & waveforms is highly desirable for clarifying exactly what is being output at what time by the RF subsystem.

- *Conditional Support*: Interesting quantum algorithms that are being considered today typically require quantum measurements in the middle of a circuit, which should then influence the rest of the circuit [43, 39]. To demonstrate these algorithms, the experimental setup must have both software and hardware support for inserting conditional logic into a previously-linear sequence.

As with any engineering problem, several of these requirements tend to conflict with each other. It is difficult to have an extremely long pulse, yet very fine-grained control over it.

There are different hardware implementations of a waveform generator, each of which has different strengths and weaknesses. These are, in order of increasing complexity:

Waveform Generator	Advantages	Disadvantages
Static RF	Stable, good phase coherence, infinite duration	No real-time control, modulation is difficult.
DDS	Stable, good phase coherence, infinite duration	Slow updates ($\approx 200\,\mathrm{ns}$)
AWG	Precise control	Limited duration sequences ($\approx 10\,\mathrm{ms}$) at high timing resolution, long upload time, limited conditional sequencing
Custom	Meet exact needs	Development time, complexity

Table 5.1: **Comparison of different waveform generation hardware implementations.**

- Static RF Source
- Direct Digital Synthesizer (DDS)
- Arbitrary Waveform Generator (AWG)
- Custom Solution

Balancing these trade-offs, as described in Table 5.1, led us to select a custom waveform generator for our coherent waveform control.

5.3 RFSoC System Description

To implement the custom waveform generator that became the core of our coherent RF control system, we first had to settle on a hardware platform. We selected the Xilinx RFSoC FPGA platform as the basis for this system.

In collaboration with a team at Sandia National Laboratory (SNL), we defined the requirements for a custom waveform generation solution. They then implemented the necessary embedded software (FPGA logic & CPU application code), which they call

Octet. Octet generates waveforms that are output by a combination of the RFSoC's FPGA logic & the RF DACs. For simplicity, we will refer to the software that runs on the embedded RFSoC CPU as "firmware," and the software that describes the RFSoC FPGA configuration as "gateware." [2]

5.4 RFSoC Physical Hardware

There can be many different physical sets of hardware that implement *Octet*, because Octet is a configuration of an FPGA chip[3], specifically the Xilinx RFSoC. This FPGA chip can be mounted in a variety of enclosures or form factors, just as a CPU can be packaged in laptops from a variety of vendors. We have demonstrated Octet working on both Xilinx evaluation boards (EK-U1-ZCU111-G, henceforth shortened to ZCU111) & Pentek 5950 platforms. Table 5.2 contains a summary of the differences between the two platforms. [4] Ultimately, the primary differences between the two platforms are price (ZCU111 is cheaper) & better support for multi-board output synchronization (Pentek offers a better solution).

Let us consider the major features (Section 5.5), and then break down the RFSoC capabilities as a set of inputs and outputs (Section 5.6).

[2] The Sandia Octet software is provided under limited access licenses to our group, under a Government Use License. To comply with licensing and export control restrictions, the EURIQA team only has access to pre-compiled binaries and does not have access to the source code for the Octet software. Thus much of this is a reconstruction based on testing, informal discussions, public APIs, and public papers [145]. For this same reason, we are not able to correct any software bugs that we might encounter.

[3] In the FPGA world, this is usually called an IP (Intellectual Property) block.

[4] It is important to note that the Octet gateware will work on any RFSoC sharing the same FPGA Integrated Circuit (IC) chip, but there must be a translation layer defined for it work properly in a given system. This translation layer is effectively responsible for standardizing the peripherals (clock, input/output pins, etc) that are available on the platform and exposing the required ones to Octet. For example, the Xilinx ZCU111 has two FMC connectors available on the evaluation board, but the more-compact Pentek 5950 does not expand all of these pins out. So a mapping is needed from the input signals that the Octet gateware expects to the available I/O pins available on e.g. the Pentek 5950 platform.

System	Pro	Con
Xilinx ZCU111	Simple, cheap, fully supported by Octet	Needs enclosure designed & built, requires $12.8\,\mathrm{MHz}$ stable clock source
Pentek 5950	COTS, corporate support offered, small form factor, phase synchronization across 8-channel RFSoCs	Cost, noisy chassis fans, lead times

Table 5.2: **Comparison of selected Xilinx RFSoC Platforms.** The Pentek 5950 & the Xilinx ZCU111 were selected as our primary options for system development. They are very different systems aimed at very different markets. The choice between these two is primarily between cost and support. The Xilinx ZCU111 is better supported by the Octet developers at Sandia National Laboratories, but the evaluation board itself is very bare-bones and is not meant for direct incorporation into finished systems. As such, it requires users to add on many features required to function in a lab, such as an external clock source at a precise frequency, and an enclosure to prevent accidental issues, such as shorts, with the sensitive electronics. Thus, a significant amount of effort is required to design, procure, construct, and test a full system that supports the ZCU111. The Pentek 5950 system, which is designed for commercial and military usage, has solved all the previously-stated deficiencies with the ZCU111. This convenience comes at a monetary cost, as a roughly equivalent Pentek system will cost $\approx 3\times$ as much as an equivalent 32-channel ZCU111 system. One major feature that the Pentek system supplies is the ability to synchronize the clocks of different RFSoC FPGAs to approximately the picosecond level.

Item Name	Description	Quantity
Xilinx ZCU111 RFSoC FPGA	Waveform generator	1
AVNet AES-LPA-502-G	RF Differential Breakout. Route RF signals from the RFSoC high-density connectors to SMA connectors	1
Minicircuits RF Baluns	Convert balanced (double-ended) RF inputs/outputs to single-ended at $50\,\Omega$ impedance	9
TI DS90LV047-48AEVM	Convert CMOS DIO to/from single-ended (external) to differential (for the ZCU111)	1
Xilinx FMC XM105 Debug Board	Breakout FPGA DIO signals	1
Bulkhead Passthroughs (Various)	Move signals from the outside of an enclosure to the interior	>10

Table 5.3: **Summary of BOM for Custom ZCU111 Enclosure for use with Octet.** I custom-designed this enclosure and accompanying BOM for testing and demonstrating Octet software running on the ZCU111 FPGA. A 32-channel system will have 4 copies of the above enclosure (8 output channels per ZCU111 RFSoC). This enclosure allows integrating an RFSoC into a server rack. It also allows easily expanding the number of RFSoC's in use as needs dictate.

5.5 RFSoC Feature Breakdown

The role of the RFSoC system is to generate waveforms in real-time to drive quantum gates on a set of qubits. The RFSoC achieves this by using digital logic to generate sine wave-like waveforms. Each RFSoC output channel[5] using Octet supports two simultaneous output tones,[6] each of which have several parameters that can be modulated using cubic splines. These parameters are frequency, amplitude, phase, and frequency, which are summarized in Table 5.4. We will call these the "spline engines". If we consider a single RFSoC output pulse as a function of time, it can be described as (where dt is the duration of the current pulse): [7] [145].

$$out_{tone}(dt) = amplitude(dt) * \sin\left(frequency(dt) * 2\pi * dt + phase(dt) + frame\right) \tag{5.1}$$

$$out(dt) = \sum_{n=0}^{1} out_n(dt) \tag{5.2}$$

The RFSoC has two operating modes:

- *Static*: This mode will output constant tones (in amplitude, frequency, and phase) until it is disabled.

- *Pulsed*: This mode allows queueing up a sequence of pulses, which can arbitrarily modulate each of the output tone parameter spline engines. This is described in

[5] i.e. one SMA cable
[6] We define a *tone* as a single parameterizable sinusoid wave, as described in Eq. (5.1)
[7] Eq. (5.1) neglects some advanced features such as crosstalk, which will be described in Section 5.5.3

Parameter Name	Description	Range	Resolution
Frequency	Tone's Output Frequency	-409.6 to $409.6\,\mathrm{MHz}$	$745.058\,\mu\mathrm{Hz}$ (40 bit)
Amplitude	Tone's Amplitude Envelope (Arbitrary units)	-1.0 to 1.0 (arb.)	6.1035×10^{-5} (16 bit)
Phase	Tone's Output Phase	0 to 2π rad	$5.7145 \times 10^{-12}\,\mathrm{rad}(40\,\mathrm{bit})$
Frame Rotation	Accumulated Frame Rotation ("R_Z"). Continuously accumulated until explicitly cleared.	0 to 2π rad	$5.7145 \times 10^{-12}\,\mathrm{rad}\ (40\,\mathrm{bit})$

Table 5.4: **RFSoC Pulse Parameters.** These are the parameters that can be controlled on a given RFSoC output tone for a given channel. Note that the frame rotation is shared between both tones of a single output channel, and can be applied to either/both of them simultaneously. See [145] for more details.

detail in Section 5.5.2.

In either operating mode, the RFSoC will receive commands over Ethernet, specifically via the Go Remote Procedure Call (GRPC) protocol. Each command is processed by a server running on the RFSoC's ARM CPU (called `rfcontrold`). `rfcontrold` will then send the appropriate control signals into the FPGA *fabric*, e.g. setting a register value, which will then affect the current or future output of that RFSoC.

A standard RFSoC has 8 RF output channels. For a system like the EURIQA Breadboard, which has individual control over up to 32 ions via a 32-channel AOM, there must be 4 parallel RFSoCs to drive all ions individually. [8] Fortunately, the RFSoC's can easily be connected and programmed in parallel. Each RFSoC requires:

- *Ethernet Connection* for receiving commands.
- *Synchronized clock*: this ensures that all RFSoC boards will output the same frequency without any system-specific drifts. [9]
- *Trigger*: to command the RFSoC to begin outputting the desired pulse sequence.
- *Feedforward signal (optional)*: This is described in Section 5.5.1. Depending on specific implementations the feedforward signal might only be routed to a single board or to every board, depending on whether feedforward is applied to the global Raman beam or the individual Raman beams.

[8] 4 RFSoCs can only drive 31 (not 32) ions, because one of the 32 channels is reserved for a globally-addressing channel. This could potentially be mitigated with a fifth RFSoC if desired.

[9] For ZCU111 boards running Octet, this must be a 12.8 MHz clock, which should be referenced to a common experiment clock source such as a 10 MHz Rb clock. Pentek systems include a clock distribution mechanism, the Pentek 5903, which takes a 10 MHz to 100 MHz input clock and generates the required RFSoC clock frequencies.

5.5.1 RFSoC Frequency Feedforward

The $355\,\mathrm{nm}$ Raman qubit laser that drives our qubit transition is known to exhibit noise in its pulse repetition rate [144]. If this noise was allowed to propagate through the system, it would lead to qubit decoherence. Conceptually, for short timescales $\approx 1\,\mathrm{s}$ the ion's transition frequency will remain stable. However, remember that the qubit transition is excited by a combination of a global laser beam, an individual laser beam, and the $RFdrive \rightarrow Amplifier \rightarrow AOM$ signal chain that modules each of these beams. Thus, a change in frequency on both the individual and global laser beam[10] can be corrected by modulating the frequency of the RF drive for either the global or individual beam in the opposite direction [113, Section 2.3.1]. [11]

In our system, where we are primarily performing phase-sensitive gates (as opposed to phase-insensitive [146]), we chose to apply the feedforward signal to the global Raman beam RF signal, and not the individual Raman RF signals. This has the side benefit of only requiring the feedforward signal to be input into a single RFSoC which drives the global beam RF signal, instead of every single RFSoC in the system.

This feature works by inputting an RF signal into an input channel of an RFSoC which is running the Octet software. The RF signal should be the result of mixing a constant RF signal with a signal that represents the error in the Raman beam repetition rate, so $f_{feedforward\ error@IF} = f_{IF} + f_{feedforward\ error}$ (where IF is an arbitrary intermediate fre-

[10] The laser frequency should shift symmetrically because both the individual and global beam are different paths for the same beam, which is split upstream of the AOMs.

[11] This is technically a frequency feedforward system, because the feedback loop is not closed by measuring the final output frequency. However, the Octet code does not specify the source of the error signal, thus it calls it a "frequency feedback" feature. We will elect to use "frequency feedforward" for clarity.

quency that users select). We use $140\,\mathrm{MHz}$ for our IF. The RFSoCs include an RF ADC, which is used to sample the incoming signal $f_{feedforward\ error@IF}$. This signal is then mixed against internal oscillators operating at the IF to generate an error signal, which drives a PID (Proportional-Integral-Derivative) lock. This lock tracks the frequency error from the IF, and then adds the error frequency to the output frequency for the RF output tone(s).

5.5.2 RFSoC Pulse Control

Executing gates on the EURIQA Breadboard quantum computer requires arbitrary and repeatable control of RF waveforms. Many quantum gate methods use amplitude, frequency, or phase modulation (AM, FM, PM), or some simultaneous combination of these [147, 148, 149]. Thus, static waveform signals, or ones that cannot be modulated at the desired rate, are impractical for the EURIQA Breadboard.

To meet this need, the Octet RFSoC gateware includes cubic spline engines for each of the amplitude, frequency, and phase parameters, each of which can be programmed independently with a duration and envelope (time-varying value). An arbitrary waveform can then (in most cases) be described as a combination of these parameters changing over time. The cubic spline engines, based on [150], will each output a signal:

$$out(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 \tag{5.3}$$

where c_n are the n-th order spline coefficients of the parameter. We will use `CubicSpline(`

$c_0 = c_0, c_1 = c_1, c_2 = c_2, c_3 = c_3$) to denote future cubic spline coefficients. [12]

Let us consider a simple triangle wave amplitude waveform sequence on a single tone, played for a total of $20\,\mu s$, the amplitude component of which is illustrated in Fig. 5.1. A triangle wave is simply a positively-sloped ramp followed by a negative slope. Considering only the amplitude spline, this waveform then looks like the following commands to the amplitude spline engine:

1. *Ramp up*: For $10\,\mu s$, increase from $0.0 \rightarrow 1.0$ (arbitrary units). The equivalent cubic spline is `CubicSpline`$(c_0 = 0.0, c_1 = 1.0, c_2 = 0.0, c_3 = 0.0)$.

2. *Ramp down*: For $10\,\mu s$, decrease from $1.0 \rightarrow 0.0$, which is equivalent to `CubicSpline`$($ $c_0 = 1.0, c_1 = -1.0, c_2 = 0.0, c_3 = 0.0)$.

Knowing that each cubic spline command to the RFSoC is represented with $256\,\mathrm{bit}$ ($32\,\mathrm{B}$), this entire amplitude ramp sequence can be described in only $512\,\mathrm{bit}$. In this formalism, because the duration of the pulse is not directly a part of the cubic spline, to extend this waveform to $10\,\mathrm{s}$ would require the same number of command bits, as the same spline word can be played for a varying duration simply by changing the duration field of the command word. [13]

[12] The actual RFSoC hardware uses PDQ Splines, instead of native cubic splines. PDQ splines [151, 150] are nearly equivalent to cubic splines, but they are simpler to compute using integer-only arithmetic on embedded devices such as an FPGA, instead of needing to perform a slow floating-point multiplication step for each time unit.

[13] This is very different than AWG-based RF waveform generators. Most AWG programming sequences require specifying the exact output amplitude at each time point (sample or Sa). This means that a waveform sequence will require storage $n_{bits} = resolution(\frac{\mathrm{bit}}{\mathrm{Sa}}) * duration(\mathrm{Sa})$. Assuming an AWG with $1\,\frac{\mathrm{GSa}}{\mathrm{s}}$ output (which outputs one sample per nanosecond), and $16\,\mathrm{bit}$ resolution ($16\,\mathrm{bit\,Sa^{-1}}$), $1\,\mathrm{s}$ of output will require $2\,\mathrm{GB}$ of data. This quantity will scale with the duration of the waveform, in contrast with the RFSoC-based approach using Octet. $2\,\mathrm{GB}$ for $1\,\mathrm{s}$ of output is significantly larger than the $64\,\mathrm{B}$ that an RFSoC running Octet requires. In reality, because amplitude, frequency, phase, and frame rotation will typically be specified for both tones, and for each channel on the RFSoC, the example triangle waveform might actually require $512\,\mathrm{B}$ on the Octet. However, this is still much more efficient encoding than an AWG: if the triangle

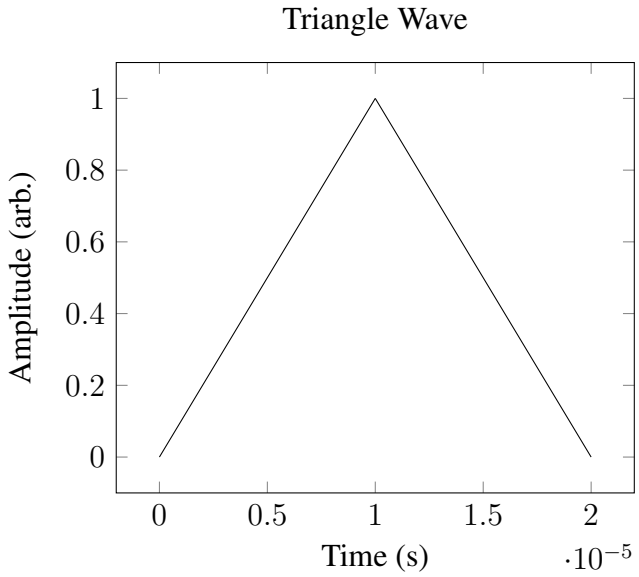


Figure 5.1: **Example Triangle Wave Amplitude.** The ramp-up can be described by $A(t) = t$, or $\texttt{CubicSpline}(c_0 = 0.0, c_1 = 1.0, c_2 = 0.0, c_3 = 0.0)$. The ramp-down can be described by $A(t) = 1 - t$ (where $t = 0$ is referenced to the start of the ramp-down pulse), which is equivalent to $\texttt{CubicSpline}(c_0 = 1.0, c_1 = -1.0, c_2 = 0.0, c_3 = 0.0)$. Similar pulses, comprised of up to cubic (3rd-order) polynomials, could be equally well-represented by a pair of CubicSplines.

Given the capability of an RFSoC to output near-arbitrary waveforms, generating the waveforms now becomes a problem of filling the spline engines with the appropriate data. In this scheme, a user will send a data stream corresponding to the waveforms to the Octet *rfcontrold*, which will transfer that data into the FPGA's memory using DMA (Direct Memory Access). This sequential buffer is then routed to the appropriate per-channel spline engines, and then output in sequence. There are two ways of programming a waveform sequence: *streaming* mode (Section 5.5.2.2) or *gate* mode (Section 5.5.2.3).

waveform lasts for more than $512\,\mathrm{B} * 8\,\frac{\mathrm{bit}}{\mathrm{B}} / 16\,\frac{\mathrm{bit}}{\mathrm{Sa}} = 256\,\mathrm{Sa} (\equiv 256\,\mathrm{ns} on an AWG operating at 1\,\mathrm{GSa\,s^{-1}})$, the RFSoC encoding will be more space-efficient.

5.5.2.1 Cubic Spline Word Format

Before considering how data words are processed in either streaming or gate mode, it is important to consider the data word format. One straightforward way of considering the RFSoC gateware is as a simplified RISC (Reduced Instruction Set Computer) processor. [14] Each RFSoC programming word is the same width ($256\,\mathrm{bit}$ or $32\,\mathrm{B}$). The programming sequence is read by the RFSoC in sequential order, so it should be time-ordered. As each word is read, it is either:

1. *Distributed to spline engine*
2. *Programmed to look-up table (LUT)*
3. *Retrieves words from LUT*

The first method (distribution) is the streaming mode, and the second and third methods are the gate mode.

In addition to its processing method, the words also contain metadata that indicates *how* the data word should be processed, typically by the spline engine. These include flags such as waiting for a trigger before output, whether the output should be toggled on/off, if frequency feedback should be toggled on/off, etc. See [145] for more metadata details.

[14] The primary simplifications are in lack of conditional sequences, arithmetic logic, and registers, but the general concept of fixed-size instruction words that are sequentially processed, decoded, and acted upon still holds.

5.5.2.2 Streaming Mode

Streaming mode is conceptually very simple. Given a queue of spline engine programming words, a simple sequencer will process each item in the queue by distributing it to a buffer for the appropriate spline engine. Each spline engine will process the words that have been distributed to it in the order that they have been received, and begin the next item in its buffer once the previous spline's duration has elapsed. To ensure that the outputs of all spline engines are aligned correctly in time (i.e. that there is no skew between different channels), it is recommended to begin all sequences with a "wait-for-trigger" word for each spline engine, which will delay all outputs until it receives a trigger signal. [15]

However, our testing has shown that streaming mode is not always reliable for long sequences of pulses. Specifically, for certain pulse sequences a channel will stop outputting partway through the sequence, and never resume. We have not been able to debug some of these issues due to license restrictions, not having the source code, and alternative options like the LUT programming mode being available. [16]

To program the RFSoC with a streaming data sequence, you must first generate the programming words for each spline engine. Then those programming words must be sorted in time across all spline engines. This ensures that all spline engines will be filled with the appropriate data for the entire sequence, and will not starve. [17] For example,

[15] This trigger can actually equally be either a hardware input line or a software signal. Because the RFSoC is a digital device, it is simple to OR two inputs (i.e. hardware trigger & software trigger), and use the OR'd signal as the trigger input.

[16] Streaming mode is also relatively low-priority for the Sandia developers, so this might be a dead-end approach. Our general belief is that there is some issue with DMA transfers, but we have not been able to investigate further.

[17] i.e. be empty of data

consider a situation where the entire sequence for a single spline engine (*SE1*) was sent sequentially, and then the second spline engine (*SE2*) was programmed. If *SE1*'s buffer overflowed before any data was sent to *SE2*'s buffer, then the gateware responsible for distributing the input data would need to select between dropping the remainder of the data destined for *SE1* (to allow *SE2* to receive any data), or waiting until *SE1*'s buffer emptied enough to continue sending it data, which would cause *SE2*'s output to be delayed behind *SE1* in time. In reality, the second situation is what has been implemented on Octet.

Calculating the number of data bits to represent a sequence in streaming mode is very straightforward: it is the sum of the number of `CubicSpline` words that you use times the size of each programming word ($256\,\mathrm{bit}$). If we make the same simplification assumptions that a `ToneData` (see Chapter 6) does (a single tone's frequency/amplitude/phase is modulated simultaneously for the same duration, thus all 4 spline engines of a tone are executed simultaneously for the same duration), then this reduces to:

$$N_{words} = \sum_{t}^{N_{tones}} N_{ToneData_t} * 4 \tag{5.4}$$

where N_{tones} is the number of tones for a given RFSoC board, and $N_{ToneData_t}$ is the number of `ToneData` to be played on a given channel's tone for a waveform sequence. For a sequence where 1024 `ToneData` are played on each of 8 channels (16 tones) simultaneously, $N_{words} = 65\,536\,\mathrm{words} = 2\,\mathrm{MiB}$. Note that in streaming mode, if this sequence needs to repeat (e.g. for multiple shots), then this entire $2\,\mathrm{MiB}$ sequence will need repeated $\times N_{shots}$.

5.5.2.3 LUT Programming Mode

To solve these data distribution problems, as well as to reduce the amount of data that needs sent to the RFSoC even further, a solution was devised that essentially caches blocks of waveforms in the RFSoC memory. We call this method *LUT mode* (Look-Up Table) or *gate mode*, as opposed to the previously-described *streaming mode*.

To maximize potential reuse, the Sandia developers decided to implement hierarchical LUTs, **one per output channel** (i.e. two output tones for the same channel share a LUT). From lowest to highest-level: [18]

1. *Pulse LUT (PLUT)*: Contains spline engine programming words, one per address. Example entry: $\{addr_{PLUT} : (256\,\text{bit spline word})\}$

2. *Memory Map LUT (MLUT)*: A pointer to an entry in the PLUT. This allows the same pulse to be re-used in different sequences. Example entry: $\{addr_{MLUT} : addr_{PLUT}\}$.

3. *Gate LUT (GLUT)*: Range of MLUT entries that comprise a gate. Playing one "gate" will execute all MLUT entries in range $[addr_{lower} \rightarrow addr_{upper}]$ (inclusive of both ends). Example entry: $\{addr_{GLUT} : \left(addr_{MLUT_{lower}}, addr_{MLUT_{upper}}\right)\}$

4. *Gate Programming Data*: List of entries from the Gate LUT that should be executed on a given channel.

[18] Note: the names "gate" and "sequence/memory map" are figurative, and not binding. That is, one entry in the "gate" LUT can actually combine the output waveforms for several gates if desired, or a single sub-segment of a gate. This will be covered more in **??**.

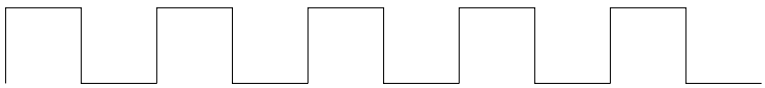


Figure 5.2: **Example of a single-channel square amplitude waveform.** This waveform sequence is used to illustrate an example in Eq. (5.5)

With this scheme, instead of specifying each individual pulse that should execute at any point in time, the desired waveform sequence can now be described as a sequence of gate entries to play on a given channel. Then the entire waveform programming sequence looks like:

1. *Program PLUT, MLUT, and GLUT* entries for channel $1 \ldots n_{channels}$.
2. *Execute gates* $1 \ldots n_{gates}$ on channel $1 \ldots n_{channels}$ (a single shot).
3. *Execute remaining shots*: repeat data from #2 n_{shots} times (total, inclusive of #2).

In this scheme, the gates are not required to be the same across all channels: the gate sequence for channel 1 might be $[0, 5, 3, 2]$, but for channel 2 it can be $[0, 2, 1, 5]$.

This scheme allows compressing the waveform sequence even smaller than that of the streaming mode (Section 5.5.2.2). In the case of a single square pulse repeated $N_{gates}\times$ (e.g. Fig. 5.2), the number of $256\,\mathrm{bit}$ words per unique pulse sequence will be: [19]

$$N_{words} = N_{PLUT\ entries} + \left\lceil \frac{N_{MLUT\ entries}}{9} \right\rceil + \left\lceil \frac{N_{GLUT\ entries}}{6} \right\rceil + \left(\left\lceil \frac{N_{gates}}{N_{gates\ per\ word}} \right\rceil * N_{shots} \right) \tag{5.5}$$

$$N_{words} = 4 + 1 + 1 + \left(\left\lceil \frac{N_{gates}}{24} \right\rceil * N_{shots} \right) \tag{5.6}$$

[19] This equation ignores a slight constant overhead due to preparation & end-of-sequence buffering pulses. These might be able to be removed with proper testing, but currently they add a per-channel overhead of $N_{PLUT\ entries} + 8, N_{MLUT\ entries} + 8, N_{GLUT\ entries} + 2, N_{gates} + 3$.

This equation assumes that data is only being written to a single output channel. If Eq. (5.5) is expanded to output to every channel then the equation becomes:

$$N_{words} = \sum_{ch=0}^{N_{channels}} \left(N_{PLUT\ entries} + \left\lceil \frac{N_{MLUT\ entries}}{9} \right\rceil + \left\lceil \frac{N_{GLUT\ entries}}{6} \right\rceil + \left(\left\lceil \frac{N_{gates_{ch}}}{24} \right\rceil * N_{shots} \right) \right) \tag{5.7}$$

If the example waveform from Fig. 5.2 is played on a single tone for each of eight channels, the equation becomes:

$$N_{PLUT\ entries} = 8 \tag{5.8}$$

$$N_{MLUT\ entries} = 8 \tag{5.9}$$

$$N_{GLUT\ entries} = 1 \tag{5.10}$$

$$N_{words} = \sum_{ch=1}^{8} \left(8 + \left\lceil \frac{8}{9} \right\rceil + \left\lceil \frac{1}{6} \right\rceil + \left(\left\lceil \frac{N_{gates_{ch}}}{24} \right\rceil * N_{shots} \right) \right) \tag{5.11}$$

Assuming 100 shots of this sequence, and 24 gates per sequence, $N_{words} = 880$ words $= 27.5$ KiB.

5.5.3 RFSoC Crosstalk

One common source of error in quantum computers is crosstalk between addressed qubits. Crosstalk can arise from several sources, whether optical, electrical, or acoustic [152, 153, 154, 155].

If the crosstalk components are relatively stable, it is theoretically possible to can-

cel this crosstalk by applying inverse tones to the offending channel(s). For clarity, we will call the channel whose output is intended the "drive" channel, and the channels that experience crosstalk will be the "crosstalk" channels.

Consider a case that we have experienced: we observed crosstalk at a desired ion from neighboring ions. We determined that the crosstalk primarily originated in the individual AOM, as acoustical waves that produced the AOM effect would also modulate the neighboring channels. This effect is approximately at the $3\,\%$ level in terms of Rabi rate[20] between neighboring ions in a chain[21]. We then demonstrated that by applying cancellation tones of a calibrated amplitude and phase, we could effectively nullify this crosstalk effect. The cancellation tones would use the calculated output on the desired channel, say Channel 5. We then observed and calibrated the crosstalk amplitude and phase on Channels 4 & 6. To cancel the crosstalk, we would then calculate the sine waves for the desired channel, and then apply a phase and amplitude adjustment to that output before adding it to the output of Channels 4 & 6. In Eq. (5.12) we express this crosstalk application mathematically, where we let:

- N be the offset of a neighboring channel from the drive channel, for $N \in \{-2, -1, 1, 2\}$.
- $W_{drive}, W_{drive+N}$ are respectively the output waveforms (sine waves) on the drive channel, and the neighboring channels offset by N, after all crosstalk correction has been applied.

[20] Rabi rate is effectively the frequency at which a Rabi flop is executed on an ion. A Rabi rate of $1\,\mathrm{MHz}$ means that one Rabi flop of $2\pi\,\mathrm{rad}$ will be executed every $1\,\mu\mathrm{s}$. Thus a crosstalk amplitude of $1\,\%$ in Rabi rate will drive a Rabi flop on the unintended ion that is $100\times$ slower than the Rabi rate intended ion. For a $1\,\mathrm{MHz}$ drive Rabi rate, that means the crosstalk will have a Rabi rate of $10\,\mathrm{kHz}$.

[21] We were later able to determine that the majority of this crosstalk was due to a slight optical misalignment into the AOM cell, which exacerbated this crosstalk effect.

- $\sum o_{drive}(\phi), \sum o_{drive+N}(\phi_{offset})$ are the outputs of all tones of a drive channel (see Eq. (6.1)), with a specified phase offset ϕ_{offset}.

- $A_{xtalk,N}, \phi_{xtalk,N}$ are the amplitude multiplier and phase offset that will be applied to a given channel.

$$W_{drive} = \sum o_{drive}(\phi_{offset} = 0) \tag{5.12}$$

$$W_{drive+N} = \sum o_{drive+N}(\phi_{offset} = 0) + A_{xtalk,N} * \left(\sum o_{drive}(\phi_{offset} = \phi_{xtalk,N})\right) \tag{5.13}$$

Note that crosstalk correction values are actually a matrix, in the above example crosstalk was simplified into a single row of the matrix. The elements of the crosstalk matrix are defined as $C_{i,j}$ equals the crosstalk from ion i onto ion j. If we consider only crosstalk amplitude for a 4-ion linear chain, with only nearest-neighbor crosstalk, an example crosstalk matrix could look like:

$$C_A = \begin{bmatrix} 0 & 0.1 & 0 & 0 \\ 0.01 & 0 & 0.01 & 0 \\ 0 & 0.01 & 0 & 0.01 \\ 0 & 0 & 0.01 & 0 \end{bmatrix} \tag{5.14}$$

Note that the diagonal has no crosstalk, because that is the desired effect of driving that ion.

On a traditional AWG-based solution, crosstalk cancellation is expensive to apply

because it requires pre-computing all intended waveforms, and then also computing the crosstalk cancellation waveforms to apply. The RFSoC running Octet neatly bypasses this computation problem by calculating crosstalk cancellation in real-time. Essentially, it routes the output of an intended drive channel's spline engines to its neighboring channels. Each of these channels then applies the specified crosstalk amplitude multiplication and phase offset to obtain the final output signal for that channel.

5.5.4 RFSoC Single-channel Frequency Synchronization

The RFSoC offers two waveform output modes of frequency output: *global synchronous* and *contiguous* (essentially whether global synchronization is applied or not). This concept is fully explained in [145, Section V], and illustrated in [145, Fig.2] and [156, Fig. 24].

In short, *global synchronous* can be thought of as referencing the phase to a per-RFSoC board frequency reference (counter), which is initialized at board start-up time. This frequency reference is essentially unique *per-frequency*, which means that a slight frequency error (even due to rounding errors) can cause a large phase discontinuity when using this mode.

The (default) alternative, *contiguous*, is that the output waveform will remain continuous over frequency changes, as illustrated in [145, Fig. 2].

5.5.5 RFSoC Real-Time Pulse Feedback

One critical feature that the RFSoC supports, enabling logical qubit error correction [39] and other interactive algorithmic uses, is the ability to select which gates will be executed (which waveforms will be output) in real time.

This feature works by implementing an "ancilla" (aka conditional) register, which can be populated either by GRPC commands or real-time hardware signals. To explain the operation of this feature, let us consider a simple sequence:

1. *Play unconditional* pulse on 4 channels simultaneously.
2. *Play conditional* pulse on selected channel of the 4.

Executing this sequence then requires the following steps:

1. *Generate data words* for each of the pulses (both conditional and unconditional).
2. *Assign Look Up Table Addresses* (see Section 5.5.2.3 for introduction). This is modified by assigning each "branch" to a different section of the Gate LUTs, based on the address prefix. For example, with a single-bit conditional address $0..._2$ implies the `False` condition, and address $1..._2$ implies the `True` condition. At execution time, if it is a conditional "gate", then the condition is pre-pended to the GLUT address, thus selecting the appropriate conditional gate.
3. *Send programming data* to the RFSoC.
4. *Trigger RFSoC* to begin sequence.
5. *RFSoC outputs unconditional sequence*.

6. *RFSoC waits for conditional choice*.

7. *Send selected condition* to the RFSoC: either via GPIO pins, SPI, or software.

8. *Trigger RFSoC* to begin conditional sequence.

9. *RFSoC outputs conditional sequence*.

10. (Optional): either *continue* with unconditional sequence, or *repeat* #6 through #8 for another conditional execution.

The limitation of this model is that it only supports linear circuit execution, and not fully arbitrary branching sequences. That is, the same condition can be re-executed multiple times, but only in a pre-defined sequence. So the following is possible:

```
1     conditional_gate_0
2     conditional_gate_1
3     conditional_gate_0
```

But this sequence is not:

```
1     while condition; do
2         conditional_gate_0
3     done
4
5     conditional_gate_1
```

Fundamentally, this is because there is no branching instruction for the control sequence. The conditional statement only applies to the output waveforms (i.e. selecting a set of PLUT entries to play), and cannot be used to choose which gate sequence to execute.

5.6 RFSoC Inputs & Outputs

5.6.1 RFSoC Inputs

The RFSoC has two types of inputs: real-time and setup. The real-time inputs are used for controlling a pulse sequence *during* a quantum operation (or to start the quantum operations), and the setup inputs are to either initialize the RFSoC board or communicate the pulse sequence to the RFSoC.

For real-time inputs, the RFSoC has:

- *Trigger*: begin outputting a sequence. This is a digital input signal.
- *"Ancilla" Control*: Choose which conditional branch to execute (see Section 5.5.5).
- *Feedforward Signal*: an RF ADC channel receives the RF error signal, as described in Section 5.5.1.

The setup inputs are configuration values, which are primarily conveyed over GRPC on a LAN (Local Area Network), from either a GUI program or a script. These include specifying which channels and spline engines should be enabled, crosstalk values, etc.

5.6.2 RFSoC Outputs

Every RFSoC board supports $8\times$ channels of RF DAC output, in addition to any digital outputs that can normally be output from an FPGA. The Sandia RFSoC gateware supports two digital status update signals: output busy (i.e. some signal is currently being output on the channels), and FIFO empty (i.e. there is no data currently queued into the spline

engines).

These signals can be used as part of a control loop, such as causing the ARTIQ control device to wait until the RFSoC has finished outputting to begin the next section of the experimental sequence.

5.7 Conclusion

In summary, an RFSoC-based RF waveform system adds the capability for performing conditional RF sequences, which allows for limited logical qubit error correction capability. In addition, it adds the major quality-of-life improvement of improving the pre-experiment waveform generation time by an order of magnitude or two, both due to inherent design and optimized waveform generation code as will be described in Chapter 6. We have successfully demonstrated this RFSoC-based system on ions, as shown in Chapter 7, and plan to use it for future experiments.

Chapter 6: PulseCompiler Waveform Synthesis & Specification

6.1 PulseCompiler

Currently, there are no existing off-the-shelf solutions that provide a high-level software interface for specifying arbitrary waveforms, while meeting all the requirements of a quantum computing laboratory. Furthermore, there are no existing COTS solutions that can compile to our RFSoC hardware described in Chapter 5. The goal of such a package would be to specify near-arbitrary waveforms, which can modulate native attributes of a RF signal such as frequency, amplitude, and phase, and then output data in a format understandable by the RFSoC. Conceptually, this system might look like a high-level ISA (Instruction Set Architecture), where users can specify a "program" to execute that changes waveform attributes in real time.

Existing solutions are either too hardware-specific, such as controlling a specific DDS IC, or not quite general enough. The closest existing solution that we found to our needs is OpenPulse from IBM's quantum computing group [26], but that still does not meet all of the needs of our quantum computing lab. For instance, OpenPulse has the following advantages and disadvantages:

- Advantages:

  - Open Source, so can be modified or investigated to resolve issues.
  - Well-documented, which reduces developer workload for producing documentation on how to use a custom solution.
  - Built-in conversions from quantum circuits to waveforms. This reduces the scope of the required work to a translation layer from OpenPulse to running on the RFSoC system.
  - Part of a common quantum computing package, to minimize the learning curve for working with the OpenPulse package (and its enclosing package `qiskit-terra`).

- Disadvantages:
  - Supports continuous amplitude modulation, but only discrete frequency or phase modulation (so continuous FM/PM gates are not natively possible).
  - Unclear or non-existent systems for outputting multiple RF tones to a single output channel/qubit. In other words, they assume one frequency per output channel, which is not compatible with standard trapped ion gates that we use on the EURIQA Breadboard.
  - Does not support arbitrary control sequences, such as branches or conditional pulses.

Ultimately, we found that OpenPulse was very close to our needs, and any shortcomings could be circumvented with relatively small effort. As such, we decided to build extensions to the OpenPulse concept, allowing it to support the full power of the

SNL Octet RFSoC software while also integrating with pre-existing quantum circuit-processing software developed by our laboratory.

This chapter will describe the `PulseCompiler` (also represented as `pulsecompiler`) package and its integration with RFSoC software. On some levels, `PulseCompiler` is a re-implementation of Sandia National Lab's GateCompiler/JAQAL interface [145] for programming an RFSoC running the SNL Octet firmware/gateware. On other levels, it is a translation from (and extension of) OpenPulse to waveforms that run on RFSoC hardware.

6.1.1 PulseCompiler Overview

At a high level, this package provides a method for declaring RF sine waves that will be played on the RFSoC, and then converting those sine waves to the binary programming data for the Octet firmware/gateware. These sine waves can then be used to construct quantum gates to perform desired quantum operations.

6.1.2 Implementation

The basic class structure of `PulseCompiler` can be seen in Fig. 6.1. One key component is `ToneData`, which represents the output of a single tone (i.e. specific frequency, amplitude, phase) on an output channel for a duration, as well as the accompanying metadata/flags. A tone's output [1] is described as a list of `ToneData`. To encapsulate a set of outputs across all tones & channels, we use a `ChannelSequence`, which is a dictionary between a channel representation (whether in Qiskit format e.g.

[1] This can also be thought of as the output of a DDS IP core.

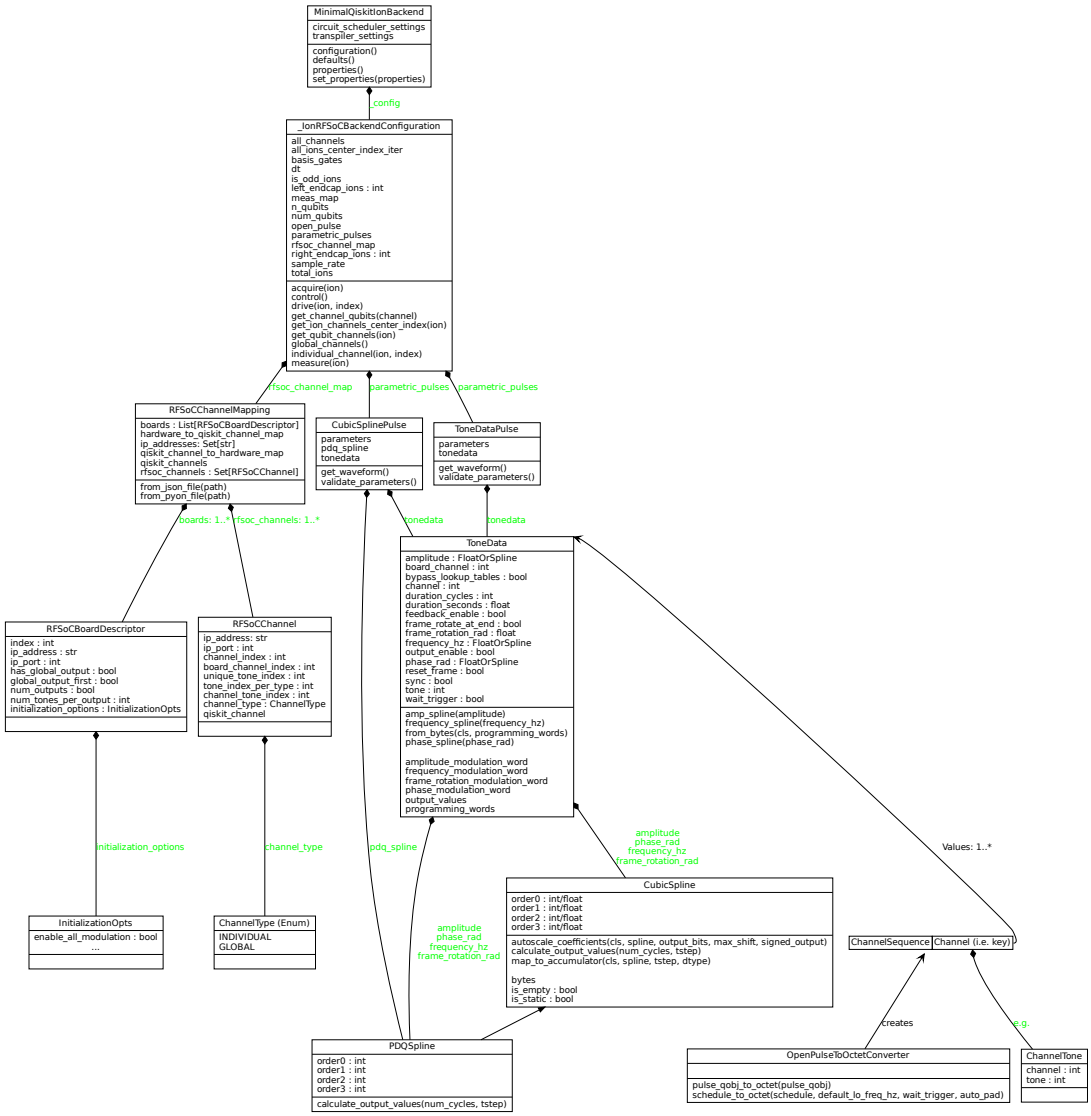


Figure 6.1: **Primary `pulsecompiler` classes, as a UML class diagram.** Higher-level classes are optional, and are only relevant if you are using `qiskit` for describing pulses. The base classes are at the bottom, and more abstract ones are towards the top. Green text w/ an arrow denotes a class attribute is an instance of the other class, most other arrows are inheritance.

`qiskit.pulse.DriveChannel` or a tuple pair of $(channel, tone)$ called `ChannelTone`) and a `List[ToneData]`. It is the user's responsibility to ensure that all tones end at the same time and have the same number of triggers (this can be handled automatically if using the built-in conversion class `OpenPulseToOctetConverter`).

The default Octet gateware provides for ≤ 2 **independent** tones on a single RF output channel. These are treated independently through most of the gateware (see Fig. 6.2), until they are summed just before being output, so `pulsecompiler` does the same.

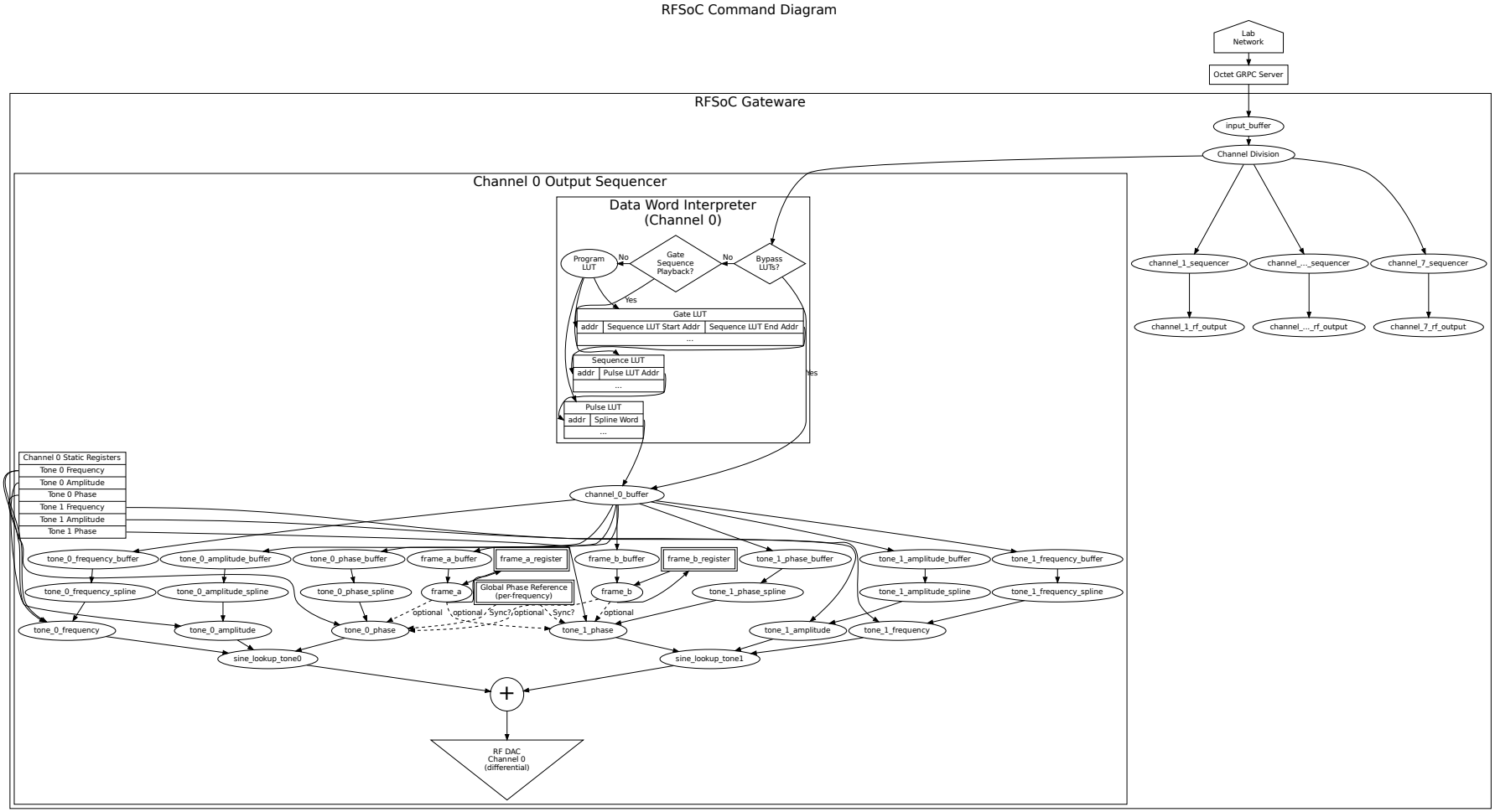


Figure 6.2: **Graph of how a data word is transferred to and interpreted by the Octet RFSoC gateware.** Note that there are a total of 64 spline engines, 8 parameters for 8 channels (not all shown). This diagram is based on my best understanding, and might not be 100% accurate without access to the binary source code.

The recommended method of declaring your pulse sequence is creating a `qiskit.pulse.Schedule`, then converting that to `ChannelSequence` with `pulsecompiler.qiskit.schedule_converter.OpenPulseToOctetConverter.schedule_to_octet()`.

6.1.3 Uploading to A RFSoC

The channel mapping (i.e. which output channel that a tone should play on) of a `ChannelSequence` can remain unspecified until upload time. This allows reusing the same schedule across different devices, or using abstract channel representations (e.g. `qiskit.pulse.DriveChannel`) across multiple boards.

However, just before a sequence is run, when uploading the desired pulse sequence, the `pulsecompiler` must know which exact RFSoC board and channel that the `List[ToneData]` should play on. Thus, `pulsecompiler` expects a mapping between abstract channel representation and a `RFSoCChannel`. If you are using `pulsecompiler` in conjunction with `qiskit`, this is handled with `RFSoCChannelMapping`, which provides a default mapping from `Union[DriveChannel, ControlChannel] -> RFSoCChannel`. It generates this using a configuration file describing the RFSoC boards, their capabilities, and their IP addresses.

If the `flush` flag is set, then any previously-queued sequence in RFSoC memory will be cleared, otherwise the new sequence will be appended to the sequence already in-memory.

6.2 RFSoC Output Modulation

The RFSoC defines the output waveform for a particular tone as roughly

$$o_{tone}(dt) = amplitude(dt) * sin(frequency(dt) * 2\pi * dt + phase(dt) + frame) \quad (6.1)$$

Note that there are four modulation parameters: amplitude, frequency, phase, and frame rotation (i.e. RZ, or an accumulated phase carried over between pulses). Each of these modulation parameters is represented as a Cubic (third-order) spline, and is fed to the DDS core that generates the digital sine wave output. The spline outputs are calculated using accumulators, and the spline outputs are relative to the start of that parameter's command word (i.e. dt). In other words, you can continuously vary any of the frequency/phase/amplitude over the duration of an RFSoC output pulse. [2]

The spline parameters have two logical types: a real-valued type `CubicSpline`, and the type `PDQSpline` that is actually used on the RFSoC. They are functionally equivalent, but the component values are slightly different due to optimizing the values to be better accumulated in the hardware. [3] Both are accepted interchangeably throughout the `pulsecompiler`, though `CubicSpline` are easier to understand.

To understand how CubicSplines can be used in a `ToneData`, consider the following example of linearly ramping the amplitude of a sine wave from $0.0 \rightarrow 1.0$ (arbitrary

[2] Though not explicitly shown, the frame rotation can also be varied over time, but it makes more logical sense to consider a constant frame rotation applied at the beginning or end of the pulse, and have the time-modulation on the phase parameter.

[3] See Footnote 12 from Chapter 5 for more details on `PDQSpline` vs `CubicSpline`.

units) over 100 clock cycles: [4]

```
1    amp = CubicSpline(0, 1, 0, 0)
2    t = ToneData(0, 0, 100, 200e6, amp, 0.0)
```

6.3 OpenPulse Waveforms

PulseCompiler was designed to work seamlessly with Qiskit Terra's Pulse module (`qiskit.pulse`), aka OpenPulse. The goal of PulseCompiler is that users can define a `qiskit.pulse.Schedule`, which can be converted to an equivalent set of pulses for the RFSoC to play.

6.3.1 Why OpenPulse?

The primary reason that a translation layer from OpenPulse to the RFSoC was decided to be used was that it can take advantage of the pre-existing `qiskit` infrastructure investment both from IBM and the greater community. So it makes sense to reuse as much of their user-interface design as possible, and just interpret their data structures. OpenPulse has the following further advantages:

- Per-channel RF control.
- Understanding of timing at the clock cycle level.
- Visualization tools for Schedules.
- Data structures for Schedules, Pulses, and serialization of those data structures as JSON dicts.

[4] The other parameters of `ToneData` above represent, respectively: channel 0, tone 0, duration 100 clock cycles, frequency 200 MHz, phase 0.0 rad.

6.3.2 PulseCompiler vs OpenPulse Assumptions

IBM made some assumptions [157] about the hardware available for OpenPulse backends that do not exactly match those available on EURIQA/Ion Systems.

- Qiskit assumes all `Channels` are mixed with a frequency source.
- One `DriveChannel` per qubit. PulseCompiler allows multiple (i.e. one RFSoC output channel has two tones, each tone is mapped to one `DriveChannel`).
- A `ControlChannel` can drive multiple qubits. In PulseCompiler, the `ControlChannel` is assumed to touch ALL qubits (i.e. as a global beam via AOM/RF channel).
- `MeasurementChannel` and `AcquireChannel` are not supported via PulseCompiler, because they are not directly driven by RFSoC. The closest equivalent for these would probably be the readout DDS, which is currently controlled by ARTIQ.

6.3.3 PulseCompiler & OpenPulse Schedules

Qiskit supports some basic classes of pulse instructions that can be added to schedules. These consist of:

- *Phase*: set or shift (i.e. adjust upwards or downwards).
- *Frequency*: set or shift.
- *Pulses*: Amplitude control, for e.g. playing a Constant-amplitude sine wave.

Schedules are defined in terms of dt, which is the native timescale of the backend's waveform generator (i.e. the RFSoC), roughly its clock cycle. For the RFSoC, $dt =$

$\frac{1}{409.6\,\mathrm{MHz}} \approx 2.45\,\mathrm{ns}$. As an example schedule, the following will play a $200\,\mathrm{MHz}$ sine wave of constant amplitude at $\phi = 90°$ for 100 dt ($\approx 250\,\mathrm{ns}$):

```
import qiskit.pulse as qp

out_chan = qp.DriveChannel(0)
with qp.build() as sched:
    qp.set_frequency(200e6, out_chan)
    qp.set_phase(3.14/2, out_chan)
    qp.play(qp.Constant(100, 1.0), out_chan)
```

This schedule can then be stored (serialized) as a `qiskit PulseQObj`.[5] If you are using a typical AWG-based waveform generator, then storing waveforms as samples makes sense. However, storing waveform samples does not make sense for the RFSoC, as that is inefficient, redundant, and does not match the hardware capabilities. The "workaround" is that `pulsecompiler` for RFSoC only supports Qiskit `ParametricPulses` (which include pulses such as `Constant`, `Gaussian`, `GaussianSquare`), which PulseCompiler then synthesizes into the appropriate `ToneData`. In other words, `qiskit.pulse.Waveform` (formerly `SamplePulse`) is not supported in PulseCompiler.

The limitation of this is that OpenPulse by default only supports discrete phase & frequency modulation (but continuous amplitude modulation), while the RFSoC supports third-order modulation of frequency, phase, or amplitude. To circumvent this limitation, we added support for two additional ParametricPulses via PulseCompiler: `CubicSplinePulse`, and `ToneDataPulse`. `CubicSplinePulse` fits in the Qiskit standard of amplitude-only modulation, but provides third-order modulation of the amplitude to match RFSoC capabilities. `ToneDataPulse` is designed to exactly mirror `ToneData` and expose its capabilities to Qiskit schedules. So any control available in `ToneData`, is also available

[5] A `PulseQObj` essentially resolves any parameters/frequencies, converts the schedule to a set of instructions, and stores any waveforms associated with the schedule.

in `ToneDataPulse`.

Once a schedule is created, it can be converted to a `ChannelSequence` with `pulsecompiler.qiskit.schedule_converter.OpenPulseToOctetConverter.schedule_to_octet()`.

6.4 Converting Circuits to RFSoC Output

While the full scope of this functionality is slightly out of the scope of PulseCompiler (primarily because we leave the definition of custom gates up to the end-user), it is still instructive to understand what pieces need to be written in order to convert a `qiskit.circuit.QuantumCircuit` to a `ChannelSequence` ready for upload.

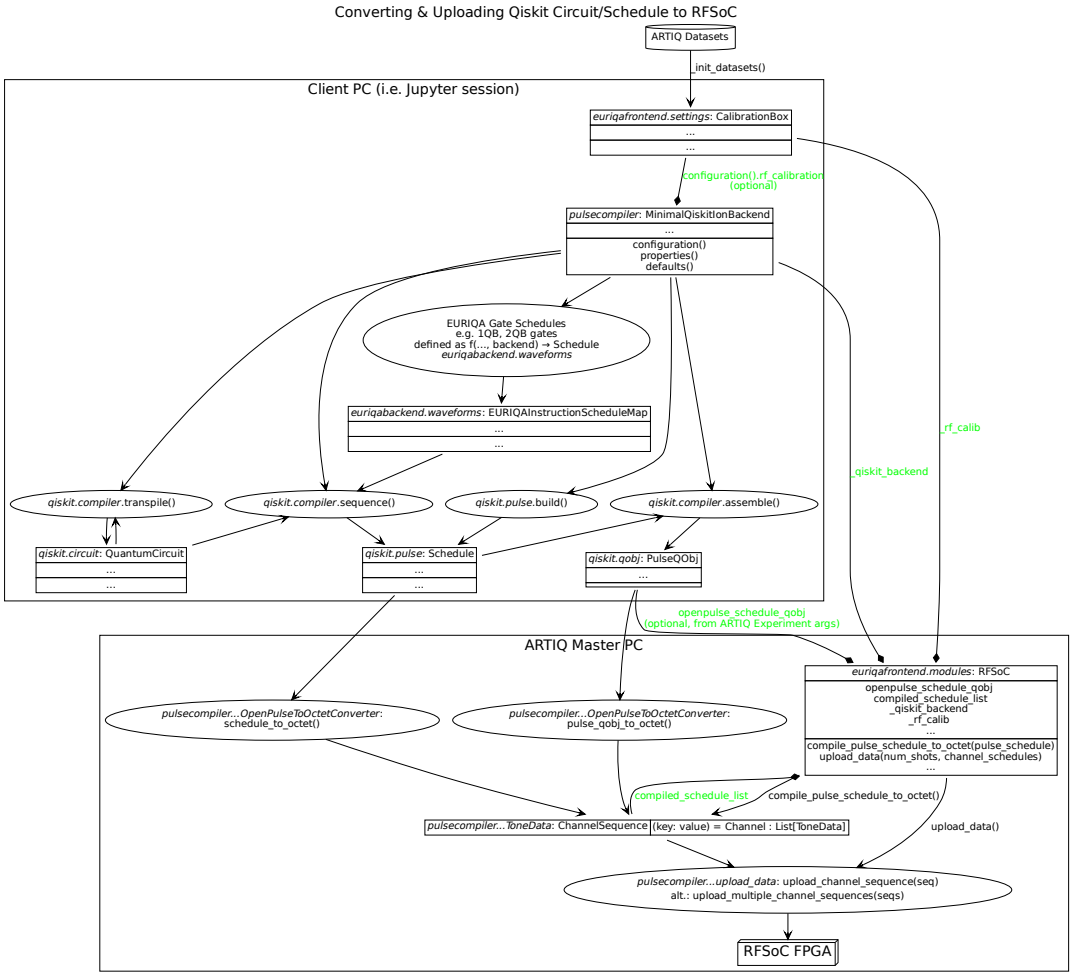


Figure 6.3: **Diagram of the different inputs needed to convert a Qiskit `QuantumCircuit` to a `ChannelSequence`.** Incoming arrows denote inputs to a given entity. Squares are classes, ovals are functions. Italics denote the package that a class/function can be found in. Green text on a diamond arrow denotes that a class attribute (text) of the pointed class is an instance of the arrow's source. Not every class is necessary in this diagram: for instance, the `PulseQObj` can be bypassed entirely if serialization is not needed, or a `CalibrationBox` is not needed if your gate waveforms do not require calibrated values.

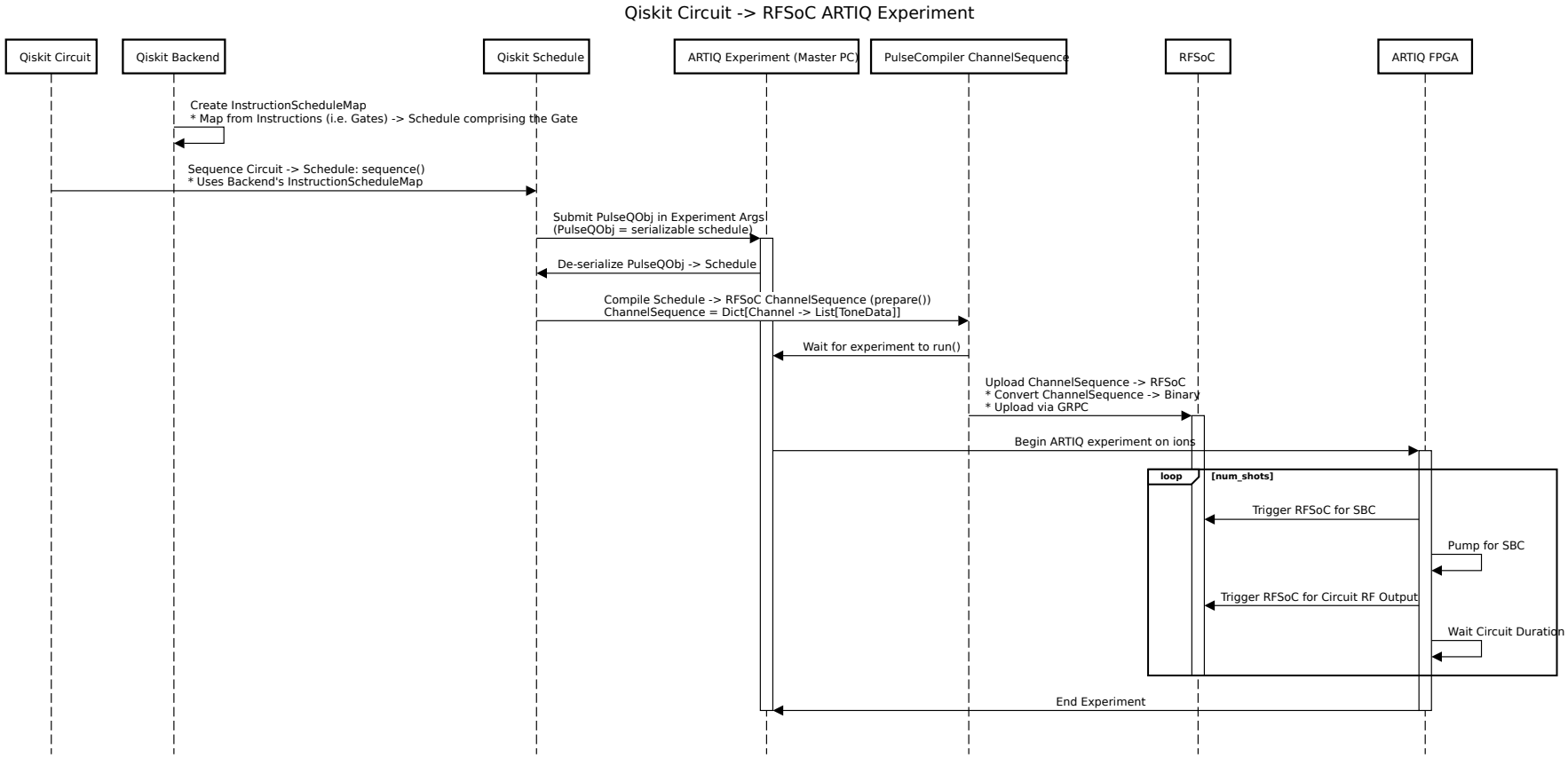


Figure 6.4: **Diagram of the order of operations for converting a `QuantumCircuit` to run on an RFSoC.** Once a `QuantumCircuit` is converted to run on an RFSoC, it can be triggered and executed as described in Chapter 5.

Note in Fig. 6.3 that only the top few functions are defined in end-users' code, which primarily relate to generating and calibrating the gate waveforms. If the most up-to-date calibration properties are not desired (i.e. using uncalibrated gate waveforms), then the Gate Schedules do not need any interaction with ARTIQ or the `CalibrationBox` class at all.

6.4.1 What is a Qiskit Backend?

A Qiskit Backend is a data structure that contains all that Qiskit needs to know about your hardware in order to compile a QuantumCircuit for it.

The exact attributes that need to be in the Backend are poorly-documented by Qiskit, but a minimal example was implemented in `pulsecompiler.qiskit.backend.MinimalQiskitIonBackend`.

Broadly, a Backend has 3 main types of data:

- *Configuration*: Basic information about the backend, which tends to be static. Examples: number of qubits, whether it supports OpenPulse, the OpenPulse ParametricPulses that it supports, and a mapping between qubits and the corresponding OpenPulse channel.
- *Properties*: measurements/status of the backend. Can be anything from calibrations, T1/T2 times, gate definitions (in terms of Schedules), etc.
- *(Unsupported/unimplemented)*: how to submit a schedule/circuit to run on the backend.

When transpiling/sequencing/assembling a QuantumCircuit/Schedule object for the Backend, it will pull default values for any desired arguments from the Backend's `properties()` or `configuration()` as needed. If it fails to find the needed property, the Qiskit conversion method will typically fail.

If you are defining a set of gates for your experiment that depend on calibration values, it makes sense to add the required calibration values as a data structure to `configuration()`, as demonstrated in Fig. 6.3 with `CalibrationBox`.

6.4.2 Schedule to Channel Sequence Conversion

Once a schedule is generated for a given pulse sequence, that schedule must still be converted to a sequence of `ToneData` that will be sent to the RFSoC. This is somewhat of a compilation process (hence the name *PulseCompiler*), though it is really closer to parallel-processing a fixed sequence of inputs.

It is important to note that though an OpenPulse Schedule appears parallel, its canonical representation when all parameters are resolved is actually a single linear sequence of instructions. Each instruction is a Python *tuple*, as demonstrated in Listing 6.1, consisting of a pair of `(timestamp, instruction)`.

```
1  import qiskit.pulse as qp
2  with qp.build() as sched:
3      chan = qp.DriveChannel(0)
4      qp.set_frequency(100e6, chan)
5      qp.set_frequency(3.14, chan)
6      qp.play(qp.Constant(100, 1.0), chan)
7
8  print(sched)
9  "((0, SetFrequency(100000000.0, DriveChannel(0))), (0, SetFrequency
       (3.14, DriveChannel(0))), (0, Play(Constant(duration=100, amp
       =(1+0j)), DriveChannel(0))))"
```

Listing 6.1: Example of creating and printing a simple Qiskit Schedule.

The key idea behind the schedule conversion process is that there are 3 channels of data that are contained in an OpenPulse schedule: Frequency, Phase, and Amplitude. However, only the Amplitude channel will actually generate any output, so every other data is effectively a "virtual" channel.

The algorithm to process the input schedule instructions is then the following:

1. *Divide/distribute instructions* by output channel.
2. *For each channel*:
   (a) *Group instructions* by "virtual" channel.
   (b) *Process frequency & phase instructions*: this creates a First-In-First-Out (FIFO) queue, with each entry consisting of a struct containing: (instruction, current value, instruction type (either set or shift), difference from previous value).
   (c) *Generate ToneData for each amplitude instruction*:
      i. *Check frequency & phase*: move to the next value in the queue if the amplitude instruction's start time is after the current amplitude instruction's start time.
      ii. *Convert amplitude instruction to ToneData*: This will generate a partially-complete `ToneData` with only the amplitude specified.[6]
      iii. *Apply current frequency & phase* to the amplitude `ToneData`.
      iv. *Apply other metadata* such as triggers.
   (d) *Append ToneData to ChannelSequence* for the given Channel.
3. *Upload ToneData* to the RFSoCs.

[6]Qiskit supports several types of pulse instructions, currently including Constant (square), Gaussian, GaussianSquare (Gaussian ramp up/down with square pulse inserted in the middle for longer time at maximum power), and DRAG [126]. Of these, we have chosen to support all but DRAG.

Chapter 7: Experiments with Ion Trap Control System

7.1 Experiments Enabled by RFSoC Control System

The development of this entire control system has been incremental over time, but its usefulness has already been demonstrated in several experiments, including: demonstrating logical qubits on a trapped ion quantum computer [39, 158], investigating measurement-induced phase transitions [159], NMR simulations [160], and demonstrating interactive protocols that could demonstrate quantum advantage [43]. The new capabilities added by the RFSoC-based control system described in Chapter 5 have enabled and will continue to enable using our ion trap quantum computer to demonstrate novel techniques and physics experiments [161]. Specifically, the new features that primarily enable new experiments and techniques are:

1. *Simultaneous control* of all output channels.

2. *Selecting output pulses* based on real-time communication.

This chapter will discuss the demonstrations and results that we have been able to achieve using the control system and improved RF capabilities that we have discussed in previous chapters.

7.2 N-Body Gates

One problem that faces near-term experimental quantum computers is a disconnect between the gates that they can demonstrate, and the theoretical capabilities of a quantum computer. One of many consequences of this is in the gates and qubit connectivity that the quantum computer has available. A given algorithm may be specified assuming any possible quantum gate/operation can be performed, but the operations that it expects might not be able to be executed in a single "native" operation (see Section 2.6.3). If an operation cannot be natively performed on a quantum computer, then a software algorithm is often responsible for "decomposing" the nominal operation into native operations that can be executed on the particular hardware.

One so-called "textbook" gate that is not typically available on quantum computing hardware is a Toffoli gate [38, 162]. This is a gate that flips the state of a target qubit only if the other qubits in the gate are in a given state (i.e. $|1\rangle$), and is sometimes called a CCNOT (Controlled-Controlled-NOT). It is heavily used for reversible digital logic in quantum circuits, and plays an important role in many quantum algorithms [163, 164, 42, 165].

While the concept of a Toffoli appears to be similar to the standard textbook CNOT gate, it actually takes significant overhead to implement compared to a CNOT gate [166, 167]. In the current NISQ [168] era, the overhead of $\approx 6\times$ more gates means that this gate is usually impractical because its use implies a significant fidelity tradeoff. For example, because a Toffoli gate is optimally decomposed into $6\times$ 2-qubit gates

(and some single-qubit gates) [166], [1], the resulting fidelity of a Toffoli gate will be $\mathcal{F}_{Toffoli} \approx (\mathcal{F}_{2\ qubit})^{N_{2\ qubit\ gates}} \approx (\mathcal{F}_{2\ qubit})^{6}$. If each of the 6 two-qubit native gates has a fidelity of $98\,\%$, then the resulting equivalent Toffoli gate will have a fidelity of $\approx 88.5\,\%$. Even $\mathcal{F}_{2\ qubit} = 99\,\%$ two-qubit gate fidelity will only produce $\approx 94.1\,\%$ Toffoli fidelity. Thus, to enable running complex algorithms on a NISQ device, there exists a need for high-fidelity multi-qubit gates, especially for Toffoli gates.

There are two approaches to achieving high-fidelity non-"textbook" multi-qubit gates: special-purpose gates for precisely the Hamiltonian that you choose[2]; or a composition of high-fidelity basic operations such as the Clifford set, which can be combined to produce any desired Hamiltonian. This trade-off between using "standard" vs "special-purpose" gates at the compiler level has been explored previously [169, 170]. Here we will consider a technique for creating one of these "special-purpose" multi-qubit gates for a trapped ion quantum computer.

7.2.1 N-body Gate Scheme

To achieve high-fidelity N-body Toffoli gates, a different gate scheme is needed. Standard techniques such as the Mølmer-Sørensen gate (MS gate) [1, 131] only operate on two qubits at a time. Other techniques are available for operating on multiple qubits simultaneously [171, 172], but they are typically not extensible to arbitrary Hamiltonians.

A recent option that has been developed by our research group is a general N-

[1] We choose to neglect the fidelity of one-qubit gates because most modern systems are primarily limited by their two-qubit gate fidelity. A standard one-qubit gate fidelity on the EURIQA Breadboard is about $99.5\,\%$ fidelity [93], so it is reasonable for a rough estimate to only consider two-qubit gate fidelity.

[2] It should be noted that one potential downside of a custom "made-to-order" quantum gate is that it can potentially require significant calibration effort, which can drift over time and require periodic re-updating.

body gate specific to trapped ions [173]. This technique modifies the standard Mølmer-Sørensen gate, which is based on techniques from [1, 2, 3] (described in [93]). Remember from Section 3.7.3 that a standard MS gate works by generating a spin-dependent force, which excites collective motion in[3] the chain of ions. Which direction that the "target" ions move is then dependent on whether the qubit is in $|0\rangle$ or $|1\rangle$, which causes the "target" qubits to accumulate different state-dependent phase that creates an effective R_{XX} gate.

In our research group's N-body gate concept developed in [173], the basic structure is the same as the MS gate. However, Katz *et al.* make the insight that an MS gate only excites motion at the first-order red and blue sidebands. If motion is instead excited at the second-order sideband, then a quantum state-dependent[4] *squeezing* or *anti-squeezing* effect is produced. Similarly to an Mølmer-Sørensen gate, this squeezing effect is still state-dependent and also generally insensitive to motional noise.

Pictorially (Figs. 7.1 and 7.2 (a)), the N-body interaction can be constructed out of linear segments interspersed with alternating squeeze-antisqueeze segments, which together create a rectangle in phase-space. The area of the rectangle then controls the accumulated phase of the gate.

This concept is shown in Fig. 7.1. Note that the frequencies for Fig. 7.1 (c) are twice as far from the resonant frequency as the frequencies in Fig. 7.1 (b). To create a complete gate, there must be linear displacement segments to create some area (Fig. 7.1 (b)), which are then scaled by the squeeze-anti-squeeze segments shown in Fig. 7.1 (c).

[3] vibrates

[4] The more accurate representation would be that it is dependent on the spin state of the qubit, but in the case of $^{171}Yb^{+}$ the spin ($|\downarrow\rangle$/$|\uparrow\rangle$) maps directly to the quantum state ($|0\rangle$/$|1\rangle$).

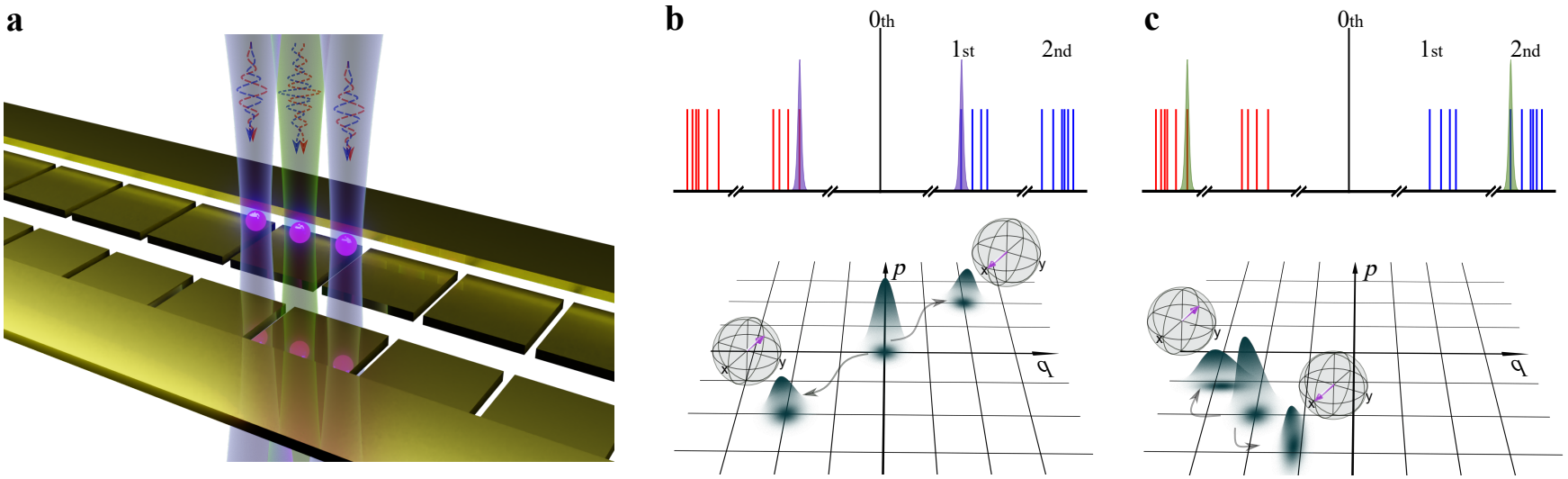


Figure 7.1: **Native N-body operations of a trapped-ion quantum processor.** **a**, Linear chain of laser-cooled ion spins in a chip trap. An array of optical beams addressing individual ions enable exquisite control over the state of each spin and its coupling to collective phonon-modes using Raman transitions (additional global beam forming the Raman pairs not shown). **b**, Spin-dependent displacement of one phonon-mode and its representation over the acoustic phase-space. Simultaneous driving of the first red and blue sideband transitions of one ion near resonance with one phonon mode displaces the phonon wavepacket (gray Gaussian) into two distinct trajectories, depending on the state of that spin along the X direction over its Bloch sphere (purple arrow). **c**, Spin-dependent squeezing of one phonon-mode. Simultaneous resonant driving of the second red and blue sideband transitions of one ion and one phonon mode squeezes (anti-squeezes) the phonon wavepacket along one direction of phase-space depending on the state of that spin along the x direction. Thin-lines in the 2D plots of (**b**-**c**) represent the sideband spectrum and narrow red and blue Gaussians represent the spectral content of the Raman coupling near the sidebands.

In Eqs. (7.1) to (7.3), we define:

- U is the unitary applied to the quantum system
- T is the total duration of the gate
- $\hat{\Phi}_{seq}$ is the gate phase accumulated by the entire waveform sequence.
- ξ is the squeezing factor applied, see [173, Eq. 4].
- $\bar{\xi}$ is the mean squeezing amplitude, $\bar{\xi} = \sum_i \frac{\xi_i}{N}$
- $\mathbb{1}$ is the identity operator on the quantum system.
- $\hat{\sigma}_{\phi_i}^{(i)}$ is the Pauli spin-flip operator of spin i set by the average phase ϕ_i of the ith pair of drives. $\hat{\sigma}_{\phi_i}^{(i)} \equiv \hat{\sigma}_x^{(i)} \cos\phi_i + \hat{\sigma}_y^{(i)} \sin\phi_i$. ϕ_i can be thought of as the axis that phase will be applied to: if the spin (qubit) is aligned to the axis ϕ, then a phase flip will be applied.
- A, B are the displacement in the phase-space regime by position and momentum, respectively. The multiple of these two factors is effectively the phase of the gate when no squeezing is applied, for a gate that makes a square in phase-space, along the lines of what was proposed in [131].
- $T_N^{(n)}$ is the N-qubit Toffoli, where the conditional spin flip is applied to the nth qubit out of N total qubits.

The effective unitary that the gate sequence *seq* applies is:

$$U_{seq}(T) = e^{-i\hat{\Phi}_{seq}} \tag{7.1}$$

where

$$\hat{\Phi}_{seq} = 2ABe^{\hat{\xi}} = 2AB\prod_{i=1}^{N}\left(\mathbb{1}\cosh\xi_i + \hat{\sigma}_{\phi_i}^{(i)}\sinh\xi_i\right) \tag{7.2}$$

In the limit of $\xi_i \gg 1$, Eq. (7.2) simplifies to:

$$\hat{\Phi}_{seq} \to 2ABe^{N\bar{\xi}}\prod_{i=1}^{N}\frac{1}{2}\left(\mathbb{1} + \hat{\phi}_{\phi_i}^{(i)}\right) \tag{7.3}$$

When Eq. (7.3) is substituted into Eq. (7.1), the applied gate becomes the N-qubit controlled-phase gate. If we let $2ABe^{N\bar{\xi}} = \pi$, and add single-qubit $\mathrm{R_Z}$ rotations on the target qubit n, the resulting unitary is:

$$T_N^{(n)} = {R_\mathrm{Z}}^{(n)}\left(\frac{\pi}{2}\right)U_{seq}{R_\mathrm{Z}}^{(n)}\left(-\frac{\pi}{2}\right) \tag{7.4}$$

The squeezing effect on momentum space relies exponentially on the number of spins in the chosen basis ϕ_i, as can be seen in Fig. 7.2 (a). It is important to note that only a moderate amount of spin squeezing is required to achieve a high-fidelity ($\geq 99\,\%$) CCNOT ($T_3^{(3)}$ also known as the 3-Toffoli) gate using this scheme, as illustrated in Fig. 7.2 (b). The matrix representation of the CCNOT gate is:

$$T_3^{(3)} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{7.5}$$

7.2.2 Executing N-Body Gate

To demonstrate the N-body interactions discussed in Section 7.2.1, we first consider the conventional MS interaction between two ions in a chain of three ions. Following cooling and spin initialization in the $|\downarrow_z^{(1)}\downarrow_z^{(3)}\rangle$ (equivalently $|0X0\rangle$, where X symbolizes an unspecified state on the middle ion) state via optical pumping, we resonantly drive the lowest-frequency radial phonon mode ("zig-zag" mode) with a sequence of displacement operations that alternately act on the two edge ions (see Fig. 7.5 (a)), generating a rectangular-shaped loop in motional phase-space as shown in Fig. 7.3 (a) [131].

The accumulated geometric phase corresponds to the phase-space area enclosed in the loop, which is given by $\Phi = \Phi_0 \sigma_x^{(1)} \sigma_x^{(3)}$. We control $\Phi_0 = \alpha^2$ by scaling the amplitude of the displacement pulses while fixing the total duration of the sequence to about $180\,\mu$s. We suppress the displacement of other phonon modes by pulse-shaping of

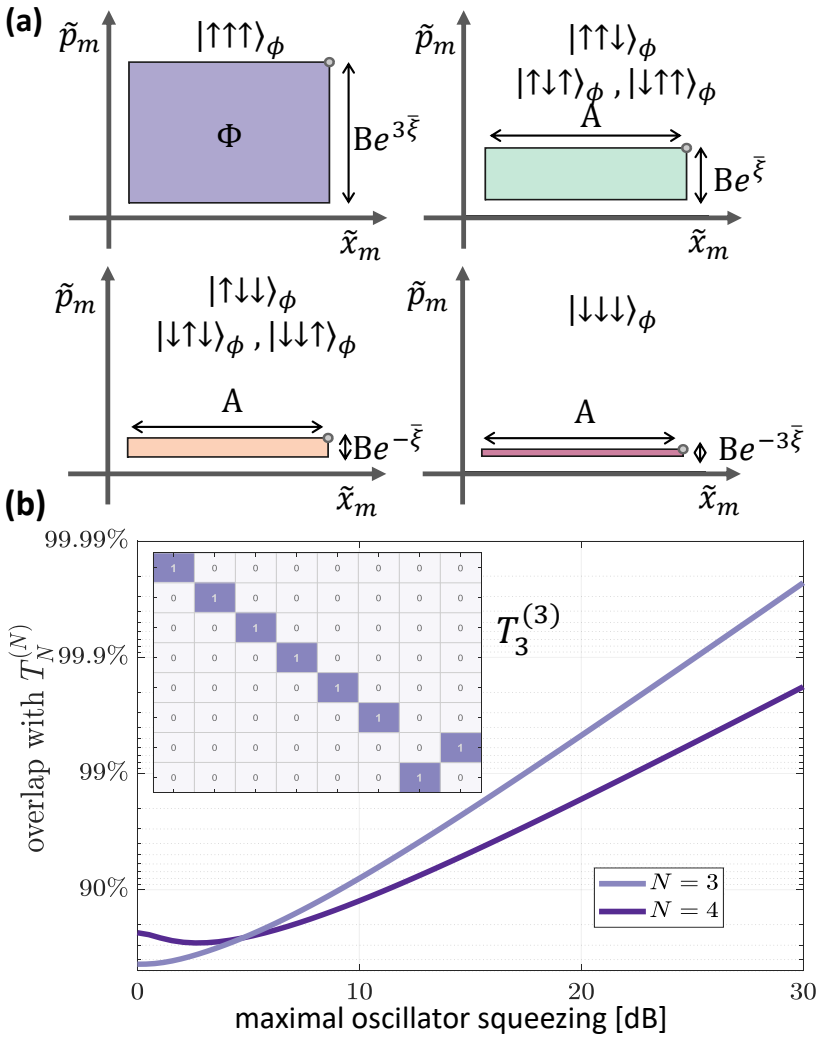


Figure 7.2: **3-body entangling gates (3-Toffoli)** **(a)** Phase space evolution for three spins, following the sequence of alternating spin-independent displacements and spin-dependent squeezing operations. Each ion squeezes the momentum quadrature of the mth motional mode by a factor $e^{-\bar{\xi}}$ if its spin points downwards ($|0\rangle$), and anti-squeezes the momentum quadrature of the mth motional mode by a factor $e^{+\bar{\xi}}$ if its spin points upwards ($|1\rangle$). The state-dependent phase-space area $\hat{\Phi}_{seq}$ accumulated in the evolution generates the gate $U_{seq} = e^{-i\hat{\Phi}_{seq}}$ (Eq. (7.1)) with a maximal squeezing of the oscillator mode by a factor $e^{N\bar{\xi}}$ when all spins are aligned. The total area enclosed by the phase-space rectangle depends on the number of spins along the axis ϕ_i: if 3 spins are along ϕ_i then the largest rectangle is produced; if 2 spins are along ϕ_i then the next-largest rectangle is produced; if 1 spin is along ϕ_i then the second-smallest rectangle is produced; if no spins are along ϕ_i then the smallest rectangle is produced.
(b) Overlap between the many-body gate sequence from Eq. (7.2) (assuming the single-qubit rotations from Eq. (7.4)), and the 3-Toffoli gate from Eq. (7.5). Inset: ideal $T_3^{(3)}$ operator in the computation basis ($|0\rangle$ & $|1\rangle$).

the displacement waveforms [132] and also suppress the effect of uncompensated level shifts using a pair of echo pulses (see Fig. 7.5). Application of this phase-gate jointly flips the spins into the state $|\uparrow_z^{(1)}\uparrow_z^{(3)}\rangle$ with probability $p_{(\uparrow_z^{(1)}\uparrow_z^{(3)})} = \sin^2(\Phi_0)$, which is detected via state-dependent fluorescence and displayed in Fig. 7.3 (a). We determine the scale of Φ_0 by fitting the data in Fig. 7.3 (a) to a sine function as a function of the Raman beam intensity.

We extend the pairwise interaction by interspersing squeezing operations in the sequence that act only on the middle spin and predominantly drive the zig-zag phonon mode (see Fig. 7.5b). These operations are realized as pairs of squeezing and anti-squeezing pulses sandwiching the displacement operations (Fig. 7.3). The squeezing forces scale the momentum displacements by the spin-dependent factor $e^{\xi\sigma_x^{(2)}} \equiv \cosh(\xi)\mathbb{1} + \sinh(\xi)\sigma_x^{(2)}$ where $\mathbb{1}$ is the identity matrix, while nulling the net deformation of the phonon wavepacket. The geometric phase is then given by the scaled rectangular area

$$\Phi = \Phi_0\big(\cosh(\xi)\sigma_x^{(1)}\sigma_x^{(3)} + \sinh(\xi)\sigma_x^{(1)}\sigma_x^{(2)}\sigma_x^{(3)}\big), \tag{7.6}$$

which includes two- and three-body interaction terms.

We demonstrate the action of this phase-gate in Fig. 7.3 (b-c) on the initial states $|\downarrow_z^{(1)}\uparrow_x^{(2)}\downarrow_z^{(3)}\rangle$ and $|\downarrow_z^{(1)}\downarrow_x^{(2)}\downarrow_z^{(3)}\rangle$, for a total sequence time of less than $300\,\mu$s including all displacement, echo and squeezing operations. Similar to the MS gate, this gate jointly flips the state of the two edge spins, but with probability $P_{(\uparrow_z^{(1)}\uparrow_z^{(3)})} = \sin^2(e^{\pm\xi}\Phi_0)$, whose dependence on α^2 is scaled by a factor e^{ξ} ($e^{-\xi}$) and is conditioned on the state of the middle spin pointing upwards (downwards) along the x direction. The calculated evolution

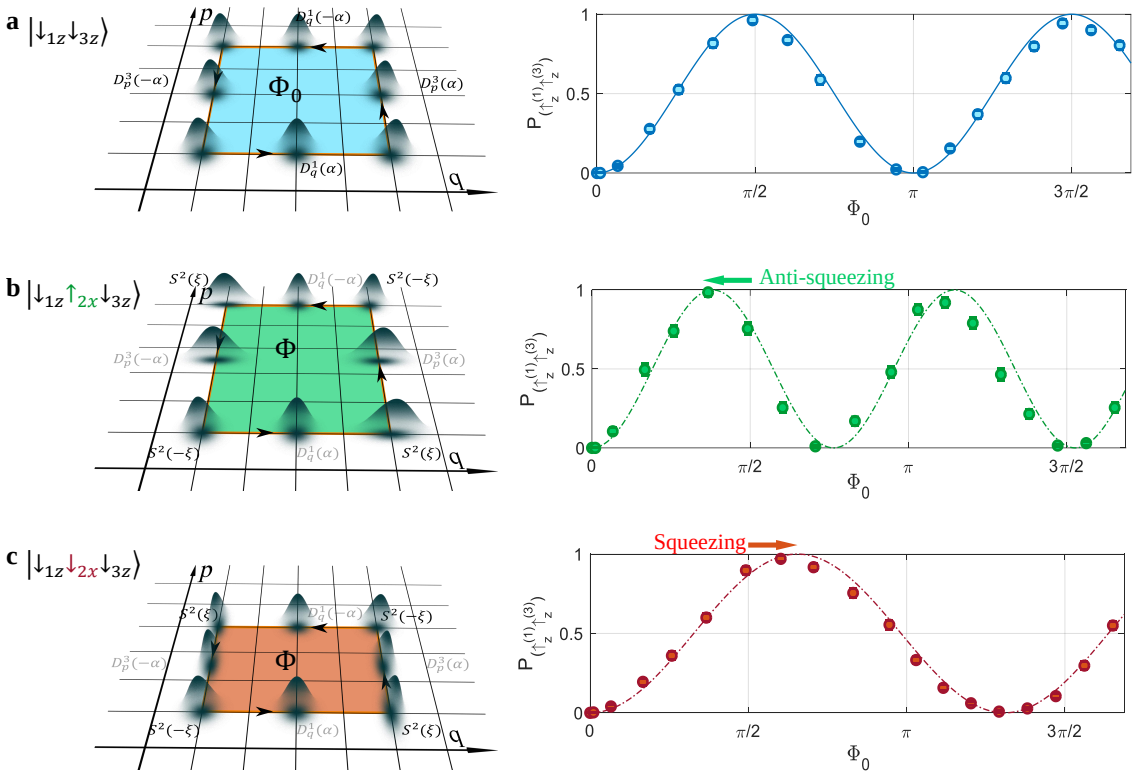


Figure 7.3: **Demonstration of Quantum phase-gates on three ions.** **a**, Mølmer-Sørensen gate between ions number 1 and 3 using displacement operations of one phonon mode using Milburn's scheme [131] as illustrated in Fig. 7.2 (a). The phase-space area of the enclosed rectangular contour Φ_0 controls the spin evolution, jointly flipping the initial state $|\downarrow_z^{(1)}\downarrow_z^{(3)}\rangle$ into the state $|\uparrow_z^{(1)}\uparrow_z^{(3)}\rangle$. $D_q^{(n)}(\alpha)$ and $D_p^{(n)}(\alpha)$ denote position and momentum displacement operations. **b-c**, Interspersing spin-dependent squeezing operations $S^{(2)}(\pm\xi)$ on ion number 2 in-between displacement stages along the p coordinate scales the accumulated phase-space area Φ conditioned on the state of that spin (c.f. Eq. (7.6)). **(a-c)**, The phonon wavepacket is brought back to its original state at the end of the gate operation to erase the spin-phonon correlations developed during the gate. Circles represent measured data, bars represent 1σ binomial uncertainties, and dash-dot lines are the analytically calculated Unitary evolution for the system parameters estimated independently. The applied experimental sequences are presented in Fig. 7.5 (a-b).

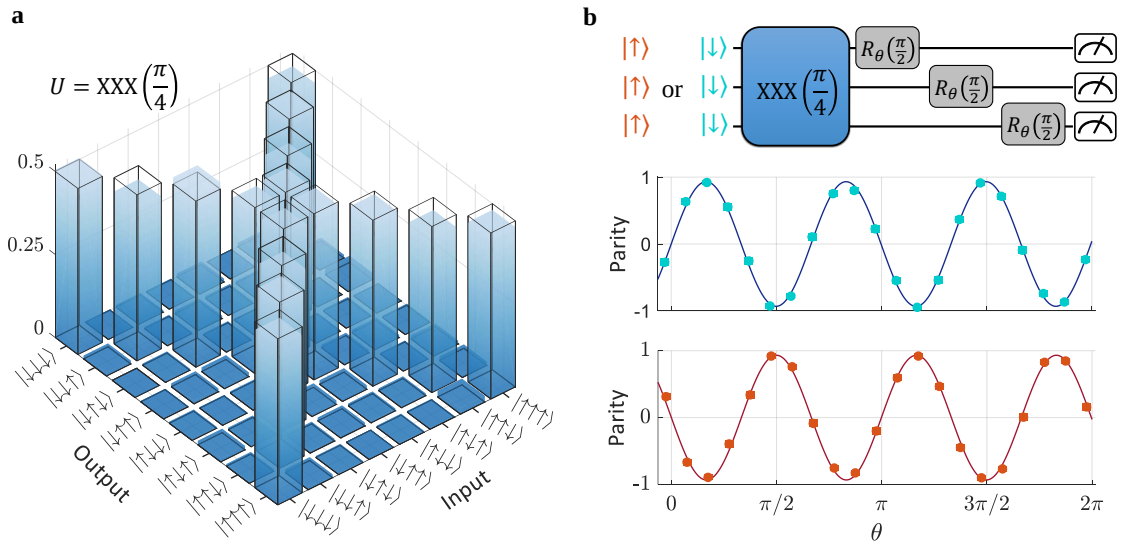


Figure 7.4: **Characterization of a three-body interaction gate.** **a**, Truth table of a three-body gate $\mathrm{XXX}(\frac{\pi}{4}) = \exp(-i\frac{\pi}{4}\sigma_x^{(1)}\sigma_x^{(2)}\sigma_x^{(3)})$ gate generated by sequence of displacement and squeezing operations. The input and output spin states are along the z basis. Each input state is ideally mapped into a pair of output states (wire frames), and raw measurement are shown in solid bars. The measured populations of these target states is $(95.8 \pm 0.9)\%$, averaged over the 8 measured configurations. **b**, Parity measurement of the output GHZ state for the two initial states $|\downarrow_z^{(1)}\downarrow_z^{(2)}\downarrow_z^{(3)}\rangle$(light blue) and $|\uparrow_z^{(1)}\uparrow_z^{(2)}\uparrow_z^{(3)}\rangle$ (red) with a fitted amplitude of 0.932 ± 0.015 for the two states. The extracted GHZ fidelities for this operation for the two states are $\mathcal{F} = (94.8 \pm 1.5)\%$ and $\mathcal{F} = (94.4 \pm 1.9)\%$. Bars represent 1σ binomial uncertainties and dashed lines are sine functions. The operation $R_\theta(\pi/2)$ denotes single qubit rotation by azimuthal angle θ and a polar angle $\pi/2$ on the Bloch sphere.

of $\xi = 0.27$ agrees well with observation.

This many-body entanglement operation features full control over the amplitudes of the two- and three-body terms appearing in Eq. (7.6). We can, for example, eliminate the contribution of the two-body term by setting $\Phi_0 = \pi/\cosh\xi$ and generate a pure three body term with amplitude $\pi \tanh\xi$. We note that maximally entangled states between three spins in this case requires only 1 dB of squeezing ($\tanh\xi = \frac{1}{4}$) [173].

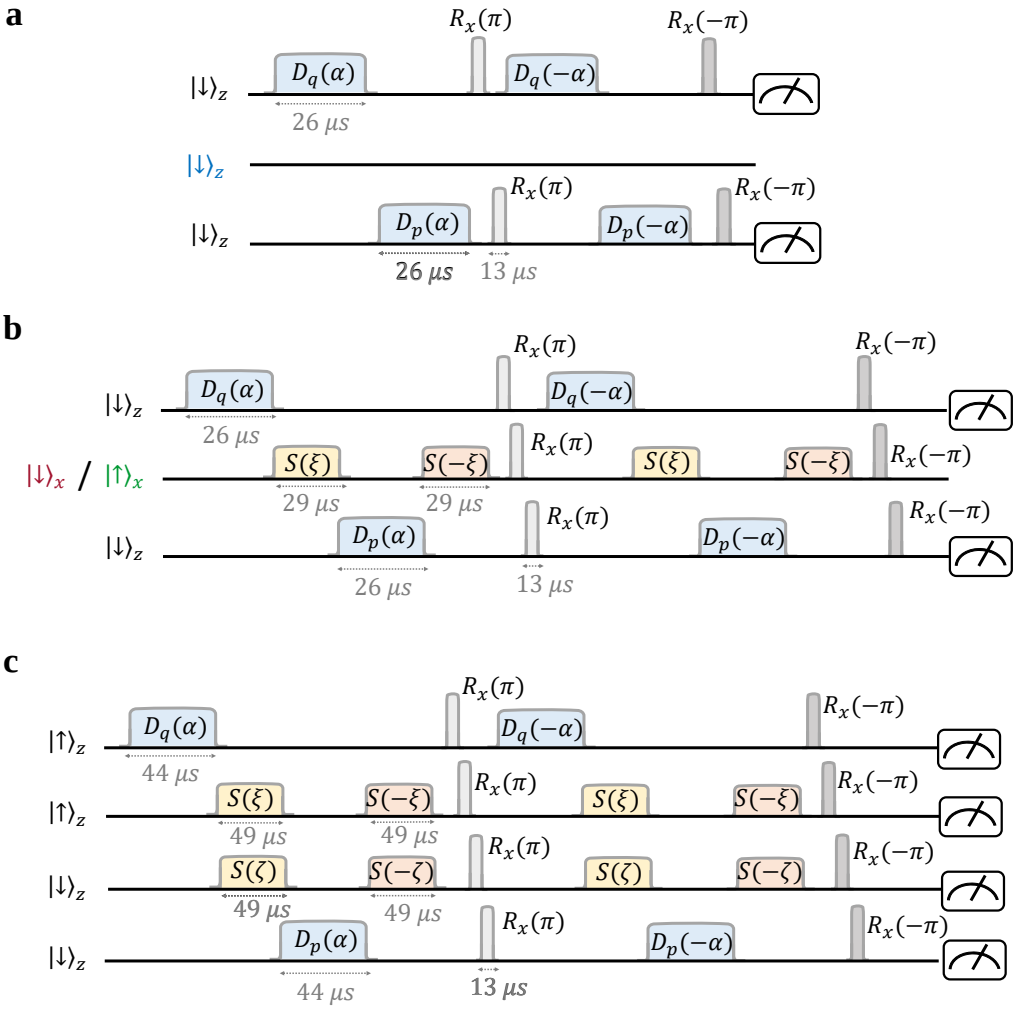


Figure 7.5: **Experimental N-Body gate sequences.** **a**, Sequence of displacement operations acting on the two edge ions and composing the MS interaction, enclosing a closed rectangular loop in phase-space and generating the evolution Fig. 7.3a. **b**, Superimposing spin-dependent squeezing operations on the second spin scales the displacement generated by the third spin by a factor $\exp(\sigma_x^{(2)}\xi)$ and consequently also the enclosed phase-space area. This sequence was applied to the configurations in Fig. 7.3 (b-c) and in Fig. 7.4 and Fig. 7.6 (a). **c**, Displacement of the edge two spins and simultaneous squeezing of the middle spins for the four ions configuration presented in Fig. 7.6 (b). The simultaneous squeezing scales the displacement generated by the fourth spin by a spin-dependent factor $\exp(\sigma_x^{(2)}\xi + \sigma_x^{(3)}\zeta)$ as seen from the identity $S^{(m)}(-\xi)D_p^{(n)}(\pm\alpha)S^{(m)}(\xi) = D_p^{(n)}(\pm e^{\sigma_x^{(m)}\xi}\alpha)$. The operators $D_p^{(n)}(\pm\alpha)$ and $D_q^{(n)}(\pm\alpha)$ denote displacement of the target phonon mode via the nth ion by $\pm\alpha$ along the p and q coordinates respectively. $S^{(m)}(\pm\xi)$ denotes the squeezing operator acting on ion m and $R_\theta^{(n)}(\pm\pi)$ denotes short single-qubit π-pulses acting on the n th ion, which commute with the spin-dependent displacement operations and which correct for slowly-varying uncompensated Stark shifts without altering the target state.

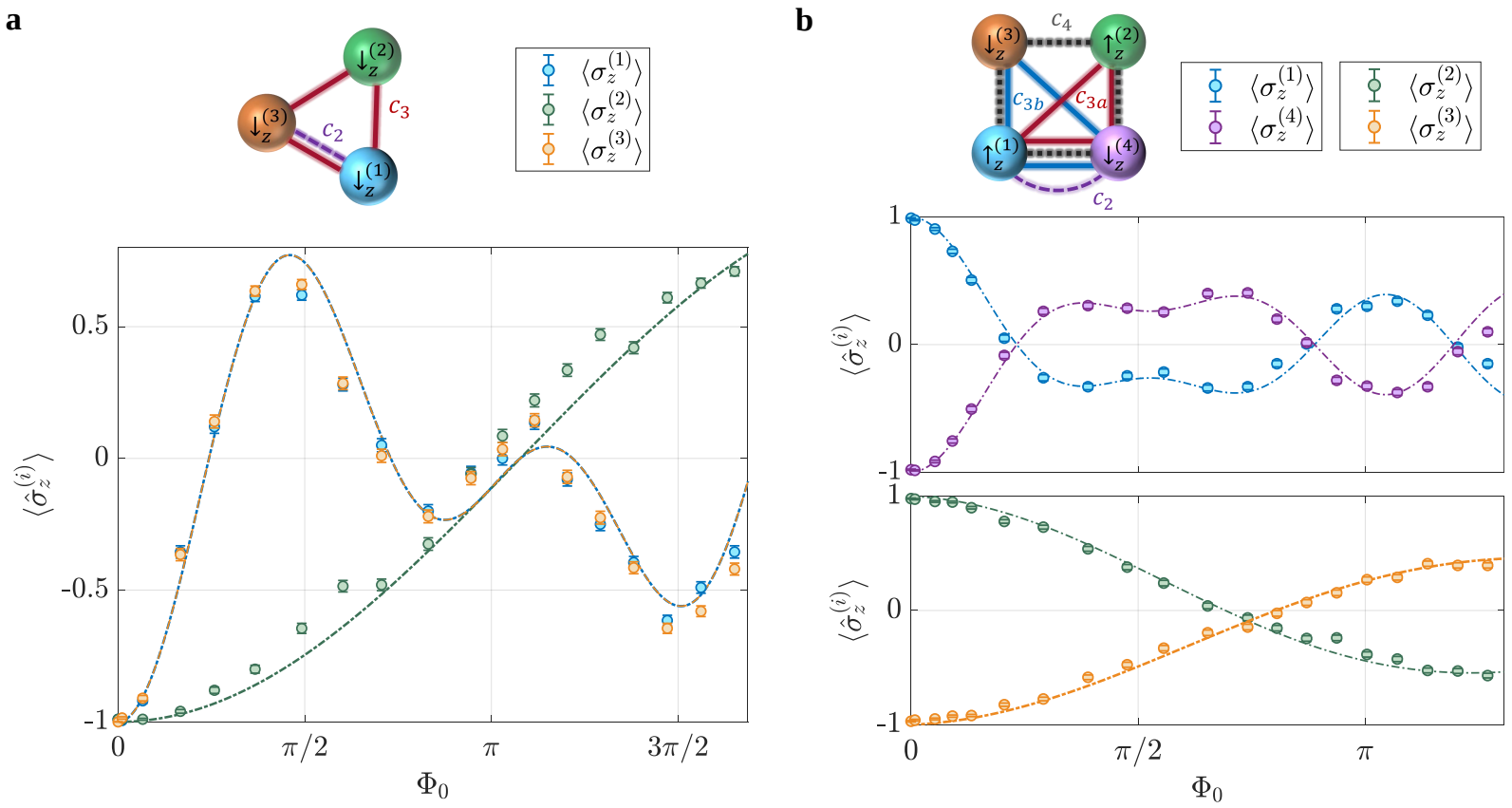


Figure 7.6: **Effective Hamiltonians with three- and four-body interactions.** **a**, Evolution of three spins by the effective Hamiltonian $H_{\text{eff}} = \hbar\Phi/T$ for Φ in Eq. (7.6) for the initial state $|\downarrow_z^{(1)}\downarrow_z^{(2)}\downarrow_z^{(3)}\rangle$. The effective Hamiltonian comprises two- and three-body terms whose magnitudes $c_2 = 1.03$ and $c_3 = 0.23$ as illustrated via the links connecting the different spins. The evolution as a function of Φ_0 is realized with a fixed sequence time. **b**, Evolution of four spins by the effective Hamiltonian in Eq. (7.7) containing simultaneously two-, three- and four-body terms with amplitudes $c_2 = 1.1$, $c_{3\text{a}} = 0.36$, $c_{3\text{b}} = 0.31$ and $c_4 = 0.1$. Dot-dashed lines are the analytically calculated magnetizations for the same initial state $|\uparrow_z^{(1)}\uparrow_z^{(2)}\downarrow_z^{(3)}\downarrow_z^{(4)}\rangle$ where the c amplitudes are determined from the calculate squeezing parameters. Bars represent $1\,\sigma$ binomial uncertainties. The applied experimental sequences are presented in Fig. 7.5 (b-c).

7.2.3 N-Body Gate Results

We perform a limited characterization of this pure three-body interaction by measuring the output state distribution given each of the eight distinct three-qubit input eigenstates, all in the Z basis. The ideal population distributions of the expected GHZ-type states are equal weightings of the two complementary three-qubit states for each input state, shown as wireframes in Fig. 7.4 (a). The measured spin population distributions are shown as solid bars (see Fig. 7.7 for the numerical values), resulting in an average population fidelity of $95.8(9)\,\%$[5], uncorrected for SPAM errors. We further measure the coherence in this three-body mapping from two particular input states $|\downarrow_z^{(1)}\downarrow_z^{(2)}\downarrow_z^{(3)}\rangle$ and $|\uparrow_z^{(1)}\uparrow_z^{(2)}\uparrow_z^{(3)}\rangle$ into the expected GHZ states $\frac{1}{\sqrt{2}}(|\downarrow_z^{(1)}\downarrow_z^{(2)}\downarrow_z^{(3)}\rangle \pm |\uparrow_z^{(1)}\uparrow_z^{(2)}\uparrow_z^{(3)}\rangle)$. We measure the entanglement of these particular GHZ states using the parity fringe witness observable [174] as shown in Fig. 7.4 (b), and extract state fidelities $\mathcal{F} = 94.8(15)\,\%$ and $\mathcal{F} = 94.4(19)\,\%$ respectively, uncorrected for SPAM. We estimate that the leading sources of errors are technical, including noise in the beams' amplitude, motional noise of the oscillator, and uncompensated Stark-shifts.

Our scheme allows the realization of a continuous set of gates produced by the unitary evolution under an effective Hamiltonian $H_{\text{eff}} = \hbar\Phi/T$ at an effective time T that is independent of the actual gate time, but instead scales linearly with Φ_0. We demonstrate the evolution by the effective Hamiltonian associated with Eq. (7.6) for $\xi = 0.23$ and for the initial state $|\downarrow_z^{(1)}\downarrow_z^{(2)}\downarrow_z^{(3)}\rangle$, presenting the magnetization $\langle\sigma_z^{(n)}\rangle$ of each spin in Fig. 7.6. The observed spin evolution manifests interference effects owing to the interplay between

[5]Here the notation $95.8(9)\,\%$ indicates that there is an uncertainty of $\pm 0.9\%$ in the least significant digit, i.e. 8

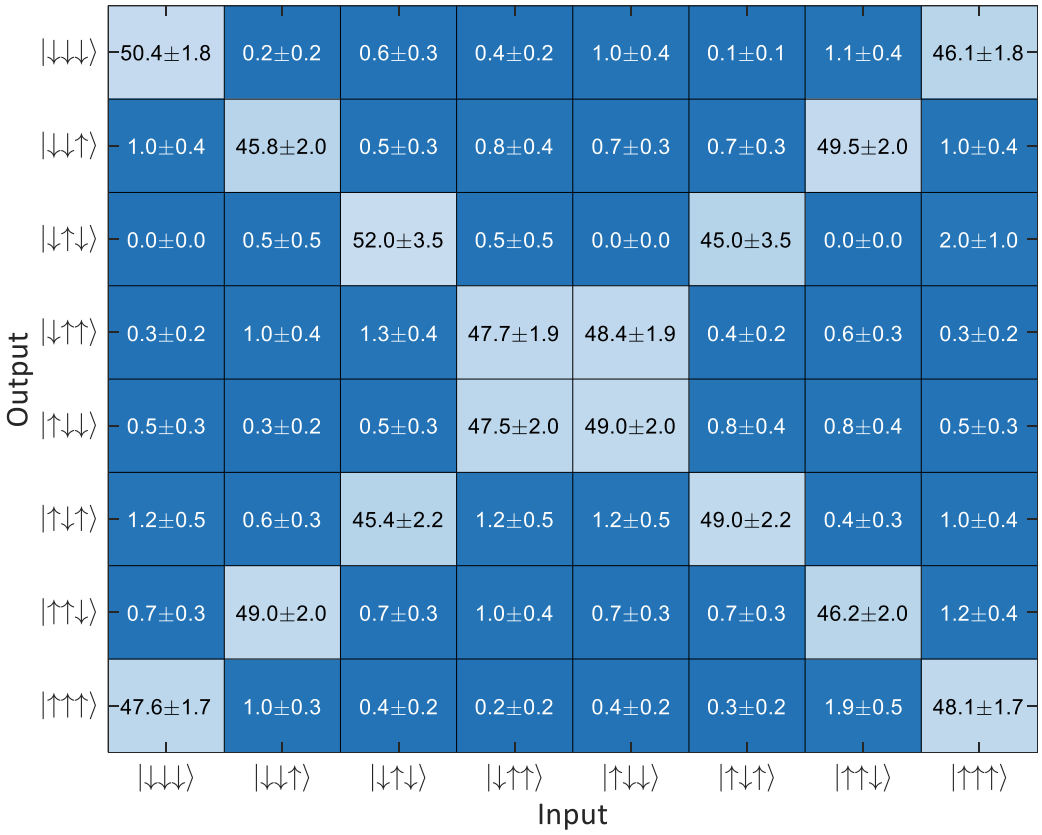


Figure 7.7: **Raw measurement statistics for the XXX$(\frac{\pi}{4})$ gate.** The spins are along the Z basis, corresponding to Fig. 7.4. The numbers are in percent, followed by 1σ binomial uncertainties.

the two- and three-body terms in the Hamiltonian, and is in good agreement with the analytically calculated evolution (dot-dashed lines).

We extend this technique to generate an effective Hamiltonian in a four-ion chain. As in the three-ion gate, we act on two edge ions with displacement operations, while the squeezing is performed simultaneously on the two middle ions with squeezing parameters ξ and ζ as shown in Fig. 7.5 (c). Unlike the displacement operation that is linear in the spin operators, the total scaling factor of the phonons, $e^{\xi\sigma_x^{(2)}}e^{\zeta\sigma_x^{(3)}}$, is multiplicative in the spin operators (Eq. (7.2)), owing to the nonlinear nature of the squeezing operation.

Therefore, we realize an effective Hamiltonian that is associated with the geometric phase of the scaled rectangle:

$$\begin{aligned}\Phi = \Phi_0 \Big(& c_2 \sigma_x^{(1)} \sigma_x^{(4)} + c_{3\mathrm{a}} \sigma_x^{(1)} \sigma_x^{(2)} \sigma_x^{(4)} \\ & + c_{3\mathrm{b}} \sigma_x^{(1)} \sigma_x^{(3)} \sigma_x^{(4)} + c_4 \sigma_x^{(1)} \sigma_x^{(2)} \sigma_x^{(3)} \sigma_x^{(4)} \Big)\end{aligned} \tag{7.7}$$

with two-, three- and four-body terms having relative amplitudes $c_2 = \cosh\xi \cosh\zeta$, $c_{3\mathrm{a}} = \sinh\xi \cosh\zeta$, $c_{3\mathrm{b}} = \cosh\xi \sinh\zeta$ and $c_4 = \sinh\xi \sinh\zeta$. In Fig. 7.6b we demonstrate the evolution by this effective Hamiltonian for the initial state $|\uparrow_z^{(1)} \uparrow_z^{(2)} \downarrow_z^{(3)} \downarrow_z^{(4)}\rangle$ and the applied values $\xi = 0.34$ and $\zeta = 0.29$, following calibration of Φ_0. The evolution in this case manifests interference between the four different terms in the Hamiltonian, and is in good agreement with the theoretically calculated evolution (dot-dashed lines).

Chapter 8: Advanced Ion Trap Operations

Enabling new applications in trapped ion quantum computing will require advancements in certain fundamental ion trap operations. Certain operations with ion trap quantum computers are more advanced and exploratory than those covered in Chapter 3. These advanced operations will be useful for improving the state of the art in trapped ion quantum computing, by enabling applications such as multi-species ion chains and logical qubit operations. However, as these applications are still in the research phase, we have identified certain unaddressed problems that these applications will encounter. The remainder of this chapter will discuss several of these operations that have not been fully studied, as well as new contributions to each of these operations.

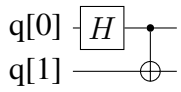
8.1 Ion Indexing

Executing quantum circuits on physical hardware typically requires a mapping from "virtual" qubits to "physical" qubits. A physical qubit refers to a physical device that encodes quantum information, and is subject to all the imperfections of a real system. In an ion trap system, this is effectively a single ion in the chain.[1] On the other hand, virtual qubits are qubits specified in a given circuit. Virtual qubits are specified without regards to any

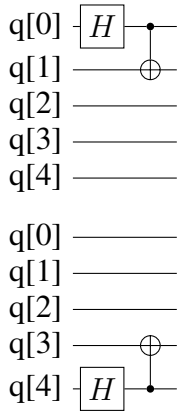
[1] More accurately, the atomic spin of a single ion in a chain.

physical limitations, such as the underlying fidelity of the physical quantum gates that will be executed.

For example, suppose that a 5-qubit ion trap quantum computer executes the following Bell circuit:

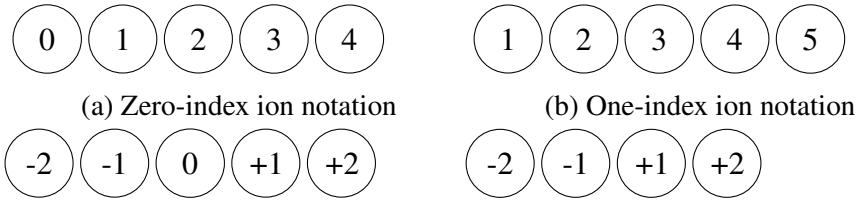


When the quantum computer executes the two-qubit circuit, some sort of compiler will assign two of the five physical qubits to execute the quantum circuit. Thus the following two example representations of this circuit below are equally valid from a conceptual perspective, though they are different in their physical execution.



Given the difference between physical and virtual qubits, it is important that a physical qubit conveys meaning about the physical state of the system. Trapped ion quantum computers have a unique opportunity to convey physical meaning through the index.

Typically, qubits have one of two numbering schemes: zero-indexed or one-indexed. These indexing schemes are straightforward. Indices are assigned beginning with zero (one), and then continuing with all integer (whole) numbers from there. These numbering schemes are illustrated in Figs. 8.1a and 8.1b.



(a) Zero-index ion notation

(b) One-index ion notation

(c) Center-index ion notation (odd number of ions)

(d) Center-index ion notation (even number of ions)

Figure 8.1: **Ion Physical Qubit Indexing Options** These indexing schemes are illustrated for ion chains of $N_{ions} \in \{4, 5\}$, though they can be extended to an arbitrary number of ions. For new systems under development, we recommend solely using center-index notation wherever possible, though the other indexing options might still be desirable in certain use cases. Some examples include when working with certain quantum circuit packages such as Qiskit, which does not support negative qubit indices natively.

These rudimentary indexing schemes have a few inconveniences that arise in situations that we regularly encounter on our EURIQA Breadboard system. These include:

- *No support for changing number of qubits*: The number of ions (physical qubits) in an ion trap quantum computer changes relatively often. In a room temperature ion trap, such as EURIQA Breadboard, collisions with background gases can cause ions to be ejected from the chain, leaving only a few ions post-collision where there might have been a dozen before the collision. [2] For example, if the center ion in a chain of 15 is labeled as 7 (zero-indexed), and then a chain loss event occurs where only one ion remains, its index will suddenly change from $7 \rightarrow 0$, while still retaining approximately the same position (and thus laser beams, etc).

- *Doesn't convey symmetry*: Many operations and issues in ion traps are a function of the ion's position in the chain. These operations also are usually symmetric relative to the ion's position in the chain, such as crosstalk or multi-qubit gate waveforms.

[2] In cryogenic traps, the number of ions present will generally not change due to background collisions, but primarily due the experimental physicist's desire to use a different number of ions for a given quantum algorithm.

The zero-index and one-index notation do not convey this information, without extra information about the number of ions in the chain, and even then still require mental math to understand this symmetry.

8.1.1 Center-Ion Indexing

The solution that we created to solve both of these issues is to use a "center-indexing" scheme, illustrated in Figs. 8.1c and 8.1d. This scheme assigns ion indices as follows:

- *For **odd** numbers of ions*: The center ion is assigned to 0. The next ion to the right of index 0 is assigned $+1$, while the next ion to the left of index 0 is assigned -1. [3] This process continues until all qubits are assigned indices.

- *For **even** numbers of ions*: There is no center ion, so the center-most ions are assigned -1 and $+1$ for the left and right sides, respectively. The left side is then assigned indices from -2 until the end of the chain, and the same process with the right side beginning at $+2$.

The idea is that the center ion will always be denoted as 0 (if even), and then ion index denotes distance from the center of the chain while showing symmetry. For example, if a chain loss event happens and only a single ion is left, then Ion 0 will refer to the same physical location both before and after the ion loss event. This also has the advantage of the same ion index referring to the same ion position, regardless of number of ions (as long as the same parity is maintained), as well as conveying information about symmetry, as it is clear that e.g. ± 1 refers to opposite ions relative to the center.

[3] We do not specify how the orientation of "left"/"right" are specified, as that should be determined by the physical system and its designers.

Conversion between the different indexing schemes can be calculated as a function of the number of ions. We let:

- N_{ions} be the number of ions in a chain (or the number of indices allowed).
- I_C be a center-indexed ion position (index). E.g. $\{-2, -1, 0, 1, 2\}$. Correspondingly, $I_{C_{even/odd}}(I_Z, N_{ions})$ denotes the equivalent center index for a zero-indexed position, when the number of ions N_{ions} is either even or odd. [4]
- I_Z is an index in zero-index notation. Then $I_Z(I_C, N_{ions})$ is the zero-indexed equivalent of a given center-indexed ion index.

The index conversions are then calculated as:

$$I_{C_{odd}}(I_Z, N_{ions}) = I_Z - \left\lfloor \frac{N_{ions}}{2} \right\rfloor \tag{8.1}$$

$$I_{C_{even}}(I_Z, N_{ions}) = I_{C_{odd}}(I_Z, N_{ions}) + \begin{cases} 1, & I_{C_{odd}}(I_Z, N_{ions}) \geq 0 \\ 0, & \text{otherwise} \end{cases} \quad [5] \tag{8.2}$$

$$I_Z(I_C, N_{ions}) = \begin{cases} I_C + \left\lfloor \frac{N_{ions}}{2} \right\rfloor, & \text{if } I_C \mod 2 == 0, \text{ or } I_C \mod 2 == 1 \ \& \ I_C < 0 \\ I_C + \left\lfloor \frac{N_{ions}-1}{2} \right\rfloor, & \text{otherwise} \end{cases} \tag{8.3}$$

[4] Remember that even/odd can be calculated as $I_C \mod 2$. Even is $I_C \mod 2 == 0$, and odd is $I_C \mod 2 == 1$

[5] The conditional addition here is to prevent the index 0 being assigned as a potential index.

8.2 Ion Split-Merge Overview

There is a certain level of basic capability of working with chains of ions that are necessary to demonstrate for minimum functionality: the ability to shuttle an ion through a junction on a surface trap[175, 105], construct chains of arbitrary size, and load ions. After these capabilities are demonstrated, another desirable feature is the ability to split and merge a chain into any permutation of the component ions. This capability has been considered at the architecture level before [176, 177], but needs more research at an implementation level.

While there is a wealth of physics involved in implementing this operation [178], here we will mostly be focusing on the system requirements needed to implement these operations. We also discuss some of the building blocks of a chain split-merge operation to consider how certain other operations that we would like to implement are modifications to the basic structure of a split-merge operation.

The goal of a chain *split-merge* operation is to isolate a subset of all the ions in the chain (called here a *subchain*), and then re-integrate those ions into the overall chain. This process is generally accomplished by modulating the ion trap DC voltages to create voltage potentials that move the ion chain, physically, in specified sequences. While the process of generating these voltage sequences (sometimes called *voltage solutions*) is nontrivial, here we will assume the correct sequence can be generated and then output exactly as desired, neglecting any imperfections in the trap voltage DACs and the filter boards.

First, we must first define some terms to clarify split-merge operations. Remember

that a chain of ions ($N_{ions} \geq 1$) is trapped in a voltage potential well that confines the ions. This voltage potential can be "moved" (i.e. the shape of the voltage potential can be translated by manipulating the trap DAC voltages), or modified to affect the positions of the ion chain residing in the well. For simplicity, we will abstract away the voltage potential well to only consider the ion chain that resides in the well. [6] We define the ion chain of size N_{ions} as $C_{N_{ions}}$. We define split operations on a chain $C_{N_{ions}}$ as occurring between the j-th and $(j+1)$-th ions, denoted by $S_{N_{ions},j}$. [7] Finally, we abbreviate a merge operation between two chains $\{C_k, C_l\}$ as M_{C_k,C_l}.

With this simplification, we can then consider the order of operations to execute the following split-merge operation in an example 3-ion chain (using center-index notation): $C_3 \rightarrow S_{3,-1} \rightarrow (C_1, C_2) \rightarrow M_{C_1,C_2} \rightarrow C_3$.

1. *Create minuscule voltage barrier* between ions $(j, j+1)$, i.e. $(-1, 0)$.

2. *Increase voltage barrier*, until the split occurs.

3. *Move new sub-chains apart.* This marks the end of the split operation, as the two chains C_1, C_2 can now be moved independently without affecting the other.

4. *Move sub-chains back together.* This marks the beginning of the merge operation. At this point, there is essentially a finite-height barrier between these chains (i.e. the voltage wells), which will need to be lowered to merge the two chains.

[6] Calibrations will usually need applied to these wells, especially as the number of ions in a well increases. In this regime, the ions are more sensitive to the background electric fields, which can lead to undesired effects such as the ion spacing changing. This can also cause a previously-effective voltage sequence for splitting an ion chain no longer splitting the chain between the desired two ions.

[7] Either center-index (Section 8.1) or zero-index notation will both be valid here for j, though we will use center-index notation to convey chain symmetry.

5. *Lower barrier height* until the two chains C_1, C_2 merge.

With this foundational sequence, we can add layers of complexity to execute certain operations that quantum information theory omits.

8.3 Ion Chain Sorting

The first of these operations is the problem of sorting (or re-ordering) an ion chain into the desired configuration, which is typically required after a collision with a background gas particle scrambles the chain. In other words, this problem is ensuring that the chain has the desired ordering of ion species. This is a relatively recent problem, because many prior trapped ion quantum computers were only working with a single ion species (i.e. isotope) and small numbers of ions. [8]

A second set of ion species is typically desired for reducing the motional energy of the chain (i.e. cooling) via e.g. Doppler cooling of a second ion species. Thus we sometimes call these *coolant ions*. This might be particularly useful for long quantum computations (to cancel the background heating rate of the ion trap), or to achieve high fidelity gates after operations that add motional energy to the chain such as a split-merge or shuttling operation.

Remember that with only a single ion species, every ion is effectively identical, so not only is there any deterministic way to tell if the ion chain has re-ordered, but the chain will never need to be re-sorted. [9]

[8] For small numbers of ions, a sorting operation is actually not required because the voltage potential well can be squeezed (increasing the slope of the barriers), which will cause the chain to reconfigure so that the lighter ions are towards the outside [179].

[9] The one exception to this is when a collision with a background gas particle causes an ion to "go dark". In this context, "go dark" means that the ion does not emit fluorescence on the cycling transition described

However, when a second (or more) ion species is introduced, the ordering of ion species now matters. The primary effect of different chain configurations is that it will change the vibrational mode structure and participation ratios. [10] In turn, this changes the gate waveforms that need to be played back to execute a gate between any pair of ions [179]. Thus, if a chain reordering event occurs, there are two choices:

- *Do nothing*: continue operating with the current chain configuration.
- *Sort the chain*: reorder the chain into the desired ion configuration.

To be able to analyze these approaches, we need to consider chain sorting operations in more depth.

Defining a few terms: N_{ions} is the total number of ions; $n_{type1}, n_{type2}, \ldots$ are the number of ions of each different species (with the constraint that $N_{ions} = \sum_{type} n_{type}$). Using discrete mathematics principles, we can determine that the number of unique orderings of a chain of N_{ions} with a given number of ions of different species (i.e. different groups/types) is:

$$O_{N_{ions},n_{type1}=X,n_{type2}=Y,\ldots} = \frac{N_{ions}!}{\prod_{type}(n_{type}!)} \tag{8.4}$$

For a simple chain of 3 ions, with $n_{type1} = 1, n_{type2} = 2$, then $O_{3,n_{type1}=1,n_{type2}=2} = \frac{3!}{2!1!} = 3$. For a 15-ion chain, which is roughly the scale that we have used for logical qubit operations, there are an order of magnitude fewer possible chain configurations by increasing the ratio of type 1 to type 2 ions (i.e. increasing the ratio of computational ions

in Section 3.6.3. We have hypothesize that the most common cause of "going dark" is the ion transitioning into the low-lying F state ($^2F_{7/2}$) due to a collision. Interestingly, we can intentionally use these "dark" ions to measure the rate of reordering collisions.

[10] A secondary effect is that the mapping of control sources (e.g. gate waveform sources) to ions will need to change in real-time to only control the physical qubits used in computations, instead of the coolant ions.

to coolant ions).

$$O_{15,n_{type1}=8,n_{type2}=7} = \frac{15!}{8!7!} = 6435 \tag{8.5}$$

$$O_{15,n_{type1}=13,n_{type2}=2} = \frac{15!}{13!2!} = 105 \tag{8.6}$$

8.3.1 Chain Sorting Operation

To sort an ion chain into a desired configuration, several steps must happen in sequence. This sequence builds on the split-merge operation from Section 8.2 by performing two sequential splits, and then performing a swap operation between the two remaining ions. To be more specific, the sequence *for a single swap* is as follows, assuming a chain of $N_{ions} \geq 4$, and that the ions that should be swapped are ions $(i, i+1)$ in zero-index notation:

1. *Split 1*: Split the chain between ions $(i-1, i)$, yielding two new subchains of lengths i and $N_{ions} - i$ (i.e. $S_{N_{ions},i-1} \rightarrow (C_i, C_{N_{ions}-i})$). Note that this step is unnecessary if the ions to swap are the 0th & 1st ions.

2. *Split 2*: Split the right subchain $C_{N_{ions}-i}$ between (in original whole-chain numbering) $i+1, i+2$. Symbolically, this is the operation $S_{N_{ions}-i,2} \rightarrow (C_2, C_{N_{ions}-i-2})$. Now there are three subchains: $C_i, C_2, C_{N_{ions}-i-2}$. [11]

3. *Swap*: For the C_2 chain, reverse the order of the two ions so that $(i, j) \rightarrow (j, i)$. This still leaves C_2 (a subchain of 2 ions).

[11] Similarly, this step is unnecessary if the ions to swap are the last two ions.

4. *Re-merge 1*: Re-merge the left subchain with the middle subchain: $(C_2, C_{N_{ions}-i-2}) \rightarrow M_{C_2,C_{N_{ions}-i-2}} \rightarrow C_{N_{ions}-i}$. [12]

5. *Re-merge 2*: Re-merge the left subchain with the right subchain: $(C_i, C_{N_{ions}-i}) \rightarrow M_{C_i,C_{N_{ions}-i}} \rightarrow C_{N_{ions}}$.

Note that this operation is symmetric, so the voltage sequence for each split can theoretically be run in reverse to produce the equivalent merge operation. We will abbreviate this swap operation as $Swap_{i,i+1}$.

Once a basic swap operation can be executed between any pair of ions, that operation can now be integrated into a sorting algorithm to produce a desired chain ordering. Fortunately, the computer science community has invested much research into sorting algorithms, which we can take advantage of. However, ions have the constraint that they are a physical item, and they cannot be swapped between arbitrary indices in an array as digital information can. In other words, to sort $\{2, 1, 0\}$, you must perform the operations $Swap(1, 0) \rightarrow Swap(2, 0) \rightarrow Swap(2, 1)$; this cannot be done in a single operation for a chain of ions.

Thus, to actually sort a chain of ions to the desired configuration, it is important to select a sorting algorithm that is near-optimal with respect to the total number of swaps that need performed. This is a different constraint than most sorting algorithms, which tend to emphasize performance as a function of the number of elements to be sorted. When sorting an ion chain, the number of elements to sort is fixed, but their configuration is unknown until the time of the sort. We choose to emphasize the total number of swaps

[12] This re-merge can be performed either before or after re-merge 2.

to be performed as the metric for selecting an algorithm, because the number of swaps is a primary factor in the equation for determining how long it will take to sort an ion chain.

The time to sort an ion chain can be approximated as

$$t_{total\ sort} = t_{calculate\ swaps} + (t_{read\ configuration} + t_{swap}) \\ *(n_{swaps} * \frac{1}{success_{swap}}) + t_{read\ configuration} \tag{8.7}$$

In this equation, we define:

- $t_{total\ sort}$: total time to perform a sorting operation.
- $t_{calculate\ swaps}$: time for a CPU to calculate the sequence of swaps to change from the initial chain configuration to the desired chain configuration.
- $t_{read\ configuration}$: time for the ion chain configuration to be recorded. We assume that this is performed after every swap to confirm the swap, though this frequency could be reduced to decrease the total measurement time. This operation will typically look very similar to the measurement operation described in Section 3.6.3, except Doppler cooling of a single ion species is used instead of only reading out $|1\rangle$, and non-computational ions are determined by appearing as "dark" (no fluorescence). A final configuration check is performed at the end to confirm the success of the sorting operation.
- t_{swap}: time to execute the swap operation $Swap_{i,i+1}$.
- n_{swaps}: total number of swaps to execute in the ideal case, as calculated during $t_{calculate\ swaps}$. This is determined by the sorting algorithm.

- $success_{swap}$: We assume some failure rate for the swap operation to provide a margin of error. We constrain $success_{swap} \leq 1.0$.

Because most of these times are constrained by physics and the system design (such as the swap time), we focus here primarily on the number of swaps (n_{swaps}) because that has not been researched elsewhere. By selecting an effective algorithm, it should be possible to reduce n_{swaps} to the minimum amount of time possible. Once we select an algorithm, it should also be possible to simulate how the number of swaps varies with the number of ions in a chain, as well as different ratios of ion species.

8.3.2 Chain Sorting Algorithm

We selected the Merge Sort algorithm [123] for generating the number of swaps, and then performed simulations on the total number of swaps with various configurations of number of ions in a chain and the ratio of coolant ions to computational ions. The Merge Sort algorithm is especially useful here because all of its operations are mostly local, and it has good worst-case performance of $O\left(N_{ions} \log N_{ions}\right)$. [13] [14]

Using the Merge Sort algorithm as a base, we need to perform the following modifications so that it can match the needs of sorting an ion chain:

- *Generate ion ordering*: this step assigns a unique index to each ion, determining where it should fall in the sorted ion configuration. This algorithm works as described in Listing 8.1.

[13] The standard Merge Sort algorithm does not specify that all of its operations are local (only on neighboring data). However, it is relatively trivial to modify the merge operation to only perform nearest-neighbor swaps, without increasing the asymptotic runtime of the algorithm.

[14] Restricting operations to only nearest-neighbor swaps changes the problem into calculating the inversions in a list [180].

- *Return inversions*: Unlike many sorting algorithms, the goal of the Merge Sort algorithm in this context is to return the *required swaps*, instead of the sorted ion configuration. Thus, the Merge Sort algorithm is modified to record every nearest-neighbor swap performed (in order) when sorting the input, and then return that sequence.

```
1     generate nominal (i.e. sorted) chain configuration
2     group nominal configuration by ion type (one sequence per type)
3     Read chain configuration
4     sorted_indices = list()
5     for each ion_index, ion_type in enumerate(chain):
6         correct_index = next index from nominal configuration for
    ion_type
7         sorted_indices.append((ion_index, correct_index))
```

Listing 8.1: **Pseudocode for assigning sorted ion indices to an arbitrary chain ordering.**

8.3.2.1 Simulation Methodology

We performed simulations to calculate the number of swaps needed to produce a desired chain configuration. The goal was to understand how the number of swaps required to reorder into the desired configuration would scale with the number of ions and the ratio of computational to coolant ions.

We began by initializing different chain configurations, with varying numbers of ions (N_{ions}) and ratios of computational to coolant ions $n_{type1} : n_{type2}$ (sometimes we use $\frac{1}{n}$ to denote the fraction of *total* ions N_{ions} that are $type2$). [15] For each "round" of simulation, we randomly shuffle the ions of a chain configuration, assign a target index

[15] For N_{ions} that is not evenly divisible by $n_{type1} + n_{type2}$, any remaining unassigned ions are assumed to be of $type1$. For example, $\frac{1}{4}$ coolant with $N_{ions} = 4$ is 3 computational ions and 1 coolant ion, while $N_{ions} = 5$ will have 4 computational ions and 1 coolant ion.

to each ion (illustrated in Fig. 8.2b, using the algorithm from Listing 8.1) [16], and then calculate the number of swaps/inversions as previously described.

By repeatedly scrambling and then sorting the chain, we can construct a histogram of the number of swaps needed to reorder a chain. Results from these simulations will be shown in Section 8.3.2.2.

This approach has some assumptions that might differ from a physical trapped ion experiment:

- *Expected ion ordering*: This simulation assumes purely random ordering after a reordering collision, which has not been statistically demonstrated in ion chains.
- *No further collisions*: We assume that another reordering collision will not occur during the sorting sequence. Functionally, this means that we assume the sort duration is much less than the interval between reordering collisions: $t_{sort} \ll interval_{reordering\ collisions}$, which might not hold true for a given experiment.

8.3.2.2 Simulation Results

We simulated the dependence of the number of swaps as a function of the number of ions in the chain, as well as various ratios of computational to coolant ions.

One example of the simulation for a given number of ions and ratio of computational to coolant ions is given by Fig. 8.3. This is essentially simulating reordering the 25-ion version of Fig. 8.2a.

[16] This algorithm is straightforward: you can essentially store a sorted queue of correct indices for each type of ion. You then iterate through the scrambled ions, and assign scrambled ions the smallest index from the queue (pop) for the corresponding type. There are likely more-efficient algorithms for this task, but this efficiency is OK for the relatively small scale considered here.

(a) Desired chain configuration.

(b) Example scrambled configuration. Ions are labeled with the "destination" index, which is the index after sorting into the desired configuration.

Figure 8.2: **Example auto-generated ion chain configuration for $N_{ions} = 5$.** Diagonal-line-filled ions denote computational ions, and vertical-line-filled ions denote coolant ions. This example has a Computational : Coolant ($n_{type1} : n_{type2}$) ion ratio of $2 : 3$. When generating chain configurations from coolant ratios, this chain configuration would be represented by the $1 : 1$ ratio (or $\frac{1}{2}$ coolant), due to the odd number of ions. This is demonstrated with 5 ion chain, but the same method applies to larger ion chains. The scrambled configuration shown in Fig. 8.2b is one of the possible permutations (described by Eq. (8.4)). This scrambled configuration is also the *worst-case* configuration for the original chain configuration, as it has the maximum number of swaps (inversions) needed to achieve the desired configuration in Fig. 8.2a: 3 swaps.

While the number of swaps appears reasonable for the length of ion chain that we are currently considering, we should also ensure that this scheme is scalable to the number of ions that we might reasonably expect to have soon. We can simulate this by varying the length of ion chains that we consider, shown in Fig. 8.4.

In our simulation, we can also vary the ratio of computational to coolant ions, as shown in Fig. 8.5. For an example 25-ion chain, we observe approximately a factor of two improvement in the average number of swaps needed to sort the chain. While significant, this is not enough to negate the exponential scaling seen in Fig. 8.4.

8.3.3 Comparison of Recalibration vs Sorting Time Cost

As previously stated, different chain configurations will have different motional mode spectra. In multi-qubit gate schemes that rely on entangling ion spin with the motional

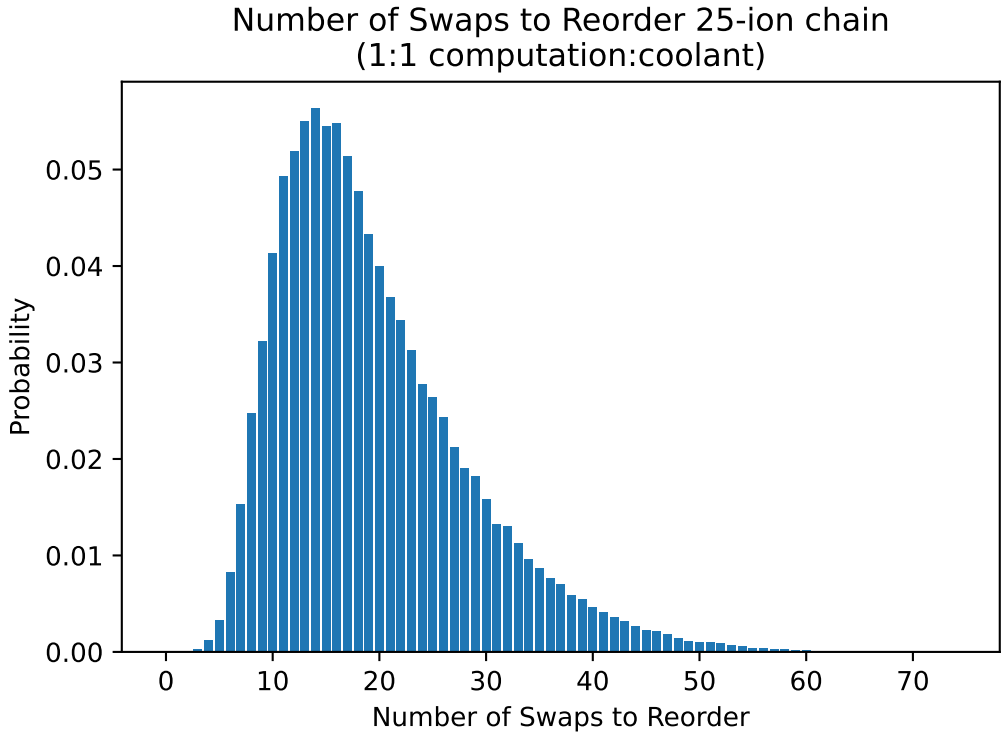
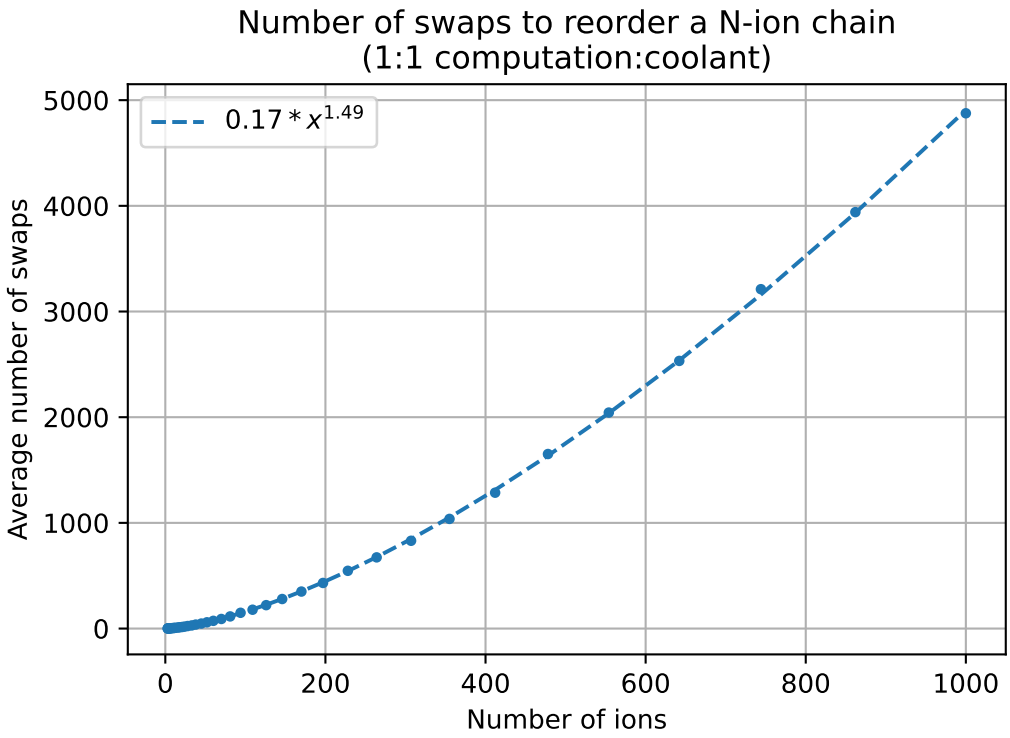
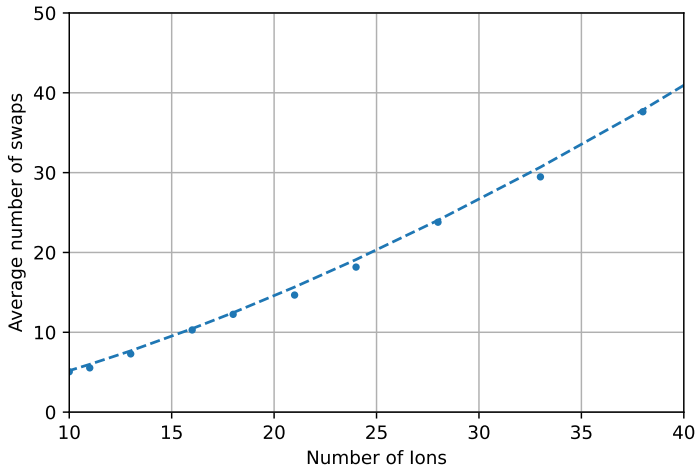


Figure 8.3: **Simulation of the number of swaps to reconfigure a randomly-scrambled 25-ion chain, with $1:1$ computational to coolant ion ratio.** This is shown as probability to reflect the random nature of the chain scrambling. Note that the tail of the distribution goes out to almost 70 swaps, approximately a factor of $3.5\times$ more than the average case.



(a) Plot of $N_{ions} \in [1, 1000]$



(b) Zoomed in on $N_{ions} \in [15, 40]$

Figure 8.4: **Simulation of the number of swaps to reconfigure a randomly-scrambled N-ion chain, with $1:1$ computational to coolant ion ratio.** For simplicity, the results here are plotted as the average of all simulations for a given number of ions. The simulation results (dots) are fitted to an exponential curve: $0.17 * {N_{ions}}^{1.49}$. Each data point is the result of simulating the number of swaps required to reorder 1000 random scrambling events. This shows that we expect this method to scale exponentially with the number of ions, so it might not be applicable to extremely long chains of ions. Fig. 8.4b is the same as Fig. 8.4a, just zoomed into the length of ion chains that are currently practical.

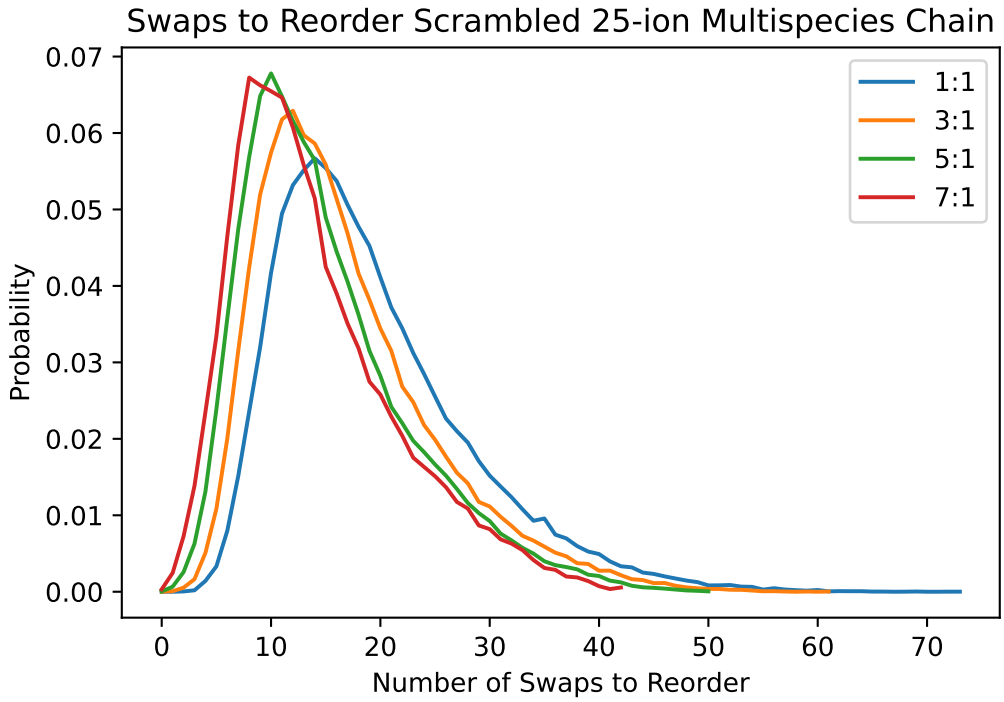


Figure 8.5: **Effect of ratio of computational to coolant ions on the number of swaps needed to sort a scrambled 25-ion chain.** Number of trials per coolant ratio is 100000. Each line is the bars from Fig. 8.3, but plotted as a line instead of a bar for visibility. Note that as the ratio of computational to coolant ions increases, the observed "tail" also decreases, implying that the worst-case number of swaps also decreases (or is at least made more unlikely).

modes of the ion chain, this will cause different ion chain configurations to require different "gate solutions", i.e. sequence of pulse modulations to achieve a given multi-qubit gate unitary [133, 93]. Thus, each chain configuration will require a different set of "gate solutions", which can be pre-computed as there are a finite number of chain configurations.

However, beyond just computing, storing, and loading gate solutions for every possible chain configuration, the issue becomes one of calibration. For each chain configuration, assume that the gate set [17] will need to be re-calibrated after a reordering event. [18] Let us consider the execution time tradeoff between the two primary approaches: using an arbitrary (after a reordering event) chain ordering; and sorting the chain into a desired configuration.

$$t_{sort} = t_{detect} + \left((t_{detect\ swap} + t_{swap}) * N_{swaps} * \frac{1}{reliability_{swap}} \right) + t_{detect} \tag{8.8}$$

$$t_{recalibrate} = t_{detect} + (N_{2Q\ gates} * t_{gate\ calibration}) \tag{8.9}$$

We can use our knowledge of our system to substitute in reasonable estimates for each of these, even assuming a smaller-than-realistic trapped ion system with only 8

[17] The set of all possible gates between each ion. Primarily referring to 2+-qubit gates as those use the motional modes for multi-qubit operations, while single-qubit gates do not. This makes multi-qubit gates sensitive to the ordering of a multi-species ion chain.

[18] We typically find that we need to recalibrate gates every $interval_{recalibrate} \approx 3\,\mathrm{h}$. Thus the only way that this assumption would be invalidated is if the chain reorders into the same configuration twice in a 3-hour window. This is exceedingly unlikely for a chain of 15 ions, assuming random scrambling/reordering.

qubits to reduce the number of two-qubit gates in favor of the recalibration approach:

$$t_{detect} = 200\,\mu\text{s}$$

$$t_{detect\ swap} = t_{detect} = 200\,\mu\text{s}$$

$$t_{swap} = 20\,\text{ms}$$

$$reliability_{swap} = 80\,\%$$

$$N_{swaps} = 50$$

$$N_{2Q\ Gates} = \binom{N_{qubits}}{2} = \binom{8}{2} = \frac{8*7}{2} = 28$$

$$t_{gate\ calibration} = 1\,\text{min}$$

Thus, we can estimate t_{sort} & $t_{recalibrate}$:

$$t_{sort} = (2 * 200\,\mu\text{s}) + \left((200\,\mu\text{s} + 20\,\text{ms}) * 50 * \frac{1}{0.8}\right) \approx 1.3\,\text{s} \tag{8.10}$$

$$t_{recalibrate} = 200\,\mu\text{s} + (28 * 1\,\text{min}) \approx 28\,\text{min} \tag{8.11}$$

Even with somewhat-conservative estimates for reordering times (i.e. imperfect swap reliability, checking each swap, etc), it is still approximately $1000\times$ faster to reorder the ions into the desired chain configuration versus recalibrating the gates. In short, if high fidelity gate operations are the goal, then it should be faster to sort the chain of ions into the desired configuration instead of attempting to re-calibrate.

8.4 Ion-Photon Entanglement Generator

One project that we developed was aimed at quantum networking, as opposed to the quantum computing focus that the remainder of this thesis focuses on. Quantum networking experiments are generally aiming to create a "quantum internet", with quantum information freely flowing between distributed nodes. This idea builds on the concept of quantum teleportation.

A quantum network requires the "resource" of entanglement to operate. The communication rate of a quantum network is limited to the number of shared entangled states that they can create (e.g. Bell pairs). Thus, many research groups are aiming to demonstrate high rates of entanglement generation.

Generating entanglement requires reliably generating single photons that can be interfered on a beamsplitter. However, even if single photons can be reliably generated, the photons' wavepackets must still overlap in space so that they can be entangled, which has tight timing requirements. And once entanglement is generated, care must be taken to ensure that the quantum state is not destroyed/collapsed via measurement, so an *entanglement witness* is observed to verify if entanglement has occurred. [19]

Finally, a device is needed to sequence all of the required operations to attempt to generate entanglement, and continue attempting to generate entanglement until success is heralded. This is a fairly unique set of requirements. The sequencer must be able to:

- *Output very short pulses*: typical pi-times for these systems are on the order of

[19] Entanglement witnesses are sometimes also called heralds. Entanglement is generally heralded by a specific pattern of photon detectors registering inputs.

10 ns.

- *Pattern-match* to confirm that a desired entanglement herald has been detected.
- *Repeat quickly*: the control device should introduce as little overhead into each entanglement attempt loop as possible. This and the short pulse requirement preclude most CPU software for generating and detecting entanglement.
- *Exit at undetermined time*: similar to an interrupt, once entanglement is successfully generated, control flow should return to the standard quantum sequencer so that any further quantum operations can be executed.
- *Integrate* into roughly-standard lab equipment.

I decided to implement this system based on an example design produced by an ion trapping group at the University of Oxford [181, 182, 183, 184, 185]. This system is effectively a semi-autonomous gateware block state machine within the ARTIQ FPGA, called *entangler-core*. [20]

The idea is that this core is dormant when not in use. When in use, it usurps control over the required inputs/outputs. It begins by outputting a sequence of digital signals to the desired channels for a given window, which collectively generate photons from both sets of ions. Then it monitors the input hardware interface blocks (PHYs) for incoming edge signals, rejecting any that fall outside of the expected time window. If the input pattern of signals *does not* match the desired pattern, it resets and again attempts to generate

[20] This gateware block was actually entirely programmed in Python, using a domain-specific language (DSL) called `migen` (as well as the `misoc` library). Using `migen` is practically required to integrate similar functionality into the gateware for an ARTIQ FPGA. The source code for this feature can be found at `https://github.com/drewrisinger/entangler-core`.

entanglement, looping until a certain number of entanglement attempts has occurred. If the input pattern *does* match the desired pattern(s), then it determines that entanglement has (probably) occurred, and it ends executing the loop. Control then returns to the standard ARTIQ processor, which determines the appropriate action to take. [21]

This feature has been demonstrated as fully functional via benchtop testing. However, its full use in a quantum context in our group has not been demonstrated due to the difficulties of hardware-based physics experiments.

[21] The *entangler-core* also includes configuration registers, which allow modifying most of the capabilities described here. For example, the herald pattern(s) can be modified between runs of the entanglement generator, or the pulse timing can be modified.

Chapter 9: Outlook

Trapped Ion Quantum Computers are a leading candidate for developing large-scale quantum computers, with notable benefits of long coherence times and high fidelity operations. Our recent work [39] has demonstrated the feasibility of using trapped ions for logical qubit-scale trapped ion quantum computers, using single chains of ions. From here, the field will likely move in some combination of several research focuses: increasing chain length, moving chains of ions around an ion trap in the Quantum CCD architecture ([105]), and incorporating photonic interconnects [186] for distributed entanglement across multiple "processors".

At this point, the fundamental physics of each of these concepts has been explored in the literature; the next step is engineering and development. This thesis has explored some of the implementation & engineering challenges that each of these capabilities will require, and our initial work of engineering for these future research directions.

The problem that the quantum computing field currently faces is transitioning quantum computers from physics lab experiments to commercial products. This requires a fundamental re-consideration of today's quantum computers; the many details of quantum computing at small scale ($\approx$ 10 qubits) very well may not work at a scale of 1000 or 1 million qubits. Problems such as "Which qubit are we referring to? And what qubits

can it perform operations on?", which are trivial when there are only a handful, become significantly more complex when this information is too large to be stored in a few bits. Accompanying this paradigm change will come a shift in perspective, from considering a handful of qubits as the "norm" to considering them an exception or a special case.

In the intermediate term, there will still be a strong need for improved control hardware and software stacks to enable these changes; near-term quantum computers will still need some form of waveform generator to perform gates, and photonic interconnects will require hardware acceleration to achieve high entanglement generation rates. We look forward to seeing the next generations of quantum computers incorporating these and many other building blocks.

Appendix A: Python: Distribution and Best Practices

A.1 Python Overview

Python is a programming/scripting language that is relatively popular in the scientific community. Some key features that drive its popularity:

- *Readability*: Python code is relatively straightforward for non-programmers to read and write. For example, a hello world program in Python looks like Listing A.1.
- *Libraries*: Python has a variety of libraries (similar to a plug-in) that offer a wide variety of basic functionality in a plethora of scientific and numeric disciplines. Easy-to-use libraries exist for many tasks, which speeds up development times when starting out.
- *Simplicity*: Python is both simple-to-read and simple-to-write. While it has a great depth of features and extensions, beginners can ignore most of these and get started with code that "just works", without having to worry about the mechanics of compiling their code.
- *Cross-Platform*: Python is generally platform-independent, so software that is written on Linux will work equally well on Windows or MacOS (generally). This is a

good fit for labs that are not standardized on a single operating system, or require a specific operating system for compatibility with a specific piece of hardware.

- *Maturity*: Python was begun in 1989. Since its inception, the majority of bugs have been resolved, many beginner-friendly tutorials have been written, and a diverse community has formed around Python.

```
1    print("Hello␣World!")
```

Listing A.1: Hello World program in Python

However, some of these benefits come at the cost of speed. In contrast with more-traditional programming languages like C, Python is not compiled. This means that the Python program must be read-in, processed, & executed all at run-time, in contrast with C which is pre-compiled, and everything except the execution can happen in advance. [1] So there is essentially a trade-off between software execution time (worse in Python) and development time (better in Python).

For many projects, this trade-off is acceptable, but it is not appropriate for every possible application. For example, in applications where latency is critical (i.e. response time), such as embedded systems or a quantum computing experiment, the overhead of Python is not acceptable. Another example is extremely lengthy numeric calculations, such as those performed on a High-Performance Computing (HPC) system, such as nuclear simulations that the USA Department of Energy performs on their supercomputers.

[1] This limitation can be worked around by writing extensions/Python libraries that wrap low-level compiled machine code (called bindings). The machine code can be produced by any programming language, such as Rust, C/C++, etc. However, there will still be some overhead in calling the wrapped machine code.

A.2 Python Packaging

Though Python is one of the most popular programming languages, one perpetual problem that arises due to its dynamic nature is that of dependencies. In a statically-compiled language like C, most dependencies are resolved at compile time. Python does not do this, and it instead relies on the users to have *all* dependencies present on their system in order to run. Generically, the process of defining dependencies, ensuring the necessary dependencies are present, and resolving any dependency conflicts falls under the umbrella of *Python packaging*.

A.2.1 Python Libraries Overview

One of Python's most useful features is its ability to easily use external libraries that other people have written in your own software. To use these libraries, they must be made available (installed) on your local PC.

Roughly, this process involves:

1. Find & Download package
2. De-compress the package
3. Any compilation/setup steps for your specific computer
4. Move the package to a directory where Python can find it
5. Use library in Python

Tool	Pros	Cons
Pip	De-facto standard. Wide support	Doesn't support non-Python/mixed-language projects well
Conda [187] (Anaconda)	Wide package support, interoperable with Pip, support for environments.	Slow to determine dependencies for many requirements. This is partially solved with the re-implementation in `mamba`.

Table A.1: **Comparison of Python Package Managers**

If you are installing a "local" package, typically one that you have written and which resides entirely on your PC, then the download & decompression steps can be skipped.

A.2.2 Python Package Managers

To ease the process of finding & installing these libraries, there are several standard tools.

A.2.3 Python Environments

In a software context, an *environment* is the set of software packages that you have available. We can extend this definition to Python, and call a Python environment the set of all installed Python packages as well as a particular version of the Python interpreter, which processes and executes the Python source code.

It is generally useful to have several different Python environments available on your system. The primary advantage is that it allows logical separation between tasks, which can avoid unintended side effects. For example, some Python package *old* that you might want to use is only available for a specific version of Python, say 3.5, but another package *shiny*, with all the latest features and bugfixes, is only available for Python

$>=$ 3.7. Thus, these two packages cannot co-exist with the same version of the Python interpreter, so they must exist in separate environments. [2] This contrived problem is unlikely to be an issue for a small set of required packages, but Python programs tend to require on many libraries which increases this risk.

There are many different solutions for creating different Python environments (typically called virtual environments) on the same PC. Each of these will perform roughly the same task, and are sorted in approximately descending order of current popularity, based on my personal observations.

1. `conda/mamba` [187]
2. `virtualenv`
3. `venv` (built-in to the Python standard library)
4. `pipenv` [188]
5. `Nix` [30]

Describing how to use each of these tools is outside the scope of this thesis, but extensive documentation exists on each of them. Any option will work well for most use cases, so I recommend either using the standard practice of your group, or just using the most popular ones. [3]

[2] There are sometimes ways to workaround this incompatibility, but they are typically labor-intensive to implement and are unlikely to continuously working due to their hand-crafted nature.

[3] My one recommendation on which Python package manager to use is to avoid Anaconda in favor of Mamba. I have personally found that Anaconda's dependency solver is slow & sometimes error-prone, and Mamba is generally much more responsive, while retaining the same functionality.

A.3 Python Best Practices

Many people who use Python are experimental physicists, who have little prior exposure to Python or general software development best practices. It is helpful to lay out some best practices when using Python, which can both help avoid bugs, teach better software development practices, and guide people to communicate better through code.

This section will lay out my somewhat opinionated set of best practices & tools. This section should be considered a set of guidelines that will have positive long-term impact, and not a rigid manual that should be followed at the expense of larger goals.

A.3.1 Recommended Development Tools

The following is a list of tools that I personally prefer to use, which have either taught me best practices or have improved my software development workflow.

Tool Name	Category	Benefit
Visual Studio Code	IDE	Useful plugins, supports many programming languages.
`black`	Python Auto-Formatter	Standardize code appearance. Faster to read standard code. Also minimizes Git differences. Extremely fast.
`pytest`	Unit Testing	Runs tests on Python code. Can be used to confirm that a feature works as designed, that functionality still works as expected, or that software upgrades have not broken any features.
`pdb`	Debugging	Built-in Python debugger. Can also be triggered by inserting and executing `breakpoint()` in Python code. *Critical* for complex Python programs.

continues on next page

Tool Name	Category	Benefit
`flake8`	Linting[4]	Checks common Python errors and best practices. `flake8-artiq` is a useful plugin for checking for errors in ARTIQ "kernels" run on the FPGA's soft CPU.
`pylint`	Linting	Slower but more in-depth linter than `flake8`. Does not support `flake8-artiq`.
`jupyter`	Interactive Notebooks	Python equivalent of a Mathematica notebook. Useful for prototyping software, or in an experiment workflow for submitting an experiment and then analyzing the collected data.
`sphinx`	Documentation	Can automatically generate documentation for functions and modules if docstrings[5] exist.
`pyment`	Docstring Generator	Auto-generates Python docstrings for a module.

Table A.2: **Suggested Python Development Tools.** This is just a sample of the tools that the author has found most helpful or intriguing. These tools primarily focus on encouraging best software development practices. If best practices in formatting and Python development are followed, it is much easier to collaborate in a team because less time is spent deciphering what legacy code is doing.

A.3.2 Profiling & Optimization

One key aspect of software development that should not be underestimated is the need for, and benefit of, performing software profiling and subsequent optimization. *Software profiling* is the process of benchmarking a piece of software to determine how long it takes to run, while tracking where execution time is spent. After *profiling*, those results can be used to perform *optimization* of certain functionality. These two concepts are intricately linked; first you run profiling to understand where a piece of software is slow,

[4]Linting is static code analysis for finding bugs, bad practice programming patterns, formatting issues, etc. Linting is the primary form of code analysis available for Python because Python is not a compiled language.

[5]Docstrings are the Python way of documenting a variable, class, function/method, or module. I personally prefer the Google-style docstring format, because it is human-readable and requires little memorization.

and then you optimize that. Repeat this process several times, and the resulting software will typically be much faster than the original software. [6]

Python has several tools to aid the profiling process, though there are fewer tools for aiding in code optimization.

A.3.2.1 Python Profiling

Profiling a piece of Python software has a few steps:

1. *Create sample program*: This should either be representative of a typical execution, or a stress test. Typically this looks like writing a sample script (a `*.py` file) that uses the software in question.

2. *Run sample program* in profiling mode. Python has several debugging libraries. A recommended built-in library is `cProfile`, which introduces relatively little overhead for recording statistics. This can be run with e.g. `$ python -m cProfile -o my_profiling_results.prof my_script.py`.

3. *Analyze Statistics*: This step is where the profiling results are considered, and slow functions are identified. Tools such as `tuna` or the built-in `pstats` are helpful for this.

Once an appropriate function/code segment has been selected, then that segment should be optimized.

[6] This process is also relatively efficient with developer resources, because it uses empirical evidence for the sections that are actually slow, instead of wasting time optimizing parts of the software that might not actually be slow.

A.3.2.2 Python Optimization

Python is a somewhat difficult programming language to optimize, because it is interpreted instead of natively compiled. So static compiler optimizations (such as turning on `gcc`'s `-O3` flags) are not available to Python.

Instead, users are left with attempting either algorithmic improvements, or compiling to native machine code in order to optimize a given segment.

Algorithmic improvements are generally easier, as the complexity is lower and it doesn't involve the use of external tools. Examples of algorithmic improvements include changing the type of sorting algorithm used, caching data to prevent iterating over an array several times, or using a data structure optimized for the operation that you are performing (i.e. dictionary for faster lookup vs a list).

Native compilation will be discussed in Appendix A.3.3.

A.3.3 Cython: Compilation from Python to Native Code

Python, as a flexible interpreted language, is able to easily bridge between different software interfaces, especially using tools such as `cython`. Cython is a python library that allows converting Python-native (or Python-like) code to an equivalent C program, and then compiling the C program. The C program/function can then be called transparently via Python. This allows CPU-bound tasks, such as floating point calculations, to happen as quickly as possible without the overhead of a Python interpreter. This technique can yield speedups of several hundred times, which can be enough to change a solution from infeasible to practically negligible.

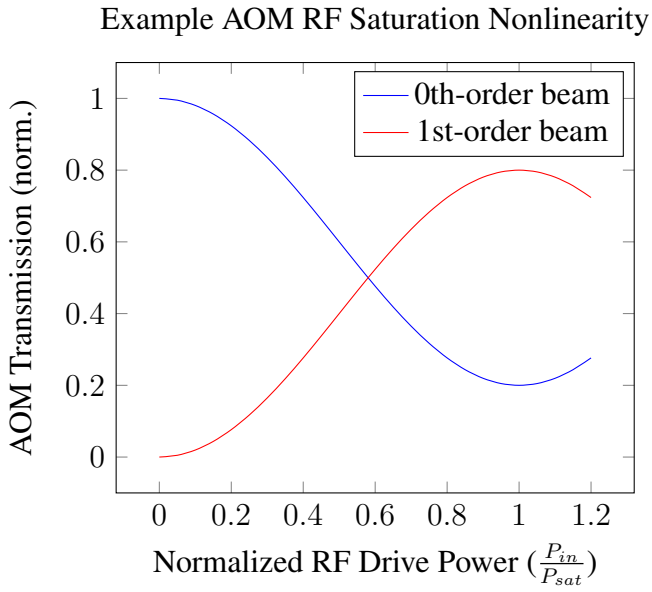


Figure A.1: **Example of AOM Transmission Saturation as a function of RF Drive Power.** The AOM transmission on the first-order beam (desired) saturates at $P_{in} = P_{sat}$. The x-axis is the normalized drive power (normalized so that 1.0 is the saturation point), and the y-axis is the amount of the total beam power diffracted into a given AOM beam order (normalized to total beam power). This AOM has diffraction efficiency of $80\,\%$.

To make the benefits of Cython more concrete, here is one example that we encountered when designing schedule modifications on top of the PulseCompiler (Chapter 6).

We perform our Raman operations by modulating laser beams using AOMs, which are themselves modulated by RF signals. The response of the AOMs to RF drive exhibits a nonlinear effect, where the transfer of power into the AOM-modulated beam saturates (illustrated in Fig. A.1). This effect manifests in diminishing returns of increased RF amplitude with respect to the rate of Rabi oscillations of an ion. We can correct for this nonlinearity by adjusting the amplitude of a waveform. However, the waveform amplitudes are not always static; the RFSoC supports amplitudes with cubic spline coefficients.

Thus, to compensate for the AOM nonlinearity, the amplitude cubic spline coeffi-

cients will need to be recalculated. Some testing determined that this could be done with an iterative fitting routine. The caveat is that this nonlinearity needs to be applied to *each set of cubic spline coefficients* of a waveform sequence. For an example sequence that might include $N_{segments} = 10000$, and initial implementations of this algorithm that took $\approx 500\,\mu\text{s}$ to execute per segment, nonlinearity correction could take $\approx 5\,\text{s}$ of the entire programming sequence.

Fortunately, this is a very isolated subroutine, which should be able to be heavily optimized. Using Cython, and various compiler optimizations, we were able to reduce the time per execution from $\approx 400\,\mu\text{s}$ per correction to $\approx 1\,\mu\text{s}$ per correction, yielding a $400\times$ speedup.

While implementing and optimizing a Cython version of all Python software is impractical[7], it does illustrate the potential of software optimizations in Python especially.

[7] Especially because this application was almost an ideal case, due to heavy use of raw number-crunching which needs minimal overhead.

Appendix B: Git and its Usage in Physics Experiments

B.1 Git Overview

Git is a software program to do version control of any set of folders, though it is most-used for software development projects. While it is not the only version control software [1] , it is the most popular and one of the most general. Git is responsible for tracking the state of a set of files (called a *repository*) over time, while also allowing changes to happen in parallel on different devices (PCs) and still be consistent.

B.1.1 How does Git work?

The most common data type used in Git is a "commit". A commit is a cryptographic hash of the state of the repository, along with some metadata information such as the commit author, a message about what has been committed, and the *previous* commit. Each commit is identified by a hash of its contents, so it is virtually guaranteed to point to a unique state of the repository and not have any collisions with other commits in the repository.

This data structure creates a *graph* data structure, where it is possible to trace and

[1] Other protocols like Subversion, Mercurial, and others exist, but are not commonly used in modern software development.

reconstruct the state of the Git-versioned files by tracking the original source of any changes. The sum of all commits and information about the source code is called a *repository*.

B.1.2 Git for Beginners

Commits can be grouped into a "branch". The idea of a branch is that it is a sequence of commits. A new branch can be easily created from a certain commit (or branch), which will allow commits to be added to *only* that branch without affecting the state of the previous commit. When work is finished on the temporary branch `B`, it can be *merged* into the main development branch `main`, which adds any changes from `B` into the `main` branch.

Git is typically interacted with using the `$ git` command-line tool. If GUI tools are used (e.g. `gitk, GitHub CLI, VS Code, PyCharm`, etc.), they are effectively a wrapper around the underlying `git` tool.

There are many introductory resources to Git that do a very good job of explaining how to use Git, but I will introduce some basic Git commands and their usage here that are especially useful. Knowing how to use these commands (Table B.1) can easily transfer to using the GUI tools, because the same concepts apply though they might be exposed differently. [2]

[2] My personal opinion is that it is better to learn using the command-line tools, because they offer better error messages and are more standardized than the GUI tools so it is easier to get support. Once basic competency in Git is obtained, it is useful to use GUI tools for intermediate operations such as looking at file history, or adding only sections (patches) of a file for commit.

Command	Short Description	Notes
`$ git clone` *`url`*	Download a copy of a Git repository	This creates a complete copy of the Git repository stored online.
`$ git pull`	Update with any remote changes	Update local repository with any changes on the online repository, then update the current branch with any newer commits.
`$ git push`	Add any new commits for current branch to the online repository	Pushes (sends) any new commits on the current branch to the corresponding branch on the online repository. The `--force` flag can be added after amending a commit, but should **only** be used if a minor fix was made to a **development** (not stable) branch.
`$ git checkout` *`branch-name`*	Switch to a different branch.	Changes all local files to how they were on the latest commit on the local branch *branch-name*.
`$ git checkout --branch` *`new-branch`*	Create new branch *new-branch* from current commit.	Creates a new branch that commits can be added to. The start of the branch is at the current commit. The branch should be pushed to the cloud with `$ git push --upstream origin` *`new-branch`*.
`$ git branch --delete` *`branch-name`*	Delete local copy of branch	This command will error if the branch has not been pushed to the remote or merged into the primary branch.
`$ git remote get-url` *`remote-name`*	Print URL of the remote repository that this will pull from/push to	Common values for *remote-name* are `origin` and `upstream`
`$ git status`	Print out any changed or staged files in the current local repository	Should always be run & checked before committing to make sure that only the desired files will be committed.

continues on next page

Command	Short Description	Notes
`$ git restore --staged file`	Unstage the selected file	Prevents the selected file from being committed with the commit that is currently being prepared.
`$ git restore file`	Undo **ALL** changes to the local file(s).	This essentially undoes any local changes since the last commit. Can be used with *file* = `./` to essentially restore to the last commit.
`$ git add file`	Adds *file* to the staging area	Files in staging area will be committed when `$ git commit` is run
`$ git commit`	Create a new commit on the current branch with any changes specified in the index.	I like to use the `-m "Commit message"` flag from the command line for short commit messages, as it will commit immediately with the given message.
`$ git commit --amend`	Modify the last commit (message or change files)	This should only be used on the LAST commit (usually). I mostly use this if I forgot to add part of a file to the commit or adjust the commit message.
`$ git merge branch-name`	Merge commits from *branch-name* into current branch	Merges the git history of two branches, typically by adding any new commits from *branch-name* to the current branch.
`$ git stash push/pop`	Temporarily save (recover) any changes to Git-versioned files	This is useful when making changes to a branch, but you temporarily want to switch to a different branch. *push* will temporarily save the changes, which can be restored with *pop*.
`$ git log`	Look at commit history	Useful for looking at what changes have recently been made to a branch.

continues on next page

Command	Short Description	Notes

Table B.1: **Common Git Command Reference.** These are command-line commands that can be used with the standard Git command-line interface tool. `$ git rebase` is a powerful tool that can fix many mistakes, but requires **extreme caution** when using. Similar caution should be used when using the `--force/-f` version of any commands, as either `$ git rebase` or `--force` can permanently wreck the state of a repository. The full documentation for `git` can be found at [29]. `$ git filter-repo` is another useful tool for advanced git operations, such as separating folders out of a repository while preserving history [189].

B.2 Git in Physics Experiment

While it may appear strange to discuss Git in the context of a physics experiment, there is a mismatch in expectations and best practices for software best practices when compared with the needs of a physics experiment that are worth examining.

Git is generally very good at managing state across several computers, but it has a certain jargon that is not immediately clear, or can conflict with expected naming conventions from physics. It is also a semi-complicated software tool that many physicists have but passing familiarity with. Git generally expects a structured software development process, which few physicists have any experience with. [3] Finally, there is a steep learning curve associated with Git, especially as its use is adapted for each particular group using it, and this learning curve can be off-putting for non-specialists.

[3] Git works well for single-user, single-branch situations, but such use is almost trivial and will not be treated here.

B.2.1 Requirements for a Physics Experiment

Physics experiments have several different characteristics that separate them from typical software engineering projects. These include:

- *Hardware is the single-source-of truth*: The software to control a physics experiment must reflect the state of the physical experiment & its hardware. This will necessarily slow down the pace of software development, as it must be kept in sync with the physical hardware which is generally slow to change. Further, any software changes must be tested to ensure compatibility with the physical experiment.

- *Jargon*: Both physics groups and software engineering tend to have field-specific jargon, which creates a language barrier between the two fields. Concepts such as *versions* and *releases* don't really apply to a single, rolling-updated piece of hardware.

- *Parallel development*: Some physics experiment teams comprise about half a dozen people. These people will necessarily diversify and work on separate projects, all of which will have their own development trajectory.

- *Need for offline development*: To continue efficient development, while being constrained to a single piece of hardware, offline development should be considered, especially on a team of more than a few people. This is an enabler of parallel development, though conceptually different. Offline development is the idea that basic functionality can be validated without using physical hardware, which can make better use of limited test time and resources.

- *Lack of training*: Training in the need for and use of version control systems is typically reserved for engineering and computer science educations. So the utility and benefits of version control are not considered by an average physicist.

- *Complex dependencies*: Physicists have a tendency to pull in various pre-built parts to their experiments, whether hardware or software. This can create a complex web of software dependencies. However, as the complexity and number of dependencies increases, the risk of incompatibility increases as well as the overhead of keeping these packages up-to-date.

Considering these requirements, version control for a physics experiment should be strongly considered. In addition to (at least partially) addressing most of the requirements above, it can also provide backups of critical software, minimize the cost of mistakes[4], and enable parallel and remote development that is not explicitly tied to a specific computer or hardware experiment.

B.2.2 Suggested Git Workflow

Given these constraints, we have developed best working practices for Git that attempt to balance the need for offline software development with the single-source-of-truth nature of a physical, ever-changing experimental hardware setup.

First, we begin by declaring two primary branches for our Git repository. These correspond to logical points of the experimental setup, and use plain physicist language instead of software terms. [5] The branches are:

[4]For example, consider a value that was mistakenly changed. Version control allows tracking down that particular change, as well as to roll back to a time before the erroneous change was made.

[5]For many years, across all of software engineering, the default Git branch name was `master`. Besides

- `experiment-LIVE`: This branch is closest to a traditional `master` or `main` branch, and corresponds to the current state of the experiment, and is closely matched with the hardware. The concept of this branch is that if a software development effort goes wrong, this branch should always be working. Further, if something about the state of the hardware changes, then this branch is *immediately* updated accordingly. Some examples of hardware changes that might need software changes would include:
  - New hardware added
  - Signals remapped
  - Calibration value or limits changed

  This branch should **only** be committed to from the computer(s) that are directly connected to the machine, and should be regularly (roughly every commit) pushed to the cloud for backup and to ensure that everyone is using the same software version.

- `experiment-stable`: This branch will be updated (`merge`d) from `experiment-LIVE` whenever an important milestone is reached, such as data for a paper is taken. The rationale is that this is an easy point to remember, and the software state at this point might either need to be referenced or recovered for future paper revisions.

However, neither of these branches allow for long-term software development, or

potential slavery connotations, this name is simply confusing to people who are not well-versed in software development or Git, so I highly recommend changing this branch name at the earliest convenience. I recommend changing *master* to either something that denotes priority such as `main`, or a name that is more specific to your specific use case, such as `experiment-LIVE`.

for team members to work offline on the same repository. As such, the following procedure is recommended for new feature software development. [6]

1. *Create development branch*. The source branch should be `experiment-LIVE` to get the most up-to-date changes[7]. I typically like to use `dev-`*`feature-name`* or `feature/`*`feature-name`* for the branch name.

2. *Add commits* by developing your new feature.

3. *Push commits* with `$ git push`

4. *Create Merge/Pull Request*. A Merge Request (MR) or Pull Request (PR) should describe the motivation and any changes that are being made, as well as any remaining TODO items, including items to test. [8] The target branch (branch that the feature is being merged into) should be `experiment-LIVE`.

5. *Checkout branch on experiment PC*

6. *Merge new commits from* `experiment-LIVE`. This ensures that your software is compatible with the latest machine state snapshot.

7. *Test new feature*. Add and push new commits as needed.

[6] One possible permutation of this scheme is to batch together several changes/features into a testing branch, which could be named e.g. `TESTING-experiment-LIVE`. This allows parallelizing development, and having an intermediate "staging" area in between having finished the software and made sure that it works as far as you can tell, and actually reaching the stable `experiment-LIVE` branch. One caution with this approach is that it should only be used for small batches (≈ 5) changes. The risk with too many changes is that they will be interdependent, which can slow troubleshooting of any bugs that arise. This can be an acceptable tradeoff when testing time for software on the machine is limited relative to the number of new features that are being developed.

[7] Make sure to pull any changes into your local branch before creating the new branch.

[8] Because you might not be able to test the MR/PR for a while.

8. *Approve Pull Request & Merge*. If performed through a web interface such as GitLab/GitHub, these steps can be performed together.

Appendix C: Nix: Deterministic, Reproducible Software

C.1 Overview

One problem that arises when control of our quantum computer is distributed across several classical computers is keeping the software running on those computers synchronized. Source-control tools such as Git [29] provide part of the solution, but they only are capable of managing the source code of our experiment control software, and cannot guarantee that the dependencies of that software (e.g. Python [28]) are standardized across multiple computers.

Imagine that the control system is written in Python, and installed on a computer. At the time of initial installation, the latest version of Python is 3.6, and all the software is designed to work with that Python version. However, a year later, a new computer is added to the distributed control system to perform some other function, but it is installed with Python 3.7. Suddenly, the software that was functioning perfectly on Python 3.6 is no longer working, or has subtle bugs.

To resolve this issue, the goal is to ensure that all computers are running the exact same versions of all dependent software packages: i.e. Python, ARTIQ, Numpy, Scipy, etc. There are some commonly used tools that exist for capturing this "software environment", such as Conda [187], pipenv [188], venv [190], poetry [191], etc. However, these

tend to collectively suffer from some flaws:

- Reliance on external package providers to host the pre-built packages (called wheels in Python).
- Cannot specify arbitrary environment variables, which is useful for reproducibly changing settings.
- Restricted to Python software only (virtualenv, pipenv, poetry).
- Cannot incorporate patches to the source code to fix issues on your own timetable (all).
- Slow dependency solvers (Conda).
- Software build cannot be fully replicated from source code.
- Upgrades to software removes the previous software versions, and it is difficult to recover the previous software if the upgrade causes bugs.

The issue is that it is difficult to capture the entire state of a software system: to fully reproduce the software environment used, you theoretically need to use the exact same version of every piece of software in the entire toolchain, as well as the build environment that produced all the software pieces.

In other words, in order to reproduce a software environment including Python, you also need to know the exact version of the software that compiled CPython (i.e. Python's source code) to the Python binary run on your local machine. You also must know the settings (typically environment variables) used in the Python compilation.

C.2 What is Nix?

Nix [30, 192] is a language that is designed to purely functionally describe software packages. In this context, *purely functional* refers to a computer science term [193] meaning that some computation is based purely on a set of inputs, and will **always** generate the same set of outputs given the same inputs, while producing no side effects. This concept is similarly seen in languages such as Haskell and Julia.

When viewed in the Nix perspective, the software build process itself becomes a function: given a set of inputs (source code, a package's dependencies, and the instructions on how to build a software package), it will always produce the same output software. This process is independent of which computer it is built on, ensuring that the software package is fully reproducible. [1]

Thus, the text of a package in Nix (a `*.nix` file) is how to build the software package (in actuality, any output file), and the output is the package itself, and will be herein called the "package description".

C.2.1 Cryptographic Hashes & Nix Store

To capture a software package's dependencies, Nix employs cryptographic hashes [192]. Cryptographic hashes are a function that converts an arbitrarily-large input into a smaller

[1] For Nix to work properly, it expects a Unix-like operating system to run on. Examples include Linux and MacOS. As such, Nix is not available directly on Windows. However, using WSL2 [194], a Linux Virtual Machine (VM) can be easily installed, and Nix can be run in the virtual machine. This method works very smoothly, but it is not the most performant option for building or running Linux software, so it is not universally recommended. WSL virtualization also does not easily support GUI software, so WSL + Nix should mostly be used for text-only software, or in an IDE that natively supports it such as Visual Studio Code.

number, and typically have the properties:

- Quick to compute.
- Unlikely to collide.[2]
- Cryptographically secure one-way functions.[3]
- Small input changes create large output changes, making changes to the input seem uncorrelated to the output.

By using cryptographic hashes, Nix is able to ensure reproducibility: if the inputs to a package description function change (either due to the build instructions themselves changing, or dependencies changing), the hash of the package description will change, and Nix will know that the package is now out-of-date and needs rebuilt.

When generating a package, Nix goes through two steps: evaluation and instantiation. At evaluation, all package dependencies are frozen and hashed, and a "derivation" (i.e. all the dependencies and instructions needed to generate the software output) is created. At instantiation, all derivations required for the chosen software are followed to create the desired software. The results of evaluation and instantiation for each package and its source are cached in the "Nix store", at unique files or folders similar to `/nix/store/`**`HASH-PACKAGE_NAME-PACKAGE_VERSION`** for software outputs, and `/nix/store/`**`DERIVATION_HASH-PACKAGE_NAME-PACKAGE_VERSION.`**`drv` for derivations. The naïve approach to instantiation is to build every required software package in the dependency tree on your local computer. However, this is simply unnec-

[2]Meaning that it is very unlikely or difficult to find two inputs will provide the same output
[3]Meaning that it is difficult to do the reverse transform from output into the original input

essary and inefficient for most use cases: most people have no need or desire to rebuild core software packages such as the *gcc* compiler from source. Instead, Nix can substitute in binary caches from trusted sources for any given output, which it does by searching remote computers for the cryptographic hash corresponding to a derivation's output. Once an item is added to the Nix store, it is immutable. If the package description or source changes, a new derivation and output are generated.

C.3 Nix Package Hierarchy

Nix packages tend to employ a hierarchy of package description sources (ordered from most general/public to most specific/private):

1. *Nixpkgs*: the primary package description source for packages described in Nix. `Nixpkgs` is a single Git repository hosted on GitHub [195]. All package descriptions defined in `Nixpkgs` are built approximately daily, and are cached on `https://cache.nixos.org`

2. *Shared Upstream Repository*: A single repository where a community of users make their package descriptions public, but generally less-reliable than `Nixpkgs`. Examples include the Nix User Repository (NUR), and ARTIQ's Nix Scripts repository [196].

3. *Personal Repositories*: You can create multiple projects, each with their own Nix package description, and use them as a dependency for a separate project.

   The Nix package descriptions can be sourced from any file location, though they

are typically from a public or private Git repository, or a publicly-accessible file on the Internet.

C.4 Using Nix

Once a Nix environment is designed, written, and builds without errors, it is straightforward for end-users to actually use. This next section will describe how to use a Nix environment practically, particularly in the context of ARTIQ. This section is written primarily for Nix without using the new flakes features, so it is primarily applicable to ARTIQ $<= v6$. However, many of the key concepts carry over, even if their actual naming or format is slightly different.

C.4.1 Example Nix Environment

Let us consider an example Nix environment from the ARTIQ installation guide [25].

```
let
  # pkgs contains the NixOS package collection.
  # ARTIQ depends on some of them, and
  # you may want some additional packages from there.
  pkgs = import <nixpkgs> {};
  artiq-full = import <artiq-full> { inherit pkgs; };
in
  pkgs.mkShell {
    buildInputs = [
      (pkgs.python3.withPackages(ps: [
        # List desired Python packages here.

        # You probably want these two.
        artiq-full.artiq
        artiq-full.artiq-comtools

        # You need a board support package if and only if you intend to flash
        # a board (those packages contain only board firmware).
        # The lines below are only examples, you need to select appropriate
        # packages for your boards.
```

```
21          #artiq-full.artiq-board-kc705-nist_clock
22          #artiq-full.artiq-board-kasli-wipm
23          #ps.paramiko  # needed if and only if flashing boards remotely
    (artiq_flash -H)
24
25          # The NixOS package collection contains many other packages
    that you may find
26          # interesting for your research. Here are some examples:
27          #ps.pandas
28          #ps.numpy
29          #ps.scipy
30          #ps.numba
31          #(ps.matplotlib.override { enableQt = true; })
32          #ps.bokeh
33          #ps.cirq
34          #ps.qiskit
35        ]))
36
37        # List desired non-Python packages here
38        #artiq-full.openocd  # needed if and only if flashing boards
39        # Other potentially interesting packages from the NixOS package
    collection:
40        #pkgs.gtkwave
41        #pkgs.spyder
42        #pkgs.R
43        #pkgs.julia
44      ];
45  }
```

Listing C.1: Nix ARTIQ Environment example. Filename: `shell.nix`

Let us consider this Nix package description Listing C.1 in sections.

Block Name	Block Code	Description	Line Numbers
Let Block	`let ... in ...`	Variable definition section.	1, 7
Comment	`# ...`	Either text comment or disabled package.	2, etc
Top-level Object	`pkgs.mkShell{}`	Create a Nix shell environment with the given set of packages available. This is actually a function, and will produce a derivation given the set of inputs (e.g. `buildInputs`)	8, 45

continues on next page

Block Name	Block Code	Description	Line Numbers
Base Package set definition	`pkgs = import <nixpkgs> {}`	Defines the "base" set of packages available for use. This is a particular instance/commit of the large `nixpkgs` repository. Basic packages like compilers, Python, etc are available from here.	5
Variable definition	`artiq-full = import <artiq-full> { inherit pkgs; }`	Defines a set of packages to be available, which are defined in a library path `<artiq-full>`. These ARTIQ packages are built against the base package set `pkgs`.	6
Set definition	`{ a = "b";}`	Defines an *attribute set*, which is a mapping from a name (e.g. `a`) to a value (e.g. `"b"`)	5, 6, 8, etc
List definition	`[ a ]`	Define a list	10 → 35, etc
Set Attribute	`a.b`	Use a given attribute from the set `pkgs`	8
Available Packages	`buildInputs = [...];`	Defines the packages available in the `mkShell`, i.e. the packages that are inputs to "building" the shell.	9
Python Package Environment	`python3 .withPackages`	Python requires that all available packages for a given *virtual environment* be placed in the same directory. This function defines a Python environment that *only* has the given packages (and their dependencies) available.	10
Python Package Usage	`ps.numpy`	Add the python package `numpy` to the Python environment. `artiq-full.artiq` is also valid, but it comes from a different package set. [4].	28
Non-Python Package Usage	`artiq-full.openocd`	Adds non-Python package `openocd` to the shell environment. Currently commented out.	38

continues on next page

[4]Note: `numpy` is currently commented out, but will still be included by default because it is a dependency of ARTIQ.

Block Name	Block Code	Description	Line Numbers

Table C.1: **Breakdown of Nix environment description sections (from Listing C.1).** The columns indicate, respectively, the type of language construct, an example of the language construct, and explanation of the construct, and some line numbers that demonstrate that construct. Other resources on learning Nix can be found at `https://nixos.org/guides/nix-pills/`, `https://nixos.org/learn.html`, `https://nix.dev/tutorials/`, and `https://nixos.org/manual/nix/stable/introduction.html`, as well as many examples in the main Nix repositories [195].

For the sake of reference, here is a roughly equivalent description using the Nix flake format from `https://m-labs.hk/artiq/manual/installing.html#installing-via-nix-linux` (Listing C.2):

```
1     {
2         inputs.artiq.url = "git+https://github.com/m-labs/artiq.git?ref
    =release-7";
3         inputs.extrapkg.url = "git+https://git.m-labs.hk/M-Labs/artiq-
    extrapkg.git?ref=release-7";
4         inputs.extrapkg.inputs.artiq.follows = "artiq";
5         outputs = { self, artiq, extrapkg }:
6           let
7             pkgs = artiq.inputs.nixpkgs.legacyPackages.x86_64-linux;
8             aqmain = artiq.packages.x86_64-linux;
9             aqextra = extrapkg.packages.x86_64-linux;
10          in {
11            defaultPackage.x86_64-linux = pkgs.buildEnv {
12              name = "artiq-env";
13              paths = [
14                # ========================================
15                # EDIT BELOW
16                # ========================================
17                (pkgs.python3.withPackages(ps: [
18                  # List desired Python packages here.
19                  aqmain.artiq
20                  #ps.paramiko  # needed if and only if flashing boards
     remotely (artiq_flash -H)
21                  #aqextra.flake8-artiq
22
23                  # The NixOS package collection contains many other
    packages that you may find
24                  # interesting. Here are some examples:
```

```
25                    #ps.pandas
26                    #ps.numpy
27                    #ps.scipy
28                    #ps.numba
29                    #ps.matplotlib
30                    # or if you need Qt (will recompile):
31                    #(ps.matplotlib.override { enableQt = true; })
32                    #ps.bokeh
33                    #ps.cirq
34                    #ps.qiskit
35                  ]))
36                  #aqextra.korad_ka3005p
37                  #aqextra.novatech409b
38                  # List desired non-Python packages here
39                  #aqmain.openocd-bscanspi  # needed if and only if
      flashing boards
40                  # Other potentially interesting packages from the NixOS
       package collection:
41                  #pkgs.gtkwave
42                  #pkgs.spyder
43                  #pkgs.R
44                  #pkgs.julia
45                  # ========================================
46                  # EDIT ABOVE
47                  # ========================================
48                ];
49              };
50            };
51        }
```

Listing C.2: ARTIQ 7 shell environment definition using Nix flake format. Filename: `flake.nix`

The primary difference here is that the exact branch & Git repository is specified for the ARTIQ shell. There are several small naming differences, such as the shell description (`pkgs.mkShell`), now at Line 11, which is defined as `defaultPackage.x86_64-linux = pkgs.buildEnv { ... };`. Also, several ARTIQ packages that had previously been part of `artiq-full` have now been moved to `aqextra`. The primary advantage that this has, other than compatibility with ARTIQ v7, is that this code is fully reproducible by simply changing the URLs in Listing C.2 Line 2, 3 to use a specific commit instead of a branch name (i.e. `release-7`).

Once you have defined the needed inputs & binary package caches as specified in

the ARTIQ manual, you can now create the shell environment on your local PC. With the ARTIQ 6 version (Listing C.1), this is as simple as executing in your command-line terminal `$ nix-shell /path/to/shell.nix`. For the ARTIQ 7 version, you can execute `$ nix shell /path/to/flake.nix`. [5]

C.4.2 Full Reproducibility

One of the primary benefits that Nix provides is that the environment with all of the packages available is fully reproducible on any other PC. This means that if your computer suffers a catastrophic meltdown, if the Internet (or the package repositories such as `PyPi`, see Appendix A.2.2) are not available (or don't support your particular system), that you can generate the entire set of packages (i.e. environment) from scratch if given enough time.

This also provides a few other benefits:

- *Scalability*: Once a Nix environment is defined once, it is fully specified, and can be replicated exactly on any other computer. This largely shifts the problem of setting up software from the end-user to the developer, and minimizes the need for long lists of installation instructions, which can be somewhat fragile (or non-repeatable).

- *Environment versioning*: A Nix software environment is just a set of text files, so the environment can be easily versioned with a program like Git (Appendix B). Using Git with Nix allows you to both fully define the software that you have available, as well as to make incremental & easily rolled-back changes to the software. It

[5] Nix has a somewhat-unique feature that it can use a URL as the source for a `*.nix`, instead of only a local on-PC path.

also allows for separate software changes on separate branches, with the software environment updating on both. One example usage of this is developing a new feature that requires some new software dependency. Until the new feature branch is merged into the main branch, there is no risk of cross-contamination or undesired side effects. This allows testing the new feature completely in isolation, instead of mixing new & pre-existing software.

- *Reproducible science & data analysis*: Across various academic disciplines, there has been a push for being able to reproduce the data analysis in papers [197]. Generally, these efforts stop at sharing the source code for the analysis code, but that is missing a lot of potential context. For instance, suppose that there is a bug in your particular version of a Python library that is resulting in a mis-ordering of data. Without information about the complete software environment, little details such as these would remain mysteries and be extremely hard to sort out, and could reflect poorly on the academic whose results cannot be reproduced. With Nix, the entire software stack can be described, with the side effect that it is actually *easier* for others to reproduce your exact data analysis environment.

- *Support for patches/bugfixes*: In many physics lab environments, there is a general resistance to changing any little detail, because the systems that are being studied are highly complex and non-robust to outside errors. However, the software that is run is typically also research-grade software, and not always very stable. As such, sometimes there are critical *small* bugs that need to be fixed, but those do not merit a large system upgrade to the latest version of software. An example would be Bug

B that is found in v1.0.0 of software S. After a bit of research, you find that the bug has been fixed in S v2.0.0, but there are many other changes that could potentially cause compatibility issues. Without Nix, the easiest solution would be to upgrade S to v2.0.0, and just handle any compatibility issues that arise. With Nix, the easiest solution is to just patch the source code with the bugfix, and then let Nix rebuild all the necessary software. In this scheme, since you manually inspected the patch and didn't do a major software upgrade, you can minimize the risk of software issues while gaining the ability to fix the relevant bug. An example of doing this can be seen in Listing C.3, which simultaneously overrides the version of `sipyco`, adds a patch from a Git commit, and then also applies a custom text replacement. [6]

```
sipycoOverride = sipyco.overridePythonAttrs(oldAttrs: rec {
  version = "1.4";
  src = pkgs.fetchFromGitHub {
      owner = "m-labs";
      repo = "sipyco";
      rev = "v${version}";
      sha256 = "sha256:0
    vw5z2m59ksc3kz4dbfwczc0k0hlklrvjcn6dc42b9kmknv1cimh";
  };
  patches = [
    # this patch is just for demonstration purposes
    (fetchpatch {
        url = "https://github.com/m-labs/sipyco/commit/
    ca911b0ac02aa466ab4f20c4dd3300cfbf577b20.patch";
        name = "pr-11-fix-server-disconnects.patch";
        sha256 = lib.fakeSha256;
    })
  ];
  postPatch = ''
    substituteInPlace sipyco/asyncio_tools.py -{}-replace "limit
    =4*1024*1024" "limit=64*1024*1024"
  '';
});
```

Listing C.3: Example of Patching a Nix package to fix a software bug

[6] Note that these overrides/patches are all independent, and any combination could be used as desired. This syntax is also only valid for Python packages. The syntax is a little more complex for Python than other packages. More information can be found in the Nix manual (`https://nixos.org/manual/nix/stable/introduction.html`).

C.4.3 Other Recommended Nix Tools

While Nix itself is feature-complete, there are several external tools that make working with Nix in projects very easy. The following is a list of my recommended projects.

- *Direnv* (`https://direnv.net/`): This tool will automatically change your shell environment variables when navigating to a particular directory. While this tool has many features, in the context of Nix & ARTIQ, the most useful feature is automatically loading a Nix shell environment when entering a given directory. In other words, when you enter a specific software directory, all of your ARTIQ command-line tools are now available because the Nix environment is automatically loaded. [7]

- *Nix Flakes*: An example of this was shown in Listing C.2. Nix flakes are a newer built-in Nix feature to allow more modularity between package repositories, as well as version-control those package dependencies. This topic is more complicated than can be fully explained here. However, readers who are new to Nix should plan to just start with learning Nix flakes, because they are becoming the *de facto* standard in the community.

- *Niv* (`https://github.com/nmattia/niv`): Niv is responsible for tracking upstream software dependencies, and making their source code easily available in Nix. It also allows quickly updating/changing the current revision that the upstream packages are at. There is an easy command-line tool for updating dependencies:

[7] We have had some issues with direnv not updating the environment when switching between branches with different Nix descriptions. This theoretically could be solved with a program like [198].

adding & updating a dependency that is on GitHub can be as easy as: `$ niv add m-labs:artiq && niv update artiq --branch release-6`. Most of the functionality of Niv was superseded by Nix Flakes when they were released, but it is still useful when legacy code has not been converted to use Nix flakes. This tool has been deprecated due to the maintainer moving on, so this should not be used in new projects.

- *NixOS*: NixOS is similar to Docker, in that it can describe how a computer should be set up from scratch, along with all configuration that it needs. However, NixOS is designed to work on bare-metal PCs as well as VMs, while Docker is primarily meant to run in a virtualization environment. While the concept is interesting, especially the ability to have your entire PC set-up using a single text file, my personal experience is that the learning curve is steep for laboratory environments, and generally not worth the effort. However, if complete consistency in the PC's available software is a critical factor, then NixOS is one of the best available options.

A reference of Nix commands can be found at `https://nixos.org/manual/nix/stable/command-ref/experimental-commands.html` and `https://nixos.wiki/wiki/Cheatsheet`.

C.5 Nix usage in EURIQA

When we upgraded to ARTIQ 5, the EURIQA team decided to invest in working with Nix. This meant that all of our Python dependencies needed to be converted to using Nix, especially packages such as PulseCompiler, Qiskit, Cirq, etc.

To aid this, we created a public repository for several of these packages, available at `https://github.com/drewrisinger/nur-packages`. These packages, along with some private packages and the public ARTIQ packages, are incorporated as dependencies of the EURIQA experimental code. One of the ways that we manage these dependencies is using `niv`, a Nix tool which simplifies dependency addition, tracking, and updating.

Bibliography

[1] Klaus Mølmer and Anders Sørensen. Multiparticle Entanglement of Hot Trapped Ions. *Physical Review Letters*, 82(9):1835–1838, March 1999.

[2] Anders Sørensen and Klaus Mølmer. Quantum Computation with Ions in Thermal Motion. *Physical Review Letters*, 82(9):1971–1974, March 1999.

[3] Anders Sørensen and Klaus Mølmer. Entanglement and quantum computation with ions in thermal motion. *Physical Review A*, 62(2):022311, July 2000. Publisher: American Physical Society.

[4] Norman S. Nise. *Control systems engineering*. Wiley, Hoboken, NJ, 6th ed edition, 2011.

[5] John L. Hennessy, David A. Patterson, and Krste Asanović. *Computer architecture: a quantitative approach*. Morgan Kaufmann/Elsevier, Waltham, MA, 5th edition, 2012. OCLC: ocn755102367.

[6] David A. Patterson and John L. Hennessy. *Computer organization and design: the hardware/software interface*. The Morgan Kaufmann series in computer architecture and design. Elsevier/Morgan Kaufmann, Amsterdam ; Boston, 5th edition, 2014. OCLC: ocn859555917.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Texts in theoretical computer science. Springer, Berlin, 2010.

[8] Ning Wang, Naiqian Zhang, and Maohua Wang. Wireless sensors in agriculture and food industry—Recent development and future perspective. *Computers and Electronics in Agriculture*, 50(1):1–14, January 2006.

[9] Maurice V. Wilkes. Computers Then and Now. *Journal of the ACM*, 15(1):1–7, January 1968.

[10] Steve Furber. Microprocessors: the engines of the digital age. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2199):20160893, March 2017. Publisher: Royal Society.

[11] M. Morris Mano and Michael D. Ciletti. *Digital design: with a introduction to the verilog hdl*. Pearson Prentice Hall, Upper Saddle River, NJ, 5th ed edition, 2013.

[12] R.R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, June 1997. Conference Name: IEEE Spectrum.

[13] Shankar Krishnan, Suresh V. Garimella, Gregory M. Chrysler, and Ravi V. Mahajan. Towards a Thermal Moore's Law. *IEEE Transactions on Advanced Packaging*, 30(3):462–474, August 2007. Conference Name: IEEE Transactions on Advanced Packaging.

[14] Quentin P. Herr, Anna Y. Herr, Oliver T. Oberg, and Alexander G. Ioannidis. Ultra-low-power superconductor logic. *Journal of Applied Physics*, 109(10):103903, May 2011. Publisher: American Institute of Physics.

[15] Thomas M. Conte, Erik P. DeBenedictis, Paolo A. Gargini, and Elie Track. Rebooting Computing: The Road Ahead. *Computer*, 50(1):20–29, January 2017. Conference Name: Computer.

[16] A.A. Sawchuk and T.C. Strand. Digital optical computing. *Proceedings of the IEEE*, 72(7):758–779, July 1984. Conference Name: Proceedings of the IEEE.

[17] Phillip R. Kaye, Raymond Laflamme, and Michele Mosca. *An introduction to quantum computing*. Oxford University Press, Oxford, 2007.

[18] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge ; New York, 10th anniversary edition, 2010.

[19] C. Monroe, W. C. Campbell, L.-M. Duan, Z.-X. Gong, A. V. Gorshkov, P. Hess, R. Islam, K. Kim, G. Pagano, P. Richerme, C. Senko, and N. Y. Yao. Programmable Quantum Simulations of Spin Systems with Trapped Ions. *arXiv:1912.07845 [cond-mat, physics:quant-ph]*, December 2019. arXiv: 1912.07845.

[20] Alán Aspuru-Guzik, Anthony D. Dutoi, Peter J. Love, and Martin Head-Gordon. Simulated Quantum Computation of Molecular Energies. *Science*, 309(5741):1704–1707, September 2005. Publisher: American Association for the Advancement of Science.

[21] B. P. Lanyon, J. D. Whitfield, G. G. Gillett, M. E. Goggin, M. P. Almeida, I. Kassal, J. D. Biamonte, M. Mohseni, B. J. Powell, M. Barbieri, A. Aspuru-Guzik, and A. G. White. Towards quantum chemistry on a quantum computer. *Nature Chemistry*, 2(2):106–111, February 2010. Number: 2 Publisher: Nature Publishing Group.

[22] I.M. Georgescu, S. Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153–185, March 2014. Publisher: American Physical Society.

[23] Juan Miguel Arrazola, Alain Delgado, Bhaskar Roy Bardhan, and Seth Lloyd. Quantum-inspired algorithms in practice. *Quantum*, 4:307, August 2020. arXiv:1905.10415 [quant-ph].

[24] A. Narayanan and M. Moore. Quantum-inspired genetic algorithms. In *Proceedings of IEEE International Conference on Evolutionary Computation*, pages 61–66, May 1996.

[25] Sebastian Bourdeauducq. ARTIQ, 2016. https://m-labs.hk/artiq/manual-legacy/.

[26] David C. McKay, Thomas Alexander, Luciano Bello, Michael J. Biercuk, Lev Bishop, Jiayin Chen, Jerry M. Chow, Antonio D. Córcoles, Daniel Egger, Stefan Filipp, Juan Gomez, Michael Hush, Ali Javadi-Abhari, Diego Moreda, Paul Nation, Brent Paulovicks, Erick Winston, Christopher J. Wood, James Wootton, and Jay M. Gambetta. Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments. *arXiv:1809.03452 [quant-ph]*, September 2018. arXiv: 1809.03452.

[27] Thomas Alexander, Naoki Kanazawa, Daniel J. Egger, Lauren Capelluto, Christopher J. Wood, Ali Javadi-Abhari, and David McKay. Qiskit Pulse: Programming Quantum Computers Through the Cloud with Pulses. *arXiv:2004.06755 [quant-ph]*, April 2020. arXiv: 2004.06755.

[28] Guido Van Rossum. Python, 1989. https://www.python.org.

[29] Linus Torvalds. Git. https://git-scm.com/. Manual: https://git-scm.com/docs/user-manual.

[30] Eelco Dolstra. Nix. https://nixos.org/.

[31] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019.

[32] Edwin Pednault, John A. Gunnels, Giacomo Nannicini, Lior Horesh, and Robert Wisnieff. Leveraging Secondary Storage to Simulate Deep 54-qubit Sycamore Circuits, October 2019. arXiv:1910.09534 [quant-ph].

[33] David P. DiVincenzo. The Physical Implementation of Quantum Computation. *Fortschritte der Physik*, 48(9-11):771–783, 2000. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E.

[34] Erwin Schrödinger. Discussion of Probability Relations between Separated Systems. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4):555–563, October 1935.

[35] W. K. Wootters and W. H. Zurek. A single quantum cannot be cloned. *Nature*, 299(5886):802–803, October 1982. Number: 5886 Publisher: Nature Publishing Group.

[36] Howard Barnum, Carlton M. Caves, Christopher A. Fuchs, Richard Jozsa, and Benjamin Schumacher. Noncommuting Mixed States Cannot Be Broadcast. *Physical Review Letters*, 76(15):2818–2821, April 1996. Publisher: American Physical Society.

[37] V. Bužek and M. Hillery. Quantum copying: Beyond the no-cloning theorem. *Physical Review A*, 54(3):1844–1852, September 1996. Publisher: American Physical Society.

[38] Tommaso Toffoli. Reversible computing. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, Lecture Notes in Computer Science, pages 632–644, Berlin, Heidelberg, 1980. Springer.

[39] Laird Egan, Dripto M. Debroy, Crystal Noel, Andrew Risinger, Daiwei Zhu, Debopriyo Biswas, Michael Newman, Muyuan Li, Kenneth R. Brown, Marko Cetina, and Christopher Monroe. Fault-tolerant control of an error-corrected qubit. *Nature*, 598(7880):281–286, October 2021.

[40] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, November 1995. Publisher: American Physical Society.

[41] Peter W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Review*, 41(2):303–332, January 1999. Publisher: Society for Industrial and Applied Mathematics.

[42] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, July 1996. Association for Computing Machinery.

[43] Daiwei Zhu, Gregory D. Kahanamoku-Meyer, Laura Lewis, Crystal Noel, Or Katz, Bahaa Harraz, Qingfeng Wang, Andrew Risinger, Lei Feng, Debopriyo Biswas, Laird Egan, Alexandru Gheorghiu, Yunseong Nam, Thomas Vidick, Umesh Vazirani, Norman Y. Yao, Marko Cetina, and Christopher Monroe. Interactive Protocols for Classically-Verifiable Quantum Advantage, June 2022. arXiv:2112.05156 [cond-mat, physics:quant-ph].

[44] R. C. Bialczak, M. Ansmann, M. Hofheinz, M. Lenander, E. Lucero, M. Neeley, A. D. O'Connell, D. Sank, H. Wang, M. Weides, J. Wenner, T. Yamamoto, A. N. Cleland, and J. M. Martinis. Fast Tunable Coupler for Superconducting Qubits. *Physical Review Letters*, 106(6):060501, February 2011. Publisher: American Physical Society.

[45] Kelly Boothby, Paul Bunyk, Jack Raymond, and Aidan Roy. Next-Generation Topology of D-Wave Quantum Processors, February 2020. arXiv:2003.00133 [quant-ph].

[46] Petar Jurcevic, Ali Javadi-Abhari, Lev S. Bishop, Isaac Lauer, Daniela F. Bogorin, Markus Brink, Lauren Capelluto, Oktay Günlük, Toshinari Itoko, Naoki Kanazawa, Abhinav Kandala, George A. Keefe, Kevin Krsulich, William Landers, Eric P. Lewandowski, Douglas T. McClure, Giacomo Nannicini, Adinath Narasgond, Hasan M. Nayfeh, Emily Pritchett, Mary Beth Rothwell, Srikanth Srinivasan, Neereja Sundaresan, Cindy Wang, Ken X. Wei, Christopher J. Wood, Jeng-Bang Yau, Eric J. Zhang, Oliver E. Dial, Jerry M. Chow, and Jay M. Gambetta. Demonstration of quantum volume 64 on a superconducting quantum computing system. *Quantum Science and Technology*, 6(2):025020, March 2021. Publisher: IOP Publishing.

[47] Jared B. Hertzberg, Eric J. Zhang, Sami Rosenblatt, Easwar Magesan, John A. Smolin, Jeng-Bang Yau, Vivekananda P. Adiga, Martin Sandberg, Markus Brink, Jerry M. Chow, and Jason S. Orcutt. Laser-annealing Josephson junctions for yielding scaled-up superconducting quantum processors. *npj Quantum Information*, 7(1):1–8, August 2021. Number: 1 Publisher: Nature Publishing Group.

[48] Stephen Brierley. Efficient implementation of Quantum circuits with limited qubit interactions, September 2016. arXiv:1507.04263 [quant-ph].

[49] Dmitri Maslov. Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics*, 19(2):023035, February 2017. arXiv: 1603.07678.

[50] D. Maslov, S. M. Falconer, and M. Mosca. Quantum Circuit Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):752–763, April 2008. arXiv: quant-ph/0703256.

[51] Dmitri Maslov, Gerhard W. Dueck, D. Michael Miller, and Camille Negrevergne. Quantum Circuit Simplification and Level Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(3):436–444, March

2008. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[52] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the qubit routing problem. page 32 pages, 2019. arXiv:1902.08091 [quant-ph].

[53] Andrew M. Childs, Eddie Schoute, and Cem M. Unsal. Circuit Transformations for Quantum Architectures. page 24, 2019. arXiv:1902.09102 [quant-ph].

[54] Will Finigan, Michael Cubeddu, Thomas Lively, Johannes Flick, and Prineha Narang. Qubit Allocation for Noisy Intermediate-Scale Quantum Computers, October 2018. arXiv:1810.08291 [quant-ph].

[55] Siyuan Niu, Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era. *IEEE Transactions on Quantum Engineering*, 1:1–14, 2020. Conference Name: IEEE Transactions on Quantum Engineering.

[56] Adam Holmes, Sonika Johri, Gian Giacomo Guerreschi, James S. Clarke, and A. Y. Matsuura. Impact of qubit connectivity on quantum algorithm performance. *Quantum Science and Technology*, 5(2):025009, March 2020. Publisher: IOP Publishing.

[57] Daniel Gottesman. Theory of fault-tolerant quantum computation. *Physical Review A*, 57(1):127–137, January 1998. Publisher: American Physical Society.

[58] A. Yu Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191, December 1997. Publisher: IOP Publishing.

[59] Christopher M. Dawson and Michael A. Nielsen. The Solovay-Kitaev algorithm. *Quantum Information & Computation*, 6(1):81–95, January 2006.

[60] J. Ignacio Cirac and Peter Zoller. Goals and opportunities in quantum simulation. *Nature Physics*, 8(4):264–266, April 2012. Number: 4 Publisher: Nature Publishing Group.

[61] Edward Farhi and Sam Gutmann. An Analog Analogue of a Digital Quantum Computation. *arXiv:quant-ph/9612026*, December 1996. arXiv: quant-ph/9612026.

[62] Dorit Aharonov, Wim van Dam, Julia Kempe, Zeph Landau, Seth Lloyd, and Oded Regev. Adiabatic Quantum Computation Is Equivalent to Standard Quantum Computation. *SIAM Review*, 50(4):755–787, January 2008. Publisher: Society for Industrial and Applied Mathematics.

[63] R. Barends, A. Shabani, L. Lamata, J. Kelly, A. Mezzacapo, U. Las Heras, R. Babbush, A. G. Fowler, B. Campbell, Yu Chen, Z. Chen, B. Chiaro, A. Dunsworth, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, P. J. J. O'Malley, C. Quintana, P. Roushan, D. Sank, A. Vainsencher, J. Wenner, T. C.

White, E. Solano, H. Neven, and John M. Martinis. Digitized adiabatic quantum computing with a superconducting circuit. *Nature*, 534(7606):222–226, June 2016. Number: 7606 Publisher: Nature Publishing Group.

[64] Andrew M. Childs, Edward Farhi, and John Preskill. Robustness of adiabatic quantum computation. *Physical Review A*, 65(1), December 2001. arXiv: quant-ph/0108048.

[65] Antti P. Vepsäläinen, Amir H. Karamlou, John L. Orrell, Akshunna S. Dogra, Ben Loer, Francisca Vasconcelos, David K. Kim, Alexander J. Melville, Bethany M. Niedzielski, Jonilyn L. Yoder, Simon Gustavsson, Joseph A. Formaggio, Brent A. VanDevender, and William D. Oliver. Impact of ionizing radiation on superconducting qubit coherence. *Nature*, 584(7822):551–556, August 2020. Number: 7822 Publisher: Nature Publishing Group.

[66] D. Binder, E. C. Smith, and A. B. Holman. Satellite Anomalies from Galactic Cosmic Rays. *IEEE Transactions on Nuclear Science*, 22(6):2675–2680, December 1975. Conference Name: IEEE Transactions on Nuclear Science.

[67] Wen Lin Tan. *Cryogenic trapped-ion system for large scale quantum simulation*. PhD thesis, University of Maryland, College Park, 2021.

[68] K. Kim, M.-S. Chang, R. Islam, S. Korenblit, L.-M. Duan, and C. Monroe. Entanglement and Tunable Spin-Spin Couplings between Trapped Ions Using Multiple Transverse Modes. *Physical Review Letters*, 103(12):120502, September 2009. Publisher: American Physical Society.

[69] Ehud Altman, Kenneth R. Brown, Giuseppe Carleo, Lincoln D. Carr, Eugene Demler, Cheng Chin, Brian DeMarco, Sophia E. Economou, Mark Eriksson, Kai-Mei C. Fu, Markus Greiner, Kaden R. A. Hazzard, Randall G. Hulet, Alicia J. Kollar, Benjamin L. Lev, Mikhail D. Lukin, Ruichao Ma, Xiao Mi, Shashank Misra, Christopher Monroe, Kater Murch, Zaira Nazario, Kang-Kuen Ni, Andrew C. Potter, Pedram Roushan, Mark Saffman, Monika Schleier-Smith, Irfan Siddiqi, Raymond Simmonds, Meenakshi Singh, I. B. Spielman, Kristan Temme, David S. Weiss, Jelena Vuckovic, Vladan Vuletic, Jun Ye, and Martin Zwierlein. Quantum Simulators: Architectures and Opportunities. *arXiv:1912.06938 [cond-mat, physics:physics, physics:quant-ph]*, December 2019. arXiv: 1912.06938.

[70] G. Pagano, P. W. Hess, H. B. Kaplan, W. L. Tan, P. Richerme, P. Becker, A. Kyprianidis, J. Zhang, E. Birckelbaw, M. R. Hernandez, Y. Wu, and C. Monroe. Cryogenic trapped-ion system for large scale quantum simulation. *Quantum Science and Technology*, 4(1):014004, October 2018. Publisher: IOP Publishing.

[71] J. Zhang, P. W. Hess, A. Kyprianidis, P. Becker, A. Lee, J. Smith, G. Pagano, I.-D. Potirniche, A. C. Potter, A. Vishwanath, N. Y. Yao, and C. Monroe. Observation of a discrete time crystal. *Nature*, 543(7644):217–220, March 2017. Number: 7644 Publisher: Nature Publishing Group.

[72] J. Zhang, G. Pagano, P. W. Hess, A. Kyprianidis, P. Becker, H. Kaplan, A. V. Gorshkov, Z.-X. Gong, and C. Monroe. Observation of a many-body dynamical phase transition with a 53-qubit quantum simulator. *Nature*, 551(7682):601–604, November 2017. Number: 7682 Publisher: Nature Publishing Group.

[73] Guido Pagano, Aniruddha Bapat, Patrick Becker, Katherine S. Collins, Arinjoy De, Paul W. Hess, Harvey B. Kaplan, Antonis Kyprianidis, Wen Lin Tan, Christopher Baldwin, Lucas T. Brady, Abhinav Deshpande, Fangli Liu, Stephen Jordan, Alexey V. Gorshkov, and Christopher Monroe. Quantum approximate optimization of the long-range Ising model with a trapped-ion quantum simulator. *Proceedings of the National Academy of Sciences*, 117(41):25396–25401, October 2020. Publisher: Proceedings of the National Academy of Sciences.

[74] A. Kyprianidis, F. Machado, W. Morong, P. Becker, K. S. Collins, D. V. Else, L. Feng, P. W. Hess, C. Nayak, G. Pagano, N. Y. Yao, and C. Monroe. Observation of a prethermal discrete time crystal. *Science*, 372(6547):1192–1196, June 2021. Publisher: American Association for the Advancement of Science.

[75] W. L. Tan, P. Becker, F. Liu, G. Pagano, K. S. Collins, A. De, L. Feng, H. B. Kaplan, A. Kyprianidis, R. Lundgren, W. Morong, S. Whitsitt, A. V. Gorshkov, and C. Monroe. Domain-wall confinement and dynamics in a quantum simulator. *Nature Physics*, 17(6):742–747, June 2021. Number: 6 Publisher: Nature Publishing Group.

[76] W. Morong, F. Liu, P. Becker, K. S. Collins, L. Feng, A. Kyprianidis, G. Pagano, T. You, A. V. Gorshkov, and C. Monroe. Observation of Stark many-body localization without disorder. *Nature*, 599(7885):393–398, November 2021. Number: 7885 Publisher: Nature Publishing Group.

[77] P. W. Hess, P. Becker, H. B. Kaplan, A. Kyprianidis, A. C. Lee, B. Neyenhuis, G. Pagano, P. Richerme, C. Senko, J. Smith, W. L. Tan, J. Zhang, and C. Monroe. Non-thermalization in trapped atomic ion spin chains. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2108):20170107, December 2017. Publisher: Royal Society.

[78] Tameem Albash and Daniel A. Lidar. Adiabatic quantum computation. *Reviews of Modern Physics*, 90(1):015002, January 2018. Publisher: American Physical Society.

[79] Sergey Novikov, Robert Hinkey, Steven Disseler, James I Basham, Tameem Albash, Andrew Risinger, David Ferguson, Daniel A. Lidar, and Kenneth M. Zick. Exploring More-Coherent Quantum Annealing. In *2018 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–7, November 2018. arXiv: 1809.04485.

[80] Satoshi Matsubara, Motomu Takatsu, Toshiyuki Miyazawa, Takayuki Shibasaki, Yasuhiro Watanabe, Kazuya Takemoto, and Hirotaka Tamura. Digital Annealer for

High-Speed Solving of Combinatorial optimization Problems and Its Applications. In *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 667–672, January 2020. ISSN: 2153-697X.

[81] Maliheh Aramon, Gili Rosenberg, Elisabetta Valiante, Toshiyuki Miyazawa, Hirotaka Tamura, and Helmut G. Katzgraber. Physics-Inspired Optimization for Quadratic Unconstrained Problems Using a Digital Annealer. *Frontiers in Physics*, 7, 2019.

[82] Hayato Goto, Kosuke Tatsumura, and Alexander R. Dixon. Combinatorial optimization by simulating adiabatic bifurcations in nonlinear Hamiltonian systems. *Science Advances*, 5(4):eaav2372, April 2019. Publisher: American Association for the Advancement of Science.

[83] Hiroki Oshiyama and Masayuki Ohzeki. Benchmark of quantum-inspired heuristic solvers for quadratic unconstrained binary optimization. *Scientific Reports*, 12(1):2146, February 2022. Number: 1 Publisher: Nature Publishing Group.

[84] Eleanor G. Rieffel, Davide Venturelli, Bryan O'Gorman, Minh B. Do, Elicia M. Prystay, and Vadim N. Smelyanskiy. A case study in programming a quantum annealer for hard operational planning problems. *Quantum Information Processing*, 14(1):1–36, January 2015.

[85] Wolfgang Paul. Electromagnetic traps for charged and neutral particles. *Reviews of Modern Physics*, 62(3):531–540, July 1990.

[86] R. T. Sutherland, R. Srinivas, S. C. Burd, H. M. Knaack, A. C. Wilson, D. J. Wineland, D. Leibfried, D. T. C. Allcock, D. H. Slichter, and S. B. Libby. Laser-free trapped-ion entangling gates with simultaneous insensitivity to qubit and motional decoherence. *Physical Review A*, 101(4):042334, April 2020. Publisher: American Physical Society.

[87] R. T. Sutherland, R. Srinivas, S. C. Burd, D. Leibfried, A. C. Wilson, D. J. Wineland, D. T. C. Allcock, D. H. Slichter, and S. B. Libby. Versatile laser-free trapped-ion entangling gates. *New Journal of Physics*, 21(3):033033, March 2019. Publisher: IOP Publishing.

[88] R. Srinivas, S. C. Burd, H. M. Knaack, R. T. Sutherland, A. Kwiatkowski, S. Glancy, E. Knill, D. J. Wineland, D. Leibfried, A. C. Wilson, D. T. C. Allcock, and D. H. Slichter. High-fidelity laser-free universal control of trapped ion qubits. *Nature*, 597(7875):209–213, September 2021. Number: 7875 Publisher: Nature Publishing Group.

[89] Ye Wang, Mark Um, Junhua Zhang, Shuoming An, Ming Lyu, Jing-Ning Zhang, L.-M. Duan, Dahyun Yum, and Kihwan Kim. Single-qubit quantum memory exceeding ten-minute coherence time. *Nature Photonics*, 11(10):646–650, October 2017.

[90] Pengfei Wang, Chun-Yang Luan, Mu Qiao, Mark Um, Junhua Zhang, Ye Wang, Xiao Yuan, Mile Gu, Jingning Zhang, and Kihwan Kim. Single ion qubit with estimated coherence time exceeding one hour. *Nature Communications*, 12(1):233, January 2021. Number: 1 Publisher: Nature Publishing Group.

[91] David Hucul, Justin E. Christensen, Eric R. Hudson, and Wesley C. Campbell. Spectroscopy of a Synthetic Trapped Ion Qubit. *Physical Review Letters*, 119(10):100501, September 2017. Publisher: American Physical Society.

[92] Falco Reissig, Klaus Kopka, and Constantin Mamat. The impact of barium isotopes in radiopharmacy and nuclear medicine – From past to presence. *Nuclear Medicine and Biology*, 98-99:59–68, July 2021.

[93] Laird Nicholas Egan. *Scaling Quantum Computers with Long Chains of Trapped Ions*. PhD thesis, University of Maryland, College Park, 2021. Accepted: 2021-07-14T05:31:59Z.

[94] Joshua M. Wilson, Julia N. Tilles, Raymond A. Haltli, Eric Ou, Matthew G. Blain, Susan M. Clark, and Melissa C. Revelle. In situ detection of RF breakdown on microfabricated surface ion traps. *Journal of Applied Physics*, 131(13):134401, April 2022. Publisher: American Institute of Physics.

[95] James D. Siverns and Qudsia Quraishi. Ion trap architectures and new directions. *Quantum Information Processing*, 16(12):314, November 2017.

[96] D. Stick, W. K. Hensinger, S. Olmschenk, M. J. Madsen, K. Schwab, and C. Monroe. Ion trap in a semiconductor chip. *Nature Physics*, 2(1):36–39, January 2006. Number: 1 Publisher: Nature Publishing Group.

[97] Peter Lukas Wilhelm Maunz. High Optical Access Trap 2.0. Technical Report SAND-2016-0796R, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), January 2016.

[98] Daniel Lobser, Matthew G. Blain, Craig William Hogle, Melissa Revelle, Daniel Lynn Stick, Christopher G. Yale, and Peter Lukas Wilhelm Maunz. Quantum and Classical Control of Ions in Sandia's HOA Trap. Technical Report SAND2017-8584C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), August 2017.

[99] Daniel Lobser, Matthew Blain, Craig Hogle, Melissa Revelle, Daniel Stick, Christopher Yale, and Peter Maunz. Precision Control of Ions in Sandia's HOA Trap. Technical Report SAND2019-11932C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), October 2019.

[100] Peter Lukas Wilhelm Maunz, Craig Robert Clark, Susan M. Clark, Paul James Resnick, Christian Lew Arrington, Francisco M. Benito, Robert R. Boye, A. Robert Ellis, Raymond A. Haltli, Edwin J. Heller, Andrew E. Hollowell, Shanalyn A. Kemme, Becky G. Loviza, Jonathan A. Mizrahi, Anathea C. Ortega, David

Scrymgeour, Jonathan David Sterk, Christopher P. Tigges, Amber Lynn Young, Daniel Lynn Stick, and Matthew Glenn Blain. Sandia Micro-fabricated Ion Traps for the MUSIQC architecture. Technical Report SAND2013-7302C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), August 2013.

[101] Peter Lukas Wilhelm Maunz, Susan M. Clark, Craig William Hogle, Raymond A. Haltli, Daniel Lobser, Jessica Marie Pehr, Melissa Revelle, Brandon Ruzic, Christopher G. Yale, and Matthew G. Blain. Microfabricated ion traps for LogiQal Qubits. Technical Report SAND2020-0374C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), January 2020.

[102] Peter Lukas Wilhelm Maunz, Craig Robert Clark, Raymond A. Haltli, Andrew E. Hollowell, John F. Rembetski, Paul J. Resnick, Jonathan David Sterk, Daniel Lynn Stick, Boyan Tabakov, and Matthew G. Blain. Characterization of a High-Optical-Access surface trap optimized for quantum information processing. Technical Report SAND2015-1045C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), February 2015.

[103] Daniel Lobser, Matthew G. Blain, Kevin Michael Fortier, Raymond A. Haltli, Andrew E. Hollowell, Jonathan Mizrahi, Jonathan David Sterk, and Peter Lukas Wilhelm Maunz. A Demonstration of the HOA Trap a Versatile Microfabricated Surface Ion Trap. Technical Report SAND2016-8060C, Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), August 2016.

[104] William D. Phillips. Nobel Lecture: Laser cooling and trapping of neutral atoms. *Reviews of Modern Physics*, 70(3):721–741, July 1998. Publisher: American Physical Society.

[105] J. M. Pino, J. M. Dreiling, C. Figgatt, J. P. Gaebler, S. A. Moses, M. S. Allman, C. H. Baldwin, M. Foss-Feig, D. Hayes, K. Mayer, C. Ryan-Anderson, and B. Neyenhuis. Demonstration of the trapped-ion quantum CCD computer architecture. *Nature*, 592(7853):209–213, April 2021. Number: 7853 Publisher: Nature Publishing Group.

[106] Harvey B. Kaplan. *Many-Body Dephasing in a Cryogenic Trapped Ion Quantum Simulator*. PhD thesis, University of Maryland, College Park, 2019.

[107] D. J. Wineland and Wayne M. Itano. Laser cooling of atoms. *Physical Review A*, 20(4):1521–1540, October 1979. Publisher: American Physical Society.

[108] Crystal Noel. *High temperature studies of electric-field noise in a surface ion trap*. PhD thesis, University of California, Berkeley, Berkeley, CA, 2019.

[109] J.-S. Chen, K. Wright, N. C. Pisenti, D. Murphy, K. M. Beck, K. Landsman, J. M. Amini, and Y. Nam. Efficient sideband cooling protocol for long trapped-ion chains. *Physical Review A*, 102(4):043110, October 2020. arXiv: 2002.04133.

[110] Wayne M. Itano and D. J. Wineland. Laser cooling of ions stored in harmonic and penning traps. *Phys. Rev. A*, 25:35–54, Jan 1982.

[111] C. Monroe, D. M. Meekhof, B. E. King, S. R. Jefferts, W. M. Itano, D. J. Wineland, and P. Gould. Resolved-Sideband Raman Cooling of a Bound Atom to the 3D Zero-Point Energy. *Physical Review Letters*, 75(22):4011–4014, November 1995. Publisher: American Physical Society.

[112] S. Olmschenk, K. C. Younge, D. L. Moehring, D. N. Matsukevich, P. Maunz, and C. Monroe. Manipulation and detection of a trapped Yb+ hyperfine qubit. *Physical Review A*, 76(5):052314, November 2007.

[113] Caroline Figgatt. *Building and Programming a Universal Ion Trap Quantum Computer*. PhD thesis, University of Maryland, College Park, 2018. Accepted: 2018-07-17T06:20:57Z.

[114] Kevin Antony Landsman. *CONSTRUCTION, OPTIMIZATION, AND APPLICATIONS OF A SMALL TRAPPED-ION QUANTUM COMPUTER*. PhD thesis, University of Maryland, College Park, 2019. Accepted: 2019-06-19T05:42:06Z.

[115] Naleli Matjelo, Nancy Payne, Charles Rigby, and Ncamiso Khanyile. Demonstration Of Rabi-Flops With Ytterbium 171 Trapped-Ion Qubits. *International Journal of Scientific and Research Publications (IJSRP)*, 11(7):137–153, July 2021.

[116] Rachel Noek, Geert Vrijsen, Daniel Gaultney, Emily Mount, Taehyun Kim, Peter Maunz, and Jungsang Kim. High Speed, High Fidelity Detection of an Atomic Hyperfine Qubit. *Optics Letters*, 38(22):4735, November 2013. arXiv: 1304.3511.

[117] D. B. Hume, T. Rosenband, and D. J. Wineland. High-Fidelity Adaptive Qubit Detection through Repetitive Quantum Nondemolition Measurements. *Physical Review Letters*, 99(12):120502, September 2007. Publisher: American Physical Society.

[118] A. H. Myerson, D. J. Szwer, S. C. Webster, D. T. C. Allcock, M. J. Curtis, G. Imreh, J. A. Sherman, D. N. Stacey, A. M. Steane, and D. M. Lucas. High-Fidelity Readout of Trapped-Ion Qubits. *Physical Review Letters*, 100(20):200502, May 2008. Publisher: American Physical Society.

[119] Daiwei Zhu. *A study of Quantum ALgorithms with Ion-trap Quantum Computers*. PhD thesis, University of Maryland, College Park, 2021. Accepted: 2021-09-16T05:33:03Z.

[120] M. Cetina, L.N. Egan, C. Noel, M.L. Goldman, D. Biswas, A.R. Risinger, D. Zhu, and C. Monroe. Control of Transverse Motion for Quantum Gates on Individually Addressed Atomic Qubits. *PRX Quantum*, 3(1):010334, March 2022. Publisher: American Physical Society.

[121] V. Negnevitsky, M. Marinelli, K. K. Mehta, H.-Y. Lo, C. Flühmann, and J. P. Home. Repeated multi-qubit readout and feedback with a mixed-species trapped-ion register. *Nature*, 563(7732):527–531, November 2018. Number: 7732 Publisher: Nature Publishing Group.

[122] Owen Astrachan. Bubble sort: an archaeological algorithmic analysis. *ACM SIGCSE Bulletin*, 35(1):1–5, January 2003.

[123] Thomas H. Cormen, editor. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 3rd ed edition, 2009. OCLC: ocn311310321.

[124] Hartmut Haffner. Quantum computing with trapped ions. *Journal of the Indian Institute of Science*, 89(3):317–331, 2009. Number: 3.

[125] D. Hayes, D. N. Matsukevich, P. Maunz, D. Hucul, Q. Quraishi, S. Olmschenk, W. Campbell, J. Mizrahi, C. Senko, and C. Monroe. Entanglement of Atomic Qubits Using an Optical Frequency Comb. *Physical Review Letters*, 104(14):140501, April 2010. Publisher: American Physical Society.

[126] F. Motzoi, J. M. Gambetta, P. Rebentrost, and F. K. Wilhelm. Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits. *Physical Review Letters*, 103(11):110501, September 2009. Publisher: American Physical Society.

[127] J. M. Gambetta, F. Motzoi, S. T. Merkel, and F. K. Wilhelm. Analytic control methods for high-fidelity unitary operations in a weakly nonlinear oscillator. *Physical Review A*, 83(1):012308, January 2011. Publisher: American Physical Society.

[128] David C. McKay, Christopher J. Wood, Sarah Sheldon, Jerry M. Chow, and Jay M. Gambetta. Efficient Z gates for quantum computing. *Physical Review A*, 96(2):022330, August 2017. Publisher: American Physical Society.

[129] Kenneth R. Brown, Aram W. Harrow, and Isaac L. Chuang. Arbitrarily accurate composite pulse sequences. *Physical Review A*, 70(5):052318, November 2004. Publisher: American Physical Society.

[130] S. Wimperis. Broadband, Narrowband, and Passband Composite Pulses for Use in Advanced NMR Experiments. *Journal of Magnetic Resonance, Series A*, 109(2):221–231, August 1994.

[131] G.J. Milburn, S. Schneider, and D.F.V James. Ion Trap Quantum Computing with Warm Ions. *Fortschritte der Physik*, 48(9-11):801–810, 2000. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/1521-3978%28200009%2948%3A9/11%3C801%3A%3AAID-PROP801%3E3.0.CO%3B2-1.

[132] Shi-Liang Zhu, C. Monroe, and L.-M. Duan. Trapped Ion Quantum Computation with Transverse Phonon Modes. *Physical Review Letters*, 97(5):050505, August 2006. Publisher: American Physical Society.

[133] Timothy Andrew Manning. *Quantum Information Processing with Trapped Ion Chains*. PhD thesis, University of Maryland, College Park, 2014. Accepted: 2014-06-24T05:39:32Z.

[134] Mohamed Abdelhafez, Brian Baker, András Gyenis, Pranav Mundada, Andrew A. Houck, David Schuster, and Jens Koch. Universal gates for protected superconducting qubits using optimal control. *Physical Review A*, 101(2):022321, February 2020. Publisher: American Physical Society.

[135] Shantanu Debnath. *A Programmable Five Qubit Quantum Computer Using Trapped Atomic Ions*. PhD thesis, 2016. Accepted: 2017-01-24T06:43:50Z.

[136] Grzegorz Kasprowicz, Paweł Kulik, Michal Gaska, Tomasz Przywozki, Krzysztof Pozniak, Jakub Jarosinski, Joseph W. Britton, Joseph W. Britton, Thomas Harty, Chris Balance, Weida Zhang, David Nadlinger, Daniel Slichter, David Allcock, Sébastien Bourdeauducq, Robert Jördens, and Krzysztof Pozniak. ARTIQ and Sinara: Open Software and Hardware Stacks for Quantum Physics. In *OSA Quantum 2.0 Conference (2020), paper QTu8B.14*, page QTu8B.14. Optica Publishing Group, September 2020.

[137] Paweł Kulik, Grzegorz Kasprowicz, and Michał Gąska. Driver module for quantum computer experiments: Kasli. In *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018*, volume 10808, pages 1255–1258. SPIE, October 2018.

[138] James Tandon. The OpenRISC processor: open hardware and Linux. *Linux Journal*, 2011(212):6:6, December 2011.

[139] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019. Conference Name: IEEE Std 754-2019 (Revision of IEEE 754-2008).

[140] AD9910 Datasheet. Datasheet, Analog Devices, October 2016. https://www.analog.com/media/en/technical-documentation/data-sheets/ad9910.pdf.

[141] Leon Riesebos, Brad Bondurant, Jacob Whitlow, Junki Kim, Mark Kuzyk, Tianyi Chen, Samuel Phiri, Ye Wang, Chao Fang, Andrew Van Horn, Jungsang Kim, and Kenneth R. Brown. Modular software for real-time quantum control systems. In *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 545–555, September 2022.

[142] Leon Riesebos, Brad Bondurant, and Kenneth R. Brown. Universal Graph-Based Scheduling for Quantum Systems. *IEEE Micro*, 41(5):57–65, September 2021. Conference Name: IEEE Micro.

[143] Jerry Moy Chow. *Quantum information processing with superconducting qubits*. Ph.D., Yale University, United States – Connecticut, 2010. ISBN: 9781124089010.

[144] Jonathan Albert Mizrahi. *Ultrafast Control of Spin and Motion in Trapped Ions*. PhD thesis, University of Maryland, College Park, 2013.

[145] Daniel Lobser, Joshua Goldberg, Andrew J Landahl, Peter Maunz, Benjamin C A Morrison, Kenneth Rudinger, Antonio Russo, and Daniel Stick. JaqalPaw: A Guide to Defining Pulses and Waveforms for Jaqal. November 2021.

[146] I. V. Inlek, G. Vittorini, D. Hucul, C. Crocker, and C. Monroe. Quantum gates with phase stability over space and time. *Physical Review A*, 90(4):042316, October 2014. Publisher: American Physical Society.

[147] Navin Khaneja, Timo Reiss, Cindie Kehlet, Thomas Schulte-Herbrüggen, and Steffen J. Glaser. Optimal control of coupled spin dynamics: design of NMR pulse sequences by gradient ascent algorithms. *Journal of Magnetic Resonance*, 172(2):296–305, February 2005.

[148] Christopher D. B. Bentley, Harrison Ball, Michael J. Biercuk, Andre R. R. Carvalho, Michael R. Hush, and Harry J. Slatyer. Numeric Optimization for Configurable, Parallel, Error-Robust Entangling Gates in Large Ion Registers. *Advanced Quantum Technologies*, 3(11):2000044, 2020. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qute.202000044.

[149] Mingyu Kang, Qiyao Liang, Bichen Zhang, Shilin Huang, Ye Wang, Chao Fang, Jungsang Kim, and Kenneth R. Brown. Batch Optimization of Frequency-Modulated Pulses for Robust Two-Qubit Gates in Ion Chains. *Physical Review Applied*, 16(2):024039, August 2021. Publisher: American Physical Society.

[150] R. Bowler, U. Warring, J. W. Britton, B. C. Sawyer, and J. Amini. Arbitrary waveform generator for quantum information processing with trapped ions. *Review of Scientific Instruments*, 84(3):033108, March 2013. Publisher: American Institute of Physics.

[151] Robert Jördens. Pdq2 V2.5, February 2016.

[152] Mohan Sarovar, Timothy Proctor, Kenneth Rudinger, Kevin Young, Erik Nielsen, and Robin Blume-Kohout. Detecting crosstalk errors in quantum information processors. *arXiv:1908.09855 [quant-ph]*, August 2019. arXiv: 1908.09855.

[153] Austin G. Fowler, William F. Thompson, Zhizhong Yan, Ashley M. Stephens, B. L. T. Plourde, and Frank K. Wilhelm. Long-range coupling and scalable architecture for superconducting flux qubits. *Physical Review B*, 76(17), November 2007. arXiv: cond-mat/0702620.

[154] Prakash Murali, David C. McKay, Margaret Martonosi, and Ali Javadi-Abhari. Software Mitigation of Crosstalk on Noisy Intermediate-Scale Quantum Computers. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1001–1016, March 2020. arXiv: 2001.02826.

[155] Alireza Seif, Kevin A. Landsman, Norbert M. Linke, Caroline Figgatt, C. Monroe, and Mohammad Hafezi. Machine learning assisted readout of trapped-ion qubits. *Journal of Physics B: Atomic, Molecular and Optical Physics*, 51(17):174006, September 2018. arXiv:1804.07718 [quant-ph].

[156] Susan M. Clark, Daniel Lobser, Melissa C. Revelle, Christopher G. Yale, David Bossert, Ashlyn D. Burch, Matthew N. Chow, Craig W. Hogle, Megan Ivory, Jessica Pehr, Bradley Salzbrenner, Daniel Stick, William Sweatt, Joshua M. Wilson, Edward Winrow, and Peter Maunz. Engineering the Quantum Scientific Computing Open User Testbed. *IEEE Transactions on Quantum Engineering*, 2:1–32, 2021. Conference Name: IEEE Transactions on Quantum Engineering.

[157] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv:1707.03429 [quant-ph]*, July 2017. arXiv: 1707.03429.

[158] Dripto M. Debroy, Laird Egan, Crystal Noel, Andrew Risinger, Daiwei Zhu, Debopriyo Biswas, Marko Cetina, Chris Monroe, and Kenneth R. Brown. Optimizing Stabilizer Parities for Improved Logical Qubit Memories. *Physical Review Letters*, 127(24):240501, December 2021. arXiv:2105.05068 [quant-ph].

[159] Crystal Noel, Pradeep Niroula, Daiwei Zhu, Andrew Risinger, Laird Egan, Debopriyo Biswas, Marko Cetina, Alexey V. Gorshkov, Michael J. Gullans, David A. Huse, and Christopher Monroe. Observation of measurement-induced quantum phases in a trapped-ion quantum computer. *Nature Physics*, 18(7):760–764, July 2022. arXiv:2106.05881 [quant-ph].

[160] Kushal Seetharam, Debopriyo Biswas, Crystal Noel, Andrew Risinger, Daiwei Zhu, Or Katz, Sambuddha Chattopadhyay, Marko Cetina, Christopher Monroe, Eugene Demler, and Dries Sels. Digital quantum simulation of NMR experiments, September 2021. arXiv:2109.13298 [physics, physics:quant-ph].

[161] Or Katz, Lei Feng, Andrew Risinger, Christopher Monroe, and Marko Cetina. Demonstration of three- and four-body interactions between trapped-ion spins, September 2022. arXiv:2209.05691 [physics, physics:quant-ph].

[162] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, April 1982.

[163] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147–153, July 1996. Publisher: American Physical Society.

[164] Adam Paetznick and Ben W. Reichardt. Universal Fault-Tolerant Quantum Computation with Only Transversal Gates and Error Correction. *Physical Review Letters*, 111(9):090505, August 2013. Publisher: American Physical Society.

[165] Yunseong Nam, Jwo-Sy Chen, Neal C. Pisenti, Kenneth Wright, Conor Delaney, Dmitri Maslov, Kenneth R. Brown, Stewart Allen, Jason M. Amini, and Joel Apisdorf. Ground-state energy estimation of the water molecule on a trapped-ion quantum computer. *npj Quantum Information*, 6(1):1–6, 2020. Publisher: Nature Publishing Group.

[166] Vivek V. Shende and Igor L. Markov. On the CNOT-cost of TOFFOLI gates, March 2008. arXiv:0803.2316 [quant-ph].

[167] Yong He, Ming-Xing Luo, E. Zhang, Hong-Ke Wang, and Xiao-Feng Wang. Decompositions of n-qubit Toffoli Gates with Linear Circuit Complexity. *International Journal of Theoretical Physics*, 56(7):2350–2361, July 2017.

[168] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften.

[169] Jonathan M. Baker, Casey Duckering, Alexander Hoover, and Frederic T. Chong. Decomposing Quantum Generalized Toffoli with an Arbitrary Number of Ancilla, April 2019. arXiv:1904.01671 [quant-ph].

[170] Yunong Shi, Pranav Gokhale, Prakash Murali, Jonathan M. Baker, Casey Duckering, Yongshan Ding, Natalie C. Brown, Christopher Chamberland, Ali Javadi Abhari, Andrew W. Cross, David I. Schuster, Kenneth R. Brown, Margaret Martonosi, and Frederic T. Chong. Resource-Efficient Quantum Computing by Breaking Abstractions. *Proceedings of the IEEE*, 108(8):1353–1370, August 2020. arXiv: 2011.00028.

[171] C. Figgatt, A. Ostrander, N. M. Linke, K. A. Landsman, D. Zhu, D. Maslov, and C. Monroe. Parallel entangling operations on a universal ion-trap quantum computer. *Nature*, 572(7769):368–372, August 2019. Number: 7769 Publisher: Nature Publishing Group.

[172] J. I. Cirac and P. Zoller. Quantum Computations with Cold Trapped Ions. *Physical Review Letters*, 74(20):4091–4094, May 1995. Publisher: American Physical Society.

[173] Or Katz, Marko Cetina, and Christopher Monroe. N-body interactions between trapped ion qubits via spin-dependent squeezing, February 2022. arXiv:2202.04230 [physics, physics:quant-ph].

[174] C. A. Sackett, D. Kielpinski, B. E. King, C. Langer, V. Meyer, C. J. Myatt, M. Rowe, Q. A. Turchette, W. M. Itano, D. J. Wineland, and C. Monroe. Experimental entanglement of four particles. *Nature*, 404(6775):256–259, March 2000. Number: 6775 Publisher: Nature Publishing Group.

[175] D. Kielpinski, C. Monroe, and D. J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417(6890):709–711, June 2002. Number: 6890 Publisher: Nature Publishing Group.

[176] Steven Balensiefer, Lucas Kregor-Stickles, and Mark Oskin. An Evaluation Framework and Instruction Set Architecture for Ion-Trap Based Quantum Micro-Architectures. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 186–196, USA, May 2005. IEEE Computer Society.

[177] Prakash Murali, Dripto M. Debroy, Kenneth R. Brown, and Margaret Martonosi. Architecting Noisy Intermediate-Scale Trapped Ion Quantum Computers. *arXiv:2004.04706 [quant-ph]*, April 2020. arXiv: 2004.04706.

[178] R. Bradford Blakestad. *Transport of trapped-ion qubits within a scalable quantum processor*. Ph.D., University of Colorado at Boulder, United States – Colorado, 2010. ISBN: 9781124193885.

[179] Ksenia Sosnova. *Mixed-Species Ion Chains for Quantum Networks*. PhD thesis, University of Maryland, College Park, 2020.

[180] Miklós Ajtai, T. S. Jayram, Ravi Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC '02, pages 370–379, New York, NY, USA, May 2002. Association for Computing Machinery.

[181] L. J. Stephenson, D. P. Nadlinger, B. C. Nichol, S. An, P. Drmota, T. G. Ballance, K. Thirumalai, J. F. Goodwin, D. M. Lucas, and C. J. Ballance. High-Rate, High-Fidelity Entanglement of Qubits Across an Elementary Quantum Network. *Physical Review Letters*, 124(11):110501, March 2020. Publisher: American Physical Society.

[182] D. P. Nadlinger, P. Drmota, B. C. Nichol, G. Araneda, D. Main, R. Srinivas, D. M. Lucas, C. J. Ballance, K. Ivanov, E. Y.-Z. Tan, P. Sekatski, R. L. Urbanke, R. Renner, N. Sangouard, and J.-D. Bancal. Experimental quantum key distribution certified by Bell's theorem. *Nature*, 607(7920):682–686, July 2022. Number: 7920 Publisher: Nature Publishing Group.

[183] B. C. Nichol, R. Srinivas, D. P. Nadlinger, P. Drmota, D. Main, G. Araneda, C. J. Ballance, and D. M. Lucas. An elementary quantum network of entangled optical atomic clocks. *Nature*, 609(7928):689–694, September 2022. Number: 7928 Publisher: Nature Publishing Group.

[184] C. J. Ballance, L. J. Stephenson, D. P. Nadlinger, B. C. Nichol, S. An, J. F. Goodwin, P. Drmota, and D. M. Lucas. Networking Trapped-ion Quantum Computers. In *Quantum Information and Measurement (QIM) V: Quantum Technologies (2019), paper S2D.1*, page S2D.1. Optica Publishing Group, April 2019.

[185] Christopher J. Ballance. *High-Fidelity Quantum Logic in Ca+*. PhD thesis, University of Oxford, 2014.

[186] C. Monroe, R. Raussendorf, A. Ruthven, K. R. Brown, P. Maunz, L.-M. Duan, and J. Kim. Large-scale modular quantum-computer architecture with atomic memory and photonic interconnects. *Physical Review A*, 89(2), February 2014.

[187] Conda, 2012. https://docs.conda.io/en/latest/.

[188] Pipenv. https://pipenv.pypa.io/en/latest/.

[189] Elijah Newren. git-filter-repo, November 2021. https://github.com/newren/git-filter-repo.

[190] venv. https://docs.python.org/3/library/venv.html.

[191] Poetry, February 2018. https://python-poetry.org/.

[192] Eelco Dolstra. *The purely functional software deployment model*. PhD thesis, Utrecht University, Netherlands, 2006. OCLC: 71702886.

[193] J. Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, January 1989.

[194] Windows Subsystem for Linux (WSL). https://docs.microsoft.com/en-us/windows/wsl/install.

[195] Nixpkgs. https://github.com/NixOS/nixpkgs.

[196] Sebastian Bourdeauducq. M-Labs Nix Scripts. https://git.m-labs.hk/M-Labs/nix-scripts.

[197] Marcus R. Munafò, Brian A. Nosek, Dorothy V. M. Bishop, Katherine S. Button, Christopher D. Chambers, Nathalie Percie du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J. Ware, and John P. A. Ioannidis. A manifesto for reproducible science. *Nature Human Behaviour*, 1(1):1–9, January 2017. Number: 1 Publisher: Nature Publishing Group.

[198] Lorri, October 2022. https://github.com/nix-community/lorri.

Index