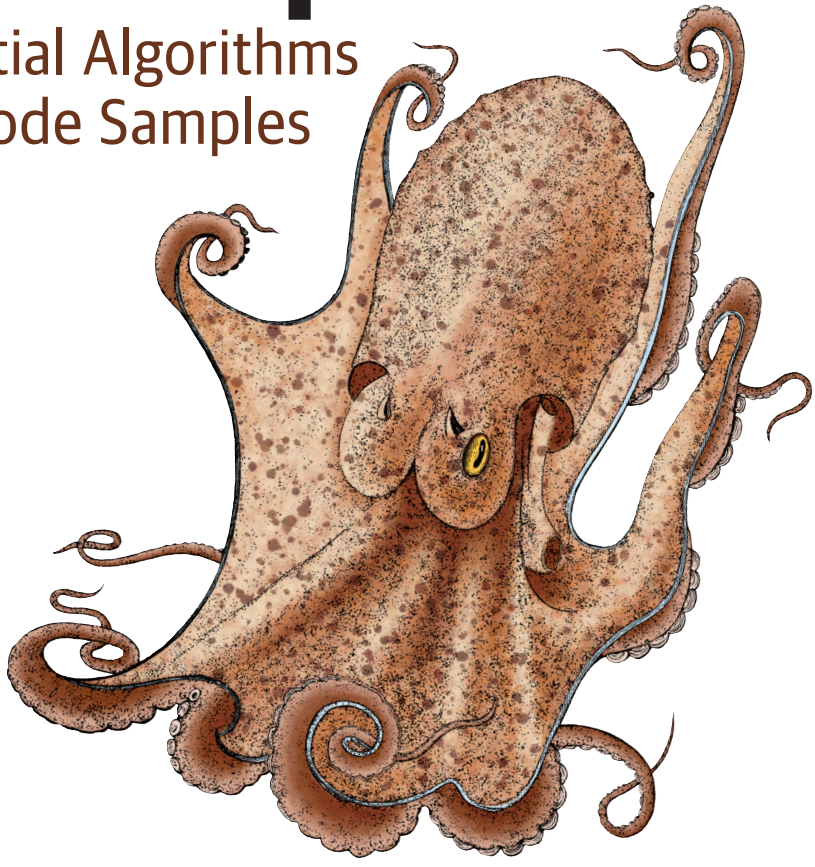
O'REILLY®

Programming Quantum Computers

Essential Algorithms and Code Samples



Eric R. Johnston, Nic Harrigan
& Mercedes Gimeno-Segovia

O'REILLY®

Programming Quantum Computers

Quantum computers are poised to kick-start a new computing revolution—and you can join in right away. If you're in software engineering, computer graphics, data science, or just an intrigued computerphile, this book provides a hands-on programmer's guide to understanding quantum computing technology. Rather than labor through math and theory, you'll work directly with examples that demonstrate this technology's unique capabilities.

Quantum computing specialists Eric Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia show you how to build the skills, tools, and intuition required to write quantum programs at the center of applications. You'll understand what quantum computers can do and learn how to identify the types of problems they can solve.

This book includes three multichapter sections:

- **Programming for a QPU**—Explore core concepts for programming quantum processing units, including how to describe and manipulate qubits and how to perform quantum teleportation.
- **QPU Primitives**—Learn algorithmic primitives and techniques, including amplitude amplification, the Quantum Fourier Transform, and phase estimation.
- **QPU Applications**—Investigate how QPU primitives are used to build existing applications, including quantum search techniques and Shor's factoring algorithm.

"This is a book written for (and by) programmers that brings inventive notation and visual tools to exploring the new world of quantum computation."

—**Mike Shapiro**
Co-creator of DTrace, DSSD, and NVMeoF

"Rather than requiring the reader to have a deep math background, the authors take a unique approach, employing visual, hands-on techniques to build quantum computing intuitions."

—**James L. Weaver**
Quantum Developer Advocate, IBM

Eric Johnston is a software engineer and creator of the QCEngine quantum computation simulator.

Nic Harrigan is a quantum architect and science communicator who has worked extensively in the foundations of quantum mechanics and quantum computing.

Mercedes Gimeno-Segovia is Director of Quantum Architecture at PsiQ, where she works on the design of a general-purpose quantum computer.

PROGRAMMING

US $69.99 CAN $92.99

ISBN: 978-1-492-03968-6

Twitter: @oreillymedia
facebook.com/oreilly

Programming Quantum Computers

Essential Algorithms and Code Samples

Eric R. Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Programming Quantum Computers
by Eric R. Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Mike Loukides
Development Editor: Michele Cronin
Production Editor: Christopher Faucher
Copyeditor: Kim Cofer
Proofreader: Rachel Head

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2019: First Edition

Revision History for the First Edition
2019-07-03: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492039686* for release details.

978-1-492-03968-6

[LSI]

Table of Contents

Part IV. Outlook

Preface

Quantum computers are no longer theoretical devices.

The authors of this book believe that the best uses for a new technology are not necessarily discovered by its inventors, but by domain experts experimenting with it as a new tool for their work. With that in mind, this book is a hands-on programmer's guide to using quantum computing technology. In the chapters ahead, you'll become familiar with symbols and operations such as those in Figure P-1, and learn how to apply them to problems you care about.

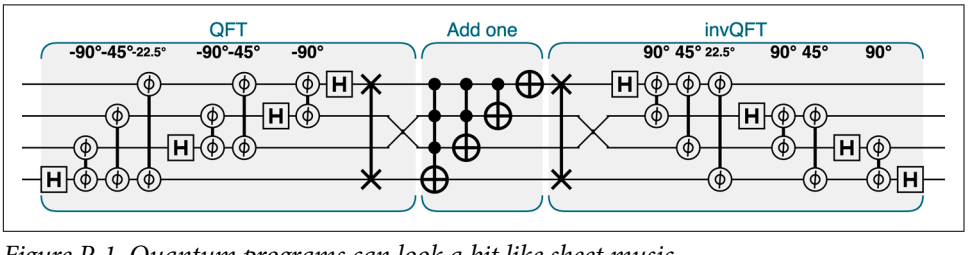


Figure P-1. Quantum programs can look a bit like sheet music

How This Book Is Structured

A tried-and-tested approach for getting hands-on with new programming paradigms is to learn a set of conceptual primitives. For example, anyone learning Graphics Processing Unit (GPU) programming should first focus on mastering the concept of parallelism, rather than on syntax or hardware specifics.

The heart of this book focuses on building an intuition for a set of quantum primitives—ideas forming a toolbox of building blocks for problem-solving with a QPU. To prepare you for these primitives, we first introduce the basic concepts of qubits (the *rules of the game*, if you like). Then, after outlining a set of Quantum Processing Unit (QPU) primitives, we show how they can be used as building blocks within useful QPU applications.

Consequently, this book is divided into three parts. The reader is encouraged to become familiar with Part I and gain some hands-on experience before proceeding to the more advanced parts:

Part I: Programming a QPU
: Here we introduce the core concepts required to program a QPU, such as qubits, essential instructions, utilizing superposition, and even quantum teleportation. Examples are provided, which can be easily run using simulators or a physical QPU.

Part II: QPU Primitives
: The second part of the book provides detail on some essential algorithms and techniques at a higher level. These include *amplitude amplification*, the *Quantum Fourier Transform*, and *phase estimation*. These can be considered "library functions" that programmers call to build applications. Understanding how they work is essential to becoming a skilled QPU programmer. An active community of researchers is working on developing new QPU primitives, so expect this library to grow in the future.

Part III: QPU Applications
: The world of QPU applications—which combine the primitives from Part II to perform useful real-world tasks—is evolving as rapidly as QPUs themselves. Here we introduce examples of existing applications.

By the end of the book we hope to provide the reader with an understanding of what quantum applications can do, what makes them powerful, and how to identify the kinds of problems that they can solve.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://oreilly-qc.github.io*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Programming Quantum Computers* by Eric R. Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia (O'Reilly). Copyright 2019 Eric R. Johnston, Nic Harrigan, and Mercedes Gimeno-Segovia, 978-1-492-03968-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

For almost 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, conferences, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/programming-quantum-computers*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

This book could not have been possible without a supporting team of talented people passionate about quantum computing. The authors would like to thank Michele, Mike, Kim, Rebecca, Chris, and the technical team at O'Reilly for sharing and amplifying our enthusiasm for the topic. Although the authors are responsible for all errors and omissions, this book benefited immensely from the invaluable feedback and inspiration of a number of technical reviewers, including Konrad Kieling, Christian Sommeregger, Mingsheng Ying, Rich Johnston, James Weaver, Mike Shapiro, Wyatt Berlinic, and Isaac Kim.

EJ would like to thank Sue, his muse. Quantum computation started to make sense to him on the week that they met. EJ also sends thanks to friends at the University of Bristol and at SolidAngle, for encouraging him to color outside the lines.

Nic would like to thank Derek Harrigan for first teaching him to talk binary, the other Harrigans for their love and support, and Shannon Burns for her pending Harrigan application.

Mercedes would like to thank José María Gimeno Blay for igniting her interest on computers early, and Mehdi Ahmadi for being a constant source of support and inspiration.

But clichéd as it may be, most of all the authors would like to thank you, the reader, for having the sense of adventure to pick up this book and learn about something so different and new.

CHAPTER 1

Introduction

Whether you're an expert in software engineering, computer graphics, data science, or just a curious computerphile, this book is designed to show how the power of quantum computing might be relevant to you, by actually allowing you to start using it.

To facilitate this, the following chapters do *not* contain thorough explanations of quantum physics (the laws underlying quantum computing) or even quantum information theory (how those laws determine our abilities to process information). Instead, they present working examples providing insight into the capabilities of this exciting new technology. Most importantly, the code we present for these examples can be tweaked and adapted. This allows you to learn from them in the most effective way possible: by getting hands-on. Along the way, core concepts are explained as they are used, and only insofar as they build an intuition for writing quantum programs.

Our humble hope is that interested readers might be able to wield these insights to apply and augment applications of quantum computing in fields that physicists may not even have heard of. Admittedly, hoping to help spark a quantum revolution isn't *that* humble, but it's definitely exciting to be a pioneer.

Required Background

The physics underlying quantum computing is full of dense mathematics. But then so is the physics behind the transistor, and yet learning C++ need not involve a single physics equation. In this book we take a similarly *programmer-centric* approach, circumventing any significant mathematical background. That said, here is a short list of knowledge that may be helpful in digesting the concepts we introduce:

- Familiarity with programming control structures (`if`, `while`, etc.). JavaScript is used in this book to provide lightweight access to samples that can be run online. If you're new to JavaScript but have some prior programming experience, the level of background you need could likely be picked up in less than an hour. For a more thorough introduction to JavaScript, see *Learning JavaScript* by Todd Brown (O'Reilly).
- Some relevant programmer-level mathematics, necessitating:
  - An understanding of using mathematical functions
  - Familiarity with trigonometric functions
  - Comfort manipulating binary numbers and converting between binary and decimal representations
  - A comprehension of the basic meaning of complex numbers
- A very elementary understanding of how to assess the computational complexity of an algorithm (i.e., *big-o* notation).

One part of the book that reaches beyond these requirements is Chapter 13, where we survey a number of applications of quantum computing to machine learning. Due to space constraints our survey gives only very cursory introductions to each machine-learning application before showing how a quantum computer can provide an advantage. Although we intend the content to be understandable to a general reader, those wishing to really experiment with these applications will benefit from a bit more of a machine-learning background.

This book is about programming (not building, nor researching) quantum computers, which is why we can do without advanced mathematics and quantum theory. However, for those interested in exploring the more academic literature on the topic, Chapter 14 provides some good initial references and links the concepts we introduce to mathematical notations commonly used by the quantum computing research community.

What Is a QPU?

Despite its ubiquity, the term "quantum computer" can be a bit misleading. It conjures images of an entirely new and alien kind of machine—one that supplants all existing computing software with a futuristic alternative.

At the time of writing this is a common, albeit huge, misconception. The promise of quantum computers stems not from them being a *conventional computer killer*, but rather from their ability to dramatically extend the kinds of problems that are tractable within computing. There are important computational problems that are easily

calculable on a quantum computer, but that would quite literally be impossible on any conceivable standard computing device that we could ever hope to build.[1]

But crucially, these kinds of speedups have only been seen for certain problems (many of which we later elucidate on), and although it is anticipated that more will be discovered, it's highly unlikely that it would ever make sense to run *all* computations on a quantum computer. For most of the tasks taking up your laptop's clock cycles, a quantum computer performs no better.

In other words—from the programmer's point of view—a quantum computer is really a *co*-processor. In the past, computers have used a wide variety of co-processors, each suited to their own specialties, such as floating-point arithmetic, signal processing, and real-time graphics. With this in mind, we will use the term *QPU* (Quantum Processing Unit) to refer to the device on which our code samples run. We feel this reinforces the important context within which quantum computing should be placed.

As with other co-processors such as the GPU (Graphics Processing Unit), programming for a QPU involves the programmer writing code that will primarily run on the CPU (Central Processing Unit) of a normal computer. The CPU issues the QPU co-processor commands only to initiate tasks suited to its capabilities.

A Hands-on Approach

Hands-on samples form the backbone of this book. But at the time of writing, a full-blown, general-purpose QPU does not exist—so how can you hope to ever run our code? Fortunately (and excitingly), even at the time of writing a few prototype QPUs *are* currently available, and can be accessed on the cloud. Furthermore, for smaller problems it's possible to *simulate* the behavior of a QPU on conventional computing hardware. Although simulating larger QPU programs becomes impossible, for smaller code snippets it's a convenient way to learn how to control an actual QPU. The code samples in this book are compatible with both of these scenarios, and will remain both usable and pedagogical even as more sophisticated QPUs become available.

There are many QPU simulators, libraries, and systems available. You can find a list of links to several well-supported systems at *http://oreilly-qc.github.io*. On that page, we provide the code samples from this book, whenever possible, in a variety of

1 One of our favorite ways to drive this point home is the following back-of-the-envelope calculation. Suppose conventional transistors could be made atom-sized, and we aimed to build a warehouse-sized conventional computer able to match even a modest quantum computer's ability to factor prime integers. We would need to pack transistors so densely that we would create a gravitational singularity. Gravitational singularities make computing (and existing) very hard.

languages. However, to prevent code samples from overwhelming the text, we provide samples only in JavaScript for QCEngine. QCEngine is a free online quantum computation simulator, allowing users to run samples in a browser, with no software installation at all. This simulator was developed by the authors, initially for their own use and now as a companion for this book. QCEngine is especially useful for us, both because it can be run without the need to download any software and because it incorporates the *circle notation* that we use as a visualization tool throughout the book.

A QCEngine Primer

Since we'll rely heavily on QCEngine, it's worth spending a little time to see how to navigate the simulator, which you can find at *http://oreilly-qc.github.io*.

Running code

The QCEngine web interface, shown in Figure 1-1, allows you to easily produce the various visualizations that we'll rely on. You can create these visualizations by simply entering code into the QCEngine code editor.

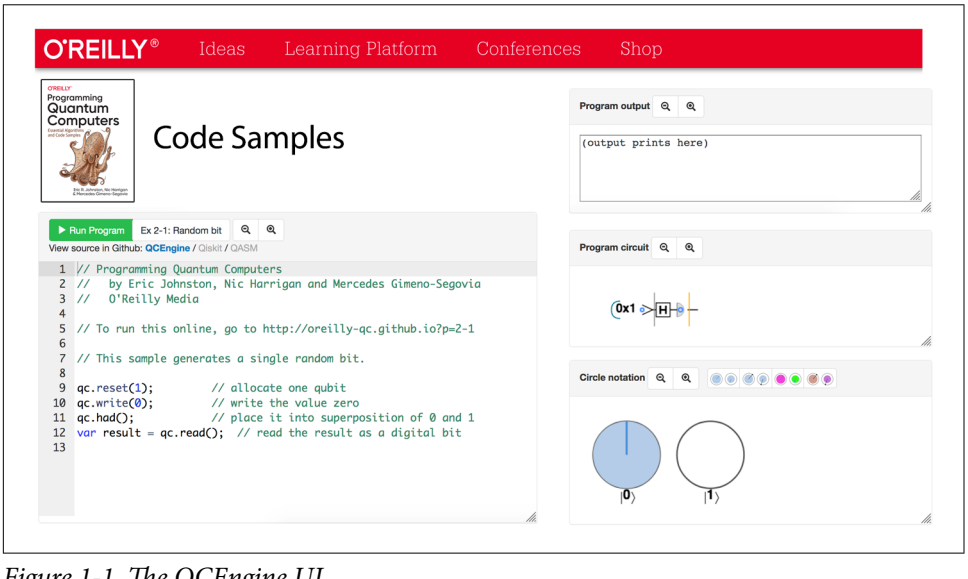


Figure 1-1. The QCEngine UI

To run one of the code samples from the book, select it from the drop-down list at the top of the editor and click the Run Program button. Some new interactive UI elements will appear for visualizing the results of running your code (see Figure 1-2).

Quantum circuit visualizer
: This element presents a visual representation of the circuit representing your code. We introduce the symbols used in these circuits in Chapters 2 and 3. This view can also be used to interactively step through the program (see Figure 1-2).

Circle-notation visualizer
: This displays the so-called *circle-notation* visualization of the QPU (or simulator) register. We explain how to read and use this notation in Chapter 2.

QCEngine output console
: This is where any text appears that may have been printed from within your code (i.e., for debugging) using the `qc.print()` command. Anything printed with the standard JavaScript `console.log()` function will still go to your web browser's JavaScript console.

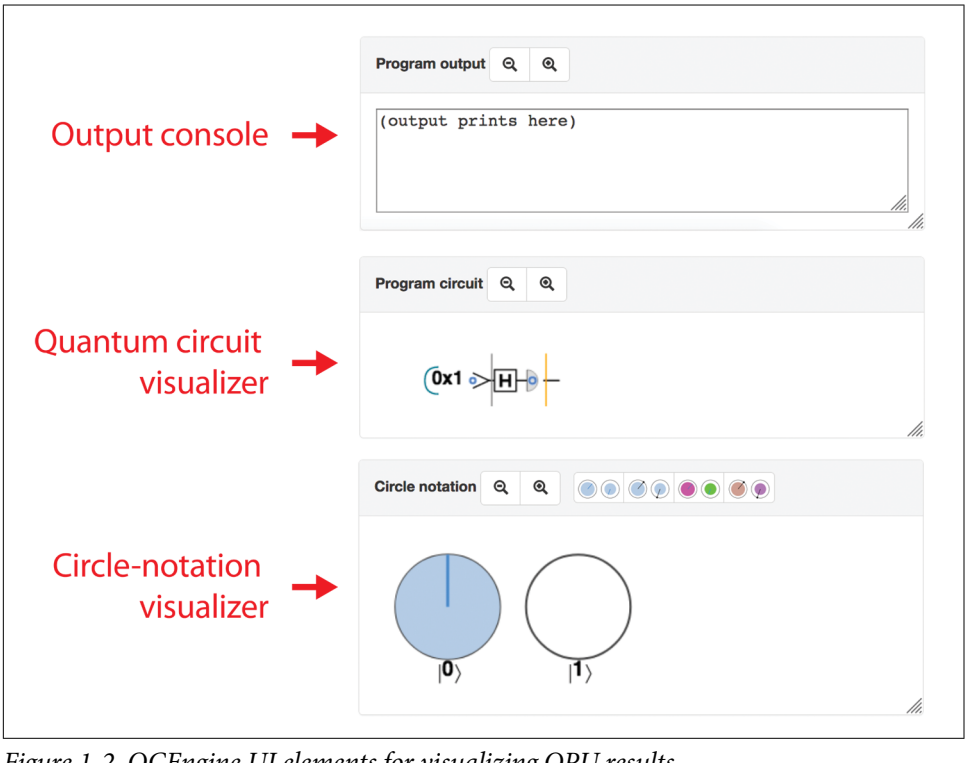


Figure 1-2. QCEngine UI elements for visualizing QPU results

Debugging code

Debugging QPU programs can be tough. Quite often the easiest way to understand what a program is doing is to slowly step through it, inspecting the visualizations at each step. Hovering your mouse over the circuit visualizer, you should see a vertical

orange line appear at a fixed position and a gray vertical line wherever in the circuit your cursor happens to be. The orange line shows which position in the circuit (and therefore the program) the circle-notation visualizer currently represents. By default this is the end of the program, but by clicking other parts of the circuit, you can have the circle-notation visualizer show the configuration of the QPU at that point in the program. For example, Figure 1-3 shows how the circle-notation visualizer changes as we switch between two different steps in the default QCEngine program.

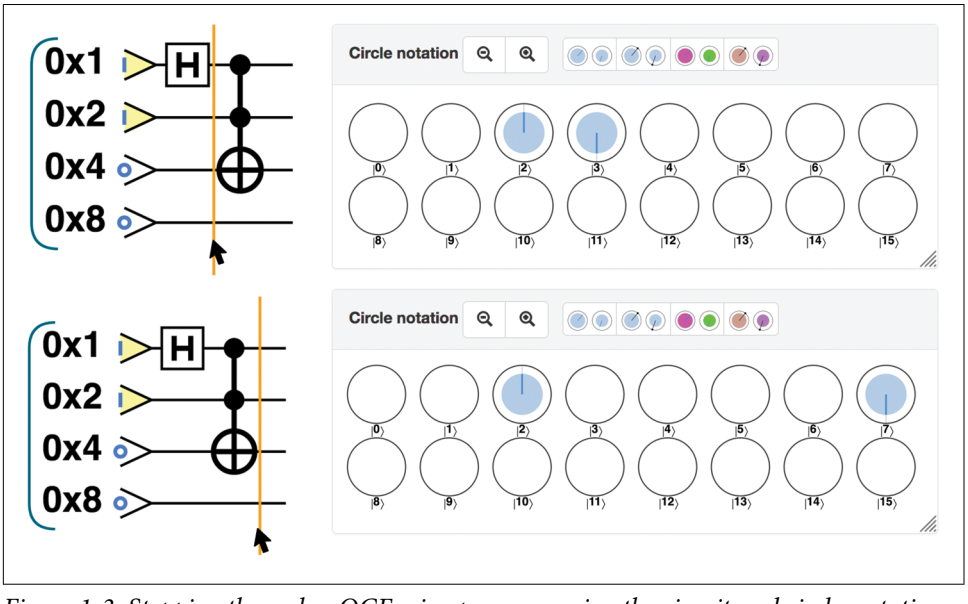


Figure 1-3. Stepping through a QCEngine program using the circuit and circle-notation visualizers

Having access to a QPU simulator, you're probably keen to start tinkering. Don't let us stop you! In Chapter 2 we'll walk through code for increasingly complex QPU programs.

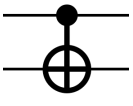
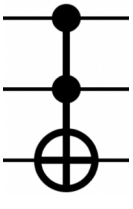
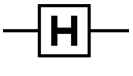
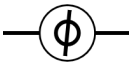
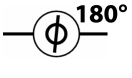
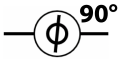
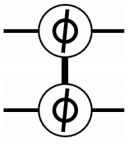
Native QPU Instructions

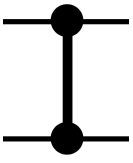
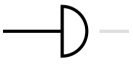
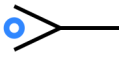
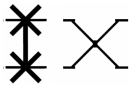
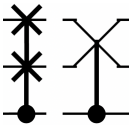
QCEngine is one of several tools allowing us to run and inspect QPU code, but what does QPU code actually look like? Conventional high-level languages are commonly used to control lower-level QPU instructions (as we've already seen with the JavaScript-based QCEngine). In this book we'll regularly cross between these levels. Describing the programming of a QPU with distinctly quantum machine-level operations helps us get to grips with the fundamental novel logic of a QPU, while seeing how to manipulate these operations from higher-level conventional languages like JavaScript, Python, or C++ gives us a more pragmatic paradigm for actually writing

code. The definition of new, bespoke, *quantum* programming languages is an active area of development. We won't highlight these in this book, but references for the interested reader are offered in Chapter 14.

To whet your appetite, we list some of the fundamental QPU instructions in Table 1-1, each of which will be explained in more detail within the chapters ahead.

Table 1-1. Essential QPU instruction set

Symbol	Name	Usage	Description
	NOT (also X)	`qc.not(t)`	Logical bitwise NOT
	CNOT	`qc.cnot(t,c)`	Controlled NOT: `if (c) then NOT(t)`
	CCNOT (Toffoli)	`qc.cnot(t,c1|c2)`	`if (c1 AND c2) then NOT(t)`
	HAD (Hadamard)	`qc.had(t)`	Hadamard gate
	PHASE	`qc.phase(angle,c)`	Relative phase rotation
	Z	`qc.phase(180,c)`	Relative phase rotation by 180 °
	S	`qc.phase(90,c)`	Relative phase rotation by 90 °
	T	`qc.phase(45,c)`	Relative phase rotation by 45 °
	CPHASE	`qc.cphase(angle,c1|c2)`	Conditional phase rotation

Symbol	Name	Usage	Description
	CZ	`qc.cphase(180,c1|c2)`	Conditional phase rotation by 180 °
	READ	`val = qc.read(t)`	Read qubits, returning digital data
	WRITE	`qc.write(t,val)`	Write conventional data to qubits
	ROOTNOT	`qc.rootnot(t)`	Root-of-NOT operation
	SWAP (EXCHANGE)	`qc.exchange(t1|t2)`	Exchange two qubits
	CSWAP	`qc.exchange(t1|t2, c)`	Conditional exchange: `if (c) then SWAP(t1,t2)`

With each of these operations, the specific instructions and timing will depend on the QPU brand and architecture. However, this is an essential set of basic operations expected to be available on all machines, and these operations form the basis of our QPU programming, just as instructions like `MOV` and `ADD` do for CPU programmers.

Simulator Limitations

Although simulators offer a fantastic opportunity to prototype small QPU programs, when compared to real QPUs they are hopelessly underpowered. One measure of the power of a QPU is the number of *qubits* it has available to operate on[2] (the quantum equivalent of bits, on which we'll have much more to say shortly).

At the time of this book's publication, the world record for the largest simulation of a QPU stands at 51 qubits. In practice, the simulators and hardware available to most

2 Despite its popularity in the media as a benchmark for quantum computing, counting the number of qubits that a piece of hardware can handle is really an oversimplification, and much subtler considerations are necessary to assess a QPU's true power.

readers of this book will typically be able to handle 26 or so qubits before grinding to a halt.

The examples in this book have been written with these limitations in mind. They make a great starting point, but each qubit added to them will double the memory required to run the simulation, cutting its speed in half.

Hardware Limitations

Conversely, the largest actual QPU hardware available at the time of writing has around 70 *physical* qubits, while the largest QPU available to the public, through the Qiskit open source software development kit, contains 16 physical qubits.[3] By physical, as opposed to logical, we mean that these 70 qubits have no error correction, making them noisy and unstable. Qubits are much more fragile than their conventional counterparts; the slightest interaction with their surroundings can derail the computation.

Dealing with *logical* qubits allows a programmer to be agnostic about the QPU hardware and implement any textbook algorithm without having to worry about specific hardware constraints. In this book, we focus solely on programming with logical qubits, and while the examples we provide are small enough to be run on smaller QPUs (such as the ones available at the time of publication), abstracting away physical hardware details means that the skills and intuitions you develop will remain invaluable as future hardware develops.

QPU Versus GPU: Some Common Characteristics

The idea of programming an entirely new kind of processor can be intimidating, even if it does already have its own Stack Exchange community. Here's a list of pertinent facts about what it's like to program a QPU:

- It is very rare that a program will run *entirely* on a QPU. Usually, a program running on a CPU will issue QPU instructions, and later retrieve the results.
- Some tasks are very well suited to the QPU, and others are not.
- The QPU runs on a separate clock from the CPU, and usually has its own dedicated hardware interfaces to external devices (such as optical outputs).
- A typical QPU has its own special RAM, which the CPU cannot efficiently access.
- A simple QPU will be one chip accessed by a laptop, or even perhaps eventually an area within another chip. A more advanced QPU is a large and expensive add-on, and always requires special cooling.

3 This figure may become out of date while this book is in press!

- Early QPUs, even simple ones, are the size of refrigerators and require special high-amperage power outlets.
- When a computation is done, a projection of the result is returned to the CPU, and most of the QPU's internal working data is discarded.
- QPU debugging can be very tricky, requiring special tools and techniques. Stepping through a program can be difficult, and often the best approach is to make changes to the program and observe their effect on the output.
- Optimizations that speed up one QPU may slow down another.

Sounds pretty challenging. But here's the thing—you can replace *QPU* with *GPU* in each and every one of those statements and they're still entirely valid.

Although QPUs are an almost alien technology of incredible power, when it comes to the problems we might face in learning to program them, a generation of software engineers have seen it all before. It's true, of course, that there are some nuances to QPU programming that are genuinely novel (this book wouldn't be necessary otherwise!), but the uncanny number of similarities should be reassuring. We can do this!

PART I

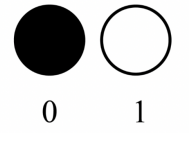
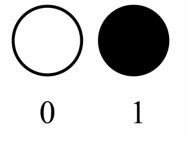
Programming for a QPU

What exactly is a qubit? How can we visualize one? How is it useful? These are short questions with complicated answers. In the first part of the book we'll answer these questions in a practical way. We begin in Chapter 2 by first describing and making use of a single qubit. The additional complexity of multi-qubit systems are then covered in Chapter 3. Along the way we'll encounter a number of single and multi-qubit operations, and Chapter 4 puts these straight to use, describing how to perform quantum teleportation. Bear in mind that the code samples that punctuate our discussion can be run on the QCEngine simulator (introduced in Chapter 1), using the provided links.

CHAPTER 2
One Qubit

Conventional bits have one binary parameter to play with—we can initialize a bit in either state 0 or state 1. This makes the mathematics of binary logic simple enough, but we *could* visually represent the possible 0/1 values of a bit by two separate empty/filled circles (see Table 2-1).

Table 2-1. Possible values of a conventional bit—a graphical representation

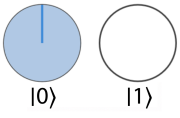
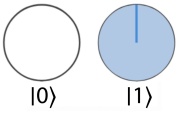
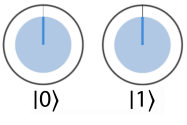
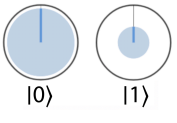
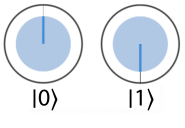
Possible values of a bit	Graphical representation
0	
1	

Now on to qubits. In some sense, qubits are very similar to bits: whenever you read the value of a qubit, you'll always get either 0 or 1. So, *after the readout* of a qubit, we can always describe it as shown in Table 2-1. But characterizing qubits *before* readout isn't so black and white, and requires a more sophisticated description. Before readout, qubits can exist in a *superposition* of states.

We'll try to tackle just what superposition means shortly. But to first give you an idea of why it might be powerful, note that there are an infinite number of possible superpositions in which a single qubit can exist prior to being read. Table 2-2 lists just

some of the different superpositions we could prepare a qubit to be in. Although we'll always end up reading out 0 or 1 at the end, if we're clever it's the availability of these extra states that will allow us to perform some very powerful computing.

Table 2-2. Some possible values of a qubit

Possible values of a qubit	Graphical representation
$\|0\rangle$	
$\|1\rangle$	
$0.707\|0\rangle + 0.707\|1\rangle$	
$0.95\|0\rangle + 0.35\|1\rangle$	
$0.707\|0\rangle - 0.707\|1\rangle$	

In the mathematics within Table 2-2 we've changed our labels from 0 and 1 to $|0\rangle$ and $|1\rangle$. This is called *bra-ket notation*, and it's commonly used in quantum computing. As a casual rule of thumb, numbers enclosed in bra-ket notation denote values that a qubit *might* be found to have when read out. When referring to a value that a qubit *has* been read out to have, we just use the number to represent the resulting digital value.

The first two rows in Table 2-2 show the quantum equivalents of the states of a conventional bit, with no superposition at all. A qubit prepared in state $|0\rangle$ is equivalent to a conventional bit being 0—it will always give a value of 0 on readout—and

similarly for $|1\rangle$. If our qubits were only ever in states $|0\rangle$ or $|1\rangle$, we'd just have some very expensive conventional bits.

But how can we start to get to grips with the more exotic possibilities of superposition shown in the other rows? To get some intuition for the bewildering variations in Table 2-2, it can be helpful to very briefly consider what a qubit actually is.[1]

A Quick Look at a Physical Qubit

One object that readily demonstrates quantum superposition is a single photon. To illustrate this, let's take a step back and suppose we tried to use the *location* of a photon to represent a conventional digital *bit*. In the device shown in Figure 2-1, a switchable mirror (that can be set as either reflective or transparent) allows us to control whether a photon ends up in one of two paths—corresponding to an encoding of either 0 or 1.

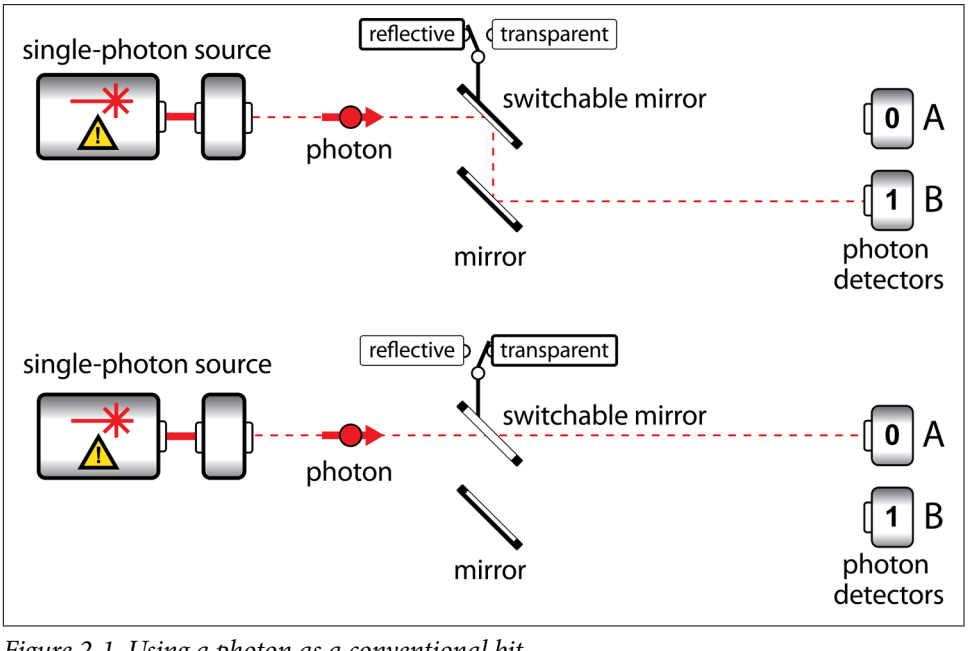


Figure 2-1. Using a photon as a conventional bit

1 In this book we'll try very hard *not* to think too much about what qubits really are. Although this may seem a little anticlimactic, it's worth remembering that conventional programming guides almost never revel in the fascinating physical nature of bits and bytes. In fact, it's precisely the ability to abstract away the physical nature of information that makes the writing of complex programs tractable.

Devices like this actually exist in digital communication technology, but nevertheless a single photon clearly makes a very fiddly bit (for starters, it won't stay in any one place for very long). To use this setup to demonstrate some qubit properties, suppose we replace the switch we use to *set* the photon as 0 or 1 with a *half*-silvered mirror.

A half-silvered mirror, as shown in Figure 2-2 (also known as a *beamsplitter*), is a semireflective surface that would, with a 50% chance, either deflect light into the path we associate with 1, or allow it to pass straight through to the path we associate with 0. There are no other options.

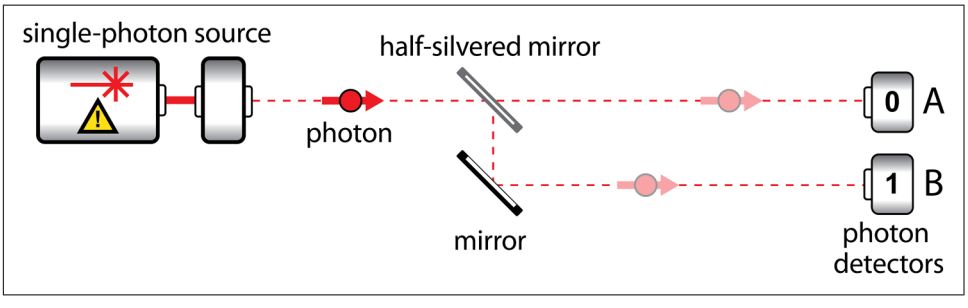


Figure 2-2. A simple implementation of one photonic qubit

When a single indivisible photon hits this surface, it suffers a sort of identity crisis. In an effect that has no conventional equivalent, it ends up existing in a state where it can be influenced by effects in both the 0 path and the 1 path. We say that the photon is in a *superposition* of traveling in each possible path. In other words, we no longer have a conventional bit, but a *qubit* that can be in a superposition of values 0 and 1.

It's very easy to misunderstand the nature of superposition (as many popular quantum computing articles do). It's *not* correct to say that the photon is in both the 0 *and* 1 paths at the same time. There is only one photon, so if we put detectors in each path, as shown in Figure 2-2, only one will go off. When this happens, it will reduce the photon's superposition into a digital bit and give a definitive 0 *or* 1 result. Yet, as we'll explore shortly, there are computationally useful ways a QPU can interact with a qubit in superposition before we need to read it out through such a detection.

The kind of superposition shown in Figure 2-2 will be central to leveraging the quantum power of a QPU. As such, we'll need to describe and control quantum superpositions a little more quantitatively. When our photon is in a superposition of paths, we say it has an *amplitude* associated with each path. There are two important aspects to these amplitudes—two *knobs* we can twiddle to alter the particular configuration of a qubit's superposition:

- The *magnitude* associated with each path of the photon's superposition is an analog value that measures how much the photon has *spread* into each path. A path's magnitude is related to the probability that the photon will be detected in that

path. Specifically, the *square* of the magnitude determines the chance we observe a photon in a given path. In Figure 2-2 we could twiddle the magnitudes of the amplitudes associated with each path by altering how reflective the half-silvered mirror is.

- The *relative phase* between the different paths in the photon's superposition captures the amount by which the photon is *delayed* on one path relative to the other. This is also an analog value that can be controlled by the difference between how far the photon travels in the paths corresponding to 0 and 1. Note that we could change the relative phase without affecting the chance of the photon being detected in each path.[2]

It's worth re-emphasizing that the term *amplitude* is a way of referring to *both* the magnitude and the relative phase associated with some value from a qubit's superposition.

For the mathematically inclined, the amplitudes associated with different paths in a superposition are generally described by *complex numbers*. The *magnitude* associated with an amplitude is precisely the modulus of this complex number (the square root of the number multipled by its complex conjugate), while its *relative phase* is the angle if the complex number is expressed in polar form. For the mathematically uninclined, we will shortly introduce a visual notation so that you need not worry about such complex issues (pun intended).

The magnitude and relative phase are values available for us to exploit when computing, and we can think of them as being *encoded* in our qubit. But if we're ever to read out any information from it, the photon must eventually strike some kind of detector. At this point both these analog values vanish—the quantumness of the qubit is gone. Herein lies the crux of quantum computing: finding a way to exploit these ethereal quantities such that some useful remnant persists after the destructive act of readout.

The setup in Figure 2-2 is equivalent to the code sample we will shortly introduce in Example 2-1, in the case where photons are used as qubits.

2 Although we introduce the idea of relative phase in terms of relative distances traveled by light, it is a general concept that applies to all flavors of qubits: photons, electrons, superconductors, etc.

Okay, enough with the photons! This is a programmer's guide, not a physics textbook. Let's abstract away the physics and see how we can describe and visualize qubits in a manner as detached from photons and quantum physics as binary logic is from electrons and semiconductor physics.

Introducing Circle Notation

We now have an idea of what superposition is, but one that's quite tied up with the specific behavior of photons. Let's find an abstract way to describe superposition that allows us to focus only on abstract information.

The full-blown mathematics of quantum physics provides such an abstraction, but as can be seen in the lefthand column of Table 2-2, this mathematics is far more unintuitive and inconvenient than the simple binary logic of conventional bits.

Fortunately, the equivalent pictorial *circle notation* in the righthand column of Table 2-2 offers a more intuitive approach. Since our goal is building a fluent and pragmatic understanding of what goes on inside a QPU without needing to entrench ourselves in opaque mathematics, from now on we'll think of qubits entirely in terms of this circle notation.

From experimenting with photons we've seen that there are two aspects of a qubit's general state that we need to keep track of in a QPU: the magnitude of its superposition amplitudes and the relative phase between them. Circle notation displays these parameters as follows:

- The *magnitude* of the amplitude associated with each value a qubit can assume (so far, $|0\rangle$ and $|1\rangle$) is related to the radius of the filled-in area shown for each of the $|0\rangle$ or $|1\rangle$ circles.
- The *relative phase* between the amplitudes of these values is indicated by the *rotation* of the $|1\rangle$ circle relative to the $|0\rangle$ circle (a darker line is drawn in the circles to make this rotation apparent).

We'll be relying on circle notation throughout the book, so it's worth taking a little more care to see precisely how circle sizes and rotations capture these concepts.

Circle Size

We previously noted that the *square* of the magnitude associated with |0> or |1> determines the probability of obtaining that value on readout. Since a circle's filled *radius* represents the *magnitude*, this means that the shaded *area* in each circle (or, more colloquially, its size) is directly proportional to the *probability* of obtaining that circle's value (0 or 1) if we read out the qubit. The examples in Figure 2-3 show the

circle notation for different qubit states and the chance of reading out a 1 in each case.

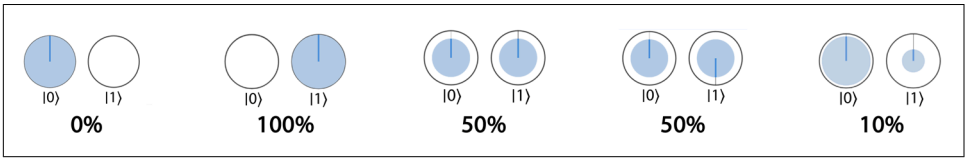


Figure 2-3. Probability of reading the value 1 for different superpositions represented in circle notation

Reading a qubit destroys information. In all of the cases illustrated in Figure 2-3, reading the qubit will produce either a 0 or a 1, and when that happens, the qubit will change its state to match the observed value. So even if a qubit was initially in a more sophisticated state, once you read out a 1 you'll always get a 1 if you immediately try to read it again.

Notice that as the area shaded in the $|0\rangle$ circle gets larger, there's more chance you'll read out a 0, and of course that means that the chance of getting a 1 outcome decreases (being whatever is left over). In the last example in Figure 2-3, there is a 90% chance of reading out the qubit as 0, and therefore a corresponding 10% chance of reading out a 1.[3] We'll often talk of the filled-area of a circle in our circle notation as representing the magnitude associated with that value in a superposition. Although it might seem like an annoying technicality, it's important to have in the back of your mind that the magnitude associated with that value really corresponds to the circle's *radius*—although often it won't hurt to equate the two for visual convenience.

It's also easy to forget, although important to remember, that in circle notation the size of a circle associated with a given outcome does *not* represent the full superposition *amplitude*. The important additional information that we're missing is the relative phase of our superposition.

Circle Rotation

Some QPU instructions will also allow us to alter the relative rotations of a qubit's $|0\rangle$ and $|1\rangle$ circles. This represents the *relative phase* of the qubit. The relative phase of a qubit's state can take any value from 0° to 360°; a few examples are shown in Figure 2-4.

3 The sum of the register's magnitudes must *always* sum up to 1. This requirement is called *normalization*, for more on this see "Caveat 2: The requirement for normalized vectors" on page 183.

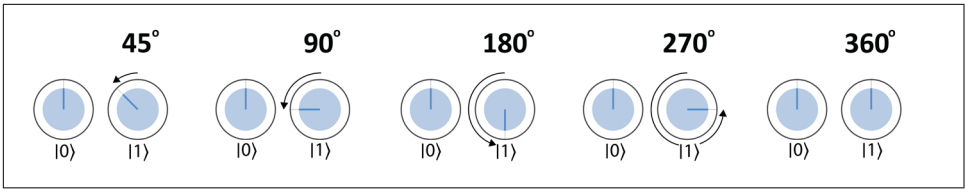


Figure 2-4. Example relative phases in a single qubit

Our convention for rotating the circles in circle notation within this book is that a positive angle rotates the relevant circle *counterclockwise*, as illustrated in Figure 2-4.

In all the preceding examples we have only rotated the $|1\rangle$ circle. Why not the $|0\rangle$ circle as well? As the name suggests, it's only the *relative* phase in a qubit's superposition that ever makes any difference. Consequently only the *relative* rotation between our circles is of interest.[4] If a QPU operation were to apply a rotation to both circles, then we could always equivalently reconsider the effect such that only the $|1\rangle$ circle was rotated, making the relative rotation more readily apparent. An example is shown in Figure 2-5.

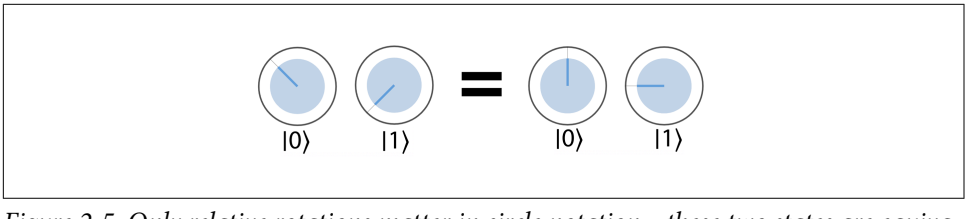


Figure 2-5. Only relative rotations matter in circle notation—these two states are equivalent because the relative phase of the two circles is the same in each case

Note that the relative phase can be varied independently of the magnitude of a superposition. This independence also works the other way. Comparing the third and fourth examples in Figure 2-3, we can see that the relative phase between outcomes for a single qubit has no direct effect on the chances of what we'll read out.

The fact that the relative phase of a single qubit has no effect on the magnitudes in a superposition means that it has no *direct* influence on observable readout results. This may make the relative phase property seem inconsequential, but the truth could not be more different! In quantum computations involving *multiple* qubits, we can

4 That only relative phases are of importance stems from the underlying quantum mechanical laws governing qubits.

crucially take advantage of this rotation to cleverly and indirectly affect the chances that we will eventually read out different values. In fact, well-engineered relative phases can provide an astonishing computational advantage. We'll now introduce operations that will allow us to do this—in particular, those that act only on a single qubit—and we'll visualize their effects using circle notation.

In contrast to the distinctly digital nature of conventional bits, magnitudes and relative phases are *continuous* degrees of freedom. This leads to a widely held misconception that quantum computing is comparable to the ill-fated *analog* computing. Remarkably, despite allowing us to manipulate continuous degrees of freedom, the errors experienced by a QPU can be corrected *digitally*. This is why QPUs are more robust than analog computing devices.

The First Few QPU Operations

Like their CPU counterparts, single-qubit QPU operations transform input information into a desired output. Only now, of course, our inputs and outputs are qubits rather than bits. Many QPU instructions[5] have an associated inverse, which can be useful to know about. In this case a QPU operation is said to be *reversible*, which ultimately means that no information is lost or discarded when it is applied. Some QPU operations, however, are *irreversible* and have no inverse (somehow they result in the loss of information). We'll eventually come to see that whether or not an operation is reversible can have important ramifications for how we make use of it.

Some of these QPU instructions may seem strange and of questionable utility, but after only introducing a handful we'll quickly begin putting them to use.

QPU Instruction: NOT

`NOT` is the quantum equivalent of the eponymous conventional operation. Zero becomes one, and vice versa. However, unlike its traditional cousin, a QPU `NOT` operation can also operate on a qubit in superposition.

In circle notation this results, very simply, in the swapping of the $|0\rangle$ and $|1\rangle$ circles, as in Figure 2-6.

5 Throughout the book we will use the terms *QPU instruction* and *QPU operation* interchangeably.

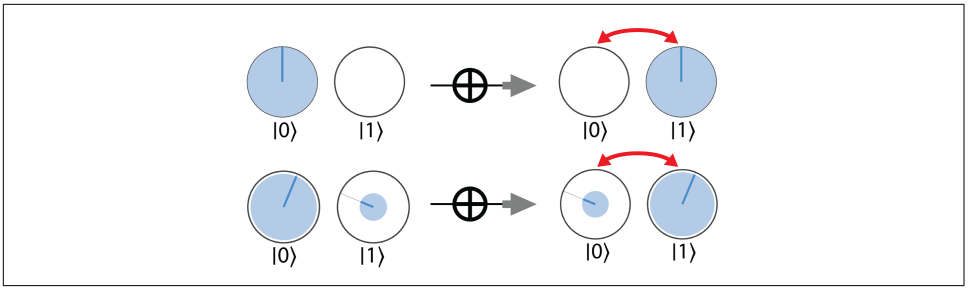


Figure 2-6. The NOT operation in circle notation

Reversibility: Just as in digital logic, the `NOT` operation is its own inverse; applying it twice returns a qubit to its original value.

QPU Instruction: HAD

The `HAD` operation (short for Hadamard) essentially creates an equal superposition when presented with either a $|0\rangle$ or $|1\rangle$ state. This is our gateway drug into using the bizarre and delicate parallelism of quantum superposition! Unlike `NOT`, it has no conventional equivalent.

In circle notation, `HAD` results in the output qubit having the same amount of area filled-in for both $|0\rangle$ and $|1\rangle$, as in Figure 2-7.

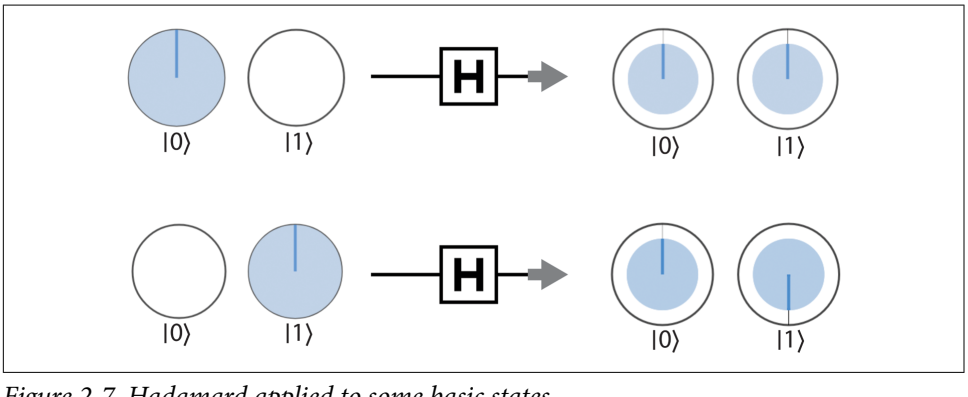


Figure 2-7. Hadamard applied to some basic states

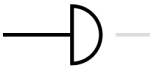
This allows `HAD` to produce *uniform superpositions* of outcomes in a qubit; i.e., a superposition where each outcome is equally likely. Notice also that `HAD`'s action on qubits initially in the states $|0\rangle$ and $|1\rangle$ is slightly different: the output of acting `HAD` on $|1\rangle$ yields a nonzero rotation (relative phase) of one of the circles, whereas the output from acting it on $|0\rangle$ doesn't.

You might wonder what happens if we apply HAD to qubits that are *already in a superposition*. The best way to find out is to experiment! Doing so you'll soon notice that the following occurs:

- HAD acts on both the $|0\rangle$ and $|1\rangle$ states separately according to the rules illustrated in Figure 2-7.
- The $|0\rangle$ and $|1\rangle$ values this generates are combined, weighted by the amplitudes of the original superpositions.[6]

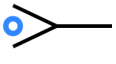
Reversibility: Similar to NOT, the HAD operation is its own inverse; applying it twice returns a qubit to its original value.

QPU Instruction: READ



The READ operation is the formal expression of the previously introduced *readout* process. READ is unique in being the only part of a QPU's instruction set that potentially returns a *random* result.

QPU Instruction: WRITE



The WRITE operation allows us to initialize a QPU register before we operate on it. This is a deterministic process.

Applying READ to a single qubit will return a value of either 0 or 1 with probabilities determined by (the square of) the associated magnitudes in the qubit's state (ignoring the relative phase). Following a READ operation, a qubit is left in the state $|0\rangle$ if the 0 outcome is obtained and state $|1\rangle$ if the 1 outcome is obtained. In other words, any superposition is irreversibly destroyed.

In circle notation an outcome occurs with a probability determined by the filled area in each associated circle. We then shift the filled-in area between circles to reflect this result: the circle associated with the occurring outcome becomes entirely filled in, while the remaining circle becomes empty. This is illustrated in Figure 2-8 for READ operations being performed on two different example superpositions.

6 For details on the mathematics behind HAD and other common operations, see Chapter 14.

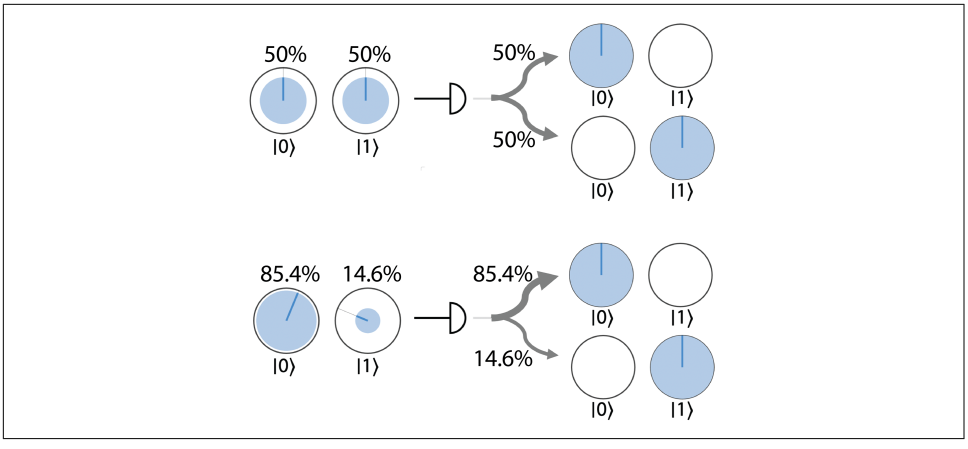


Figure 2-8. The READ operation produces random results

In the second example of Figure 2-8, the `READ` operation removes all meaningful relative phase information. As a result, we reorient the state so that the circle points upward.

Using a `READ` and a `NOT`, we can also construct a simple `WRITE` operation that allows us to prepare a qubit in a desired state of either $|0\rangle$ or $|1\rangle$. First we `READ` the qubit, and then, if the value does not match the value we plan to `WRITE`, we perform a `NOT` operation. Note that this `WRITE` operation does *not* allow us to prepare a qubit in an arbitrary superposition (with arbitrary magnitude and relative phase), but only in either state $|0\rangle$ or state $|1\rangle$.[7]

Reversibility: The `READ` and `WRITE` operations are not reversible. They destroy superpositions and lose information. Once that is done, the analog values of the qubit (both magnitude and phase) are gone forever.

Hands-on: A Perfectly Random Bit

Before moving on to introduce a few more single-qubit operations, let's pause to see how—armed with the `HAD`, `READ`, and `WRITE` operations—we can create a program to perform a task that is impossible on any conventional computer. We'll generate a *truly* random bit.

7 We will see that being able to prepare an arbitrary superposition is tricky but useful, especially in quantum machine-learning applications, and we introduce an approach for this in Chapter 13.

Throughout the history of computation, a vast amount of time and effort has gone into developing Pseudo-Random Number Generator (PRNG) systems, which find usage in applications ranging from cryptography to weather forecasting. PRNGs are *pseudo* in the sense that if you know the contents of the computer's memory and the PRNG algorithm, you can—in principle—predict the next number to be generated.

According to the known laws of physics, the readout behavior of a qubit in superposition is fundamentally and perfectly unpredictable. This allows a QPU to create the world's greatest random number generator by simply preparing a qubit in state $|0\rangle$, applying the `HAD` instruction, and then reading out the qubit. We can illustrate this combination of QPU operations using a *quantum circuit* diagram, where a line moving left to right illustrates the sequence of different operations that are performed on our (single) qubit, as shown in Figure 2-9.

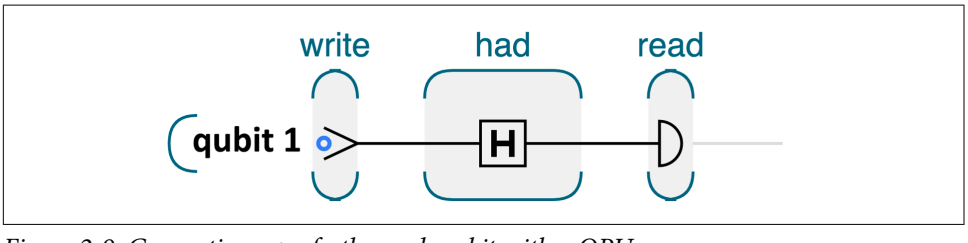


Figure 2-9. Generating a perfectly random bit with a QPU

It might not look like much, but there we have it, our first QPU program: a Quantum Random Number Generator (QRNG)! You can simulate this using the code snippet in Example 2-1. If you repeatedly run these four lines of code on the QCEngine simulator, you'll receive a binary random string. Of course, CPU-powered simulators like QCEngine are approximating our QRNG with a PRNG, but running the equivalent code on a real QPU will produce a perfectly random binary string.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=2-1*.

Example 2-1. One random bit

```
qc.reset(1);          // allocate one qubit
qc.write(0);          // write the value zero
qc.had();             // place it into superposition of 0 and 1
var result = qc.read();  // read the result as a digital bit
```

All of the code samples in this book can be found online at *http://oreilly-qc.github.io*, and can be run either on QPU simulators or on actual QPU hardware. Running these samples is an essential part of learning to program a QPU. For more information, see Chapter 1.

Since this might be your first quantum program (congratulations!), let's break it down just to be sure each step makes sense:

- `qc.reset(1)` sets up our simulation of the QPU, requesting one qubit. All the programs we write for QCEngine will initialize a set of qubits with a line like this.
- `qc.write(0)` simply initializes our single qubit in the $|0\rangle$ state—the equivalent of a conventional bit being set to the value 0.
- `qc.had()` applies `HAD` to our qubit, placing it into a superposition of $|0\rangle$ and $|1\rangle$, just as in Figure 2-7.
- `var result = qc.read()` reads out the value of our qubit at the end of the computation as a random digital bit, assigning the value to the `result` variable.

It might look like all we've really done here is find a very expensive way of flipping a coin, but this underestimates the power of `HAD`. If you could somehow *look inside* `HAD` you would find neither a pseudo nor a hardware random number generator. Unlike these, `HAD` is guaranteed unpredictable by the laws of quantum physics. Nobody in the known universe can do any better than a hopeless random guess as to whether a qubit's value following a `HAD` will be read to be 0 or 1—even if they know exactly the instructions we are using to generate our random numbers.

In fact, although we'll properly introduce dealing with multiple qubits in the next chapter, we can easily run our single random qubit program in parallel eight times to produce a random *byte*. Figure 2-10 shows what this looks like.

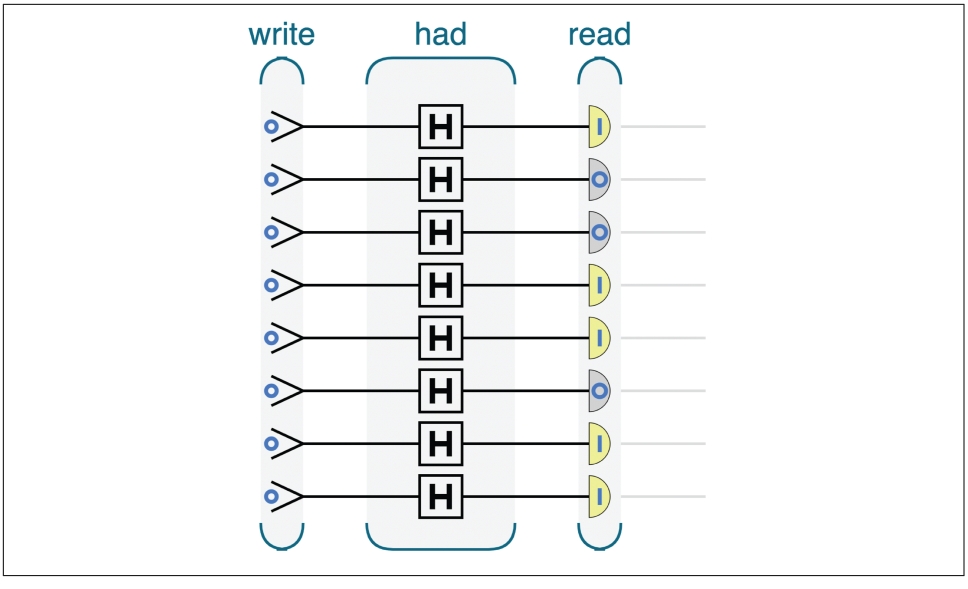


Figure 2-10. Generating one random byte

This code in Example 2-2 for creating a random byte is almost identical to Example 2-1.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=2-2*.

Example 2-2. One random byte

```
qc.reset(8);
qc.write(0);
qc.had();
var result = qc.read();
qc.print(result);
```

Note that we make use of the fact that QCEngine operations like `WRITE` and `HAD` default to acting on all initialized qubits, unless we explicitly pass specific qubits for them to act on.

Although Example 2-2 uses multiple qubits, there are no actual multi-qubit operations that take more than one of the qubits as input. The same program could be serialized to run on only a single qubit.

QPU Instruction: PHASE(θ)

The `PHASE(θ)` operation also has no conventional equivalent. This instruction allows us to directly manipulate the *relative phase* of a qubit, changing it by some specified angle. Consequently, as well as a qubit to operate on, the `PHASE(θ)` operation takes an additional (numerical) input parameter—the angle to rotate by. For example, `PHASE(45)` denotes a `PHASE` operation that performs a 45° rotation.

In circle notation, the effect of `PHASE(θ)` is to simply rotate the circle associated with $|1\rangle$ by the angle we specify. This is shown in Figure 2-11 for the case of `PHASE(45)`.

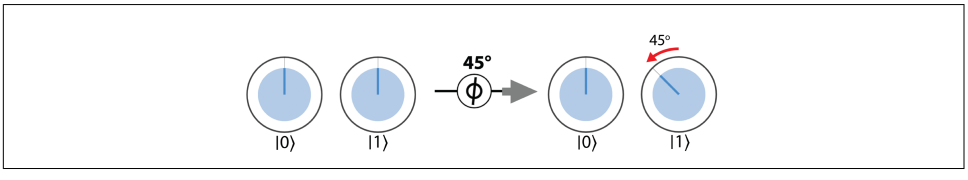


Figure 2-11. Action of a PHASE(45) operation

Note that the `PHASE` operation only rotates the circle associated with the $|1\rangle$ state, so it would have no effect on a qubit in the $|0\rangle$ state.

Reversibility: `PHASE` operations are reversible, although they are not generally their own inverse. The `PHASE` operation may be reversed by applying a `PHASE` with the *negative* of the original angle. In circle notation, this corresponds to *undoing* the rotation, by rotating in the opposite direction.

Using `HAD` and `PHASE`, we can produce some single-qubit quantum states that are so commonly used that they've been named: $|+\rangle$, $|-\rangle$, $|+Y\rangle$, and $|-Y\rangle$, as shown in Figure 2-12. If you feel like flexing your QPU muscles, see whether you can determine how to produce these states using `HAD` and `PHASE` operations (each superposition shown has an equal *magnitude* in each of the $|0\rangle$ and $|1\rangle$ states).

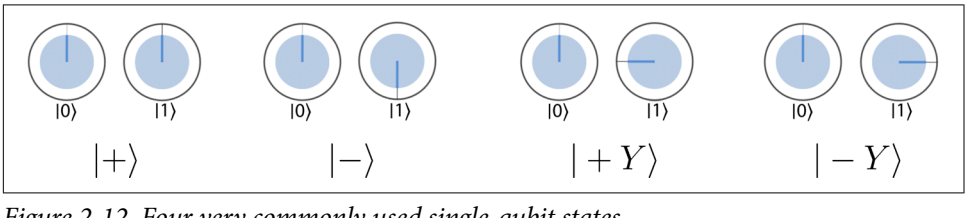


Figure 2-12. Four very commonly used single-qubit states

These four states will be used in Example 2-4, and although one way to produce them is using `HAD` and `PHASE`, we can also understand them as being the result of so-called single-qubit *rotation* operations.

QPU Instructions: ROTX(θ) and ROTY(θ)

We've seen that `PHASE` rotates the relative phase of a qubit, and that in circle notation this corresponds to rotating the circle associated with the $|1\rangle$ value. There are two other common operations related to `PHASE` called `ROTX(`θ`)` and `ROTY(`θ`)`, which also perform slightly different kinds of rotations on our qubit.

Figure 2-13 shows the application of `ROTX(45)` and `ROTY(45)` on the $|0\rangle$ and $|1\rangle$ states in circle notation.

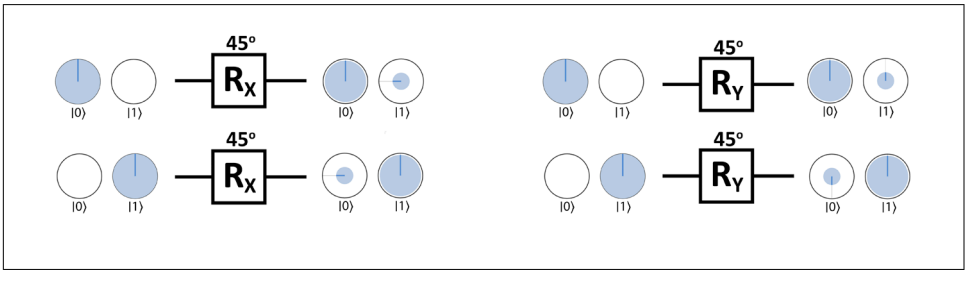


Figure 2-13. ROTX and ROTY actions on 0 and 1 input states

These operations don't *look* like very intuitive rotations, at least not as obviously as `PHASE` did. However, their rotation names stem from their action in another common visual representation of a single qubit's state, known as the *Bloch sphere*. In the Bloch sphere representation, a qubit is visualized by a point somewhere on the surface of a three-dimensional sphere. In this book we use circle notation instead of the Bloch sphere visualization, as the Bloch sphere doesn't extend well to multiple qubits. But to satisfy any etymological curiosity, if we represent a qubit on the Bloch sphere, then `ROTY` and `ROTX` operations correspond to rotating the qubit's point about the sphere's y- and x-axes, respectively. This meaning is lost in our circle notation, since we use two 2D circles rather than a single three-dimensional sphere. In fact, the `PHASE` operation actually corresponds to a rotation about the z-axis when visualizing qubits in the Bloch sphere, so you may also hear it referred to as `ROTZ`.

COPY: The Missing Operation

There is one operation available to conventional computers that *cannot* be implemented on a QPU. Although we can make many copies of a *known* state by repeatedly preparing it (if the state is either $|0\rangle$ or $|1\rangle$, we can do this simply with `WRITE` operations), there is no way of copying some state partway through a quantum computation without determining what it is. This constraint arises due to the fundamental laws of physics governing our qubits.

This is definitely an inconvenience, but as we will learn in the following chapters, other possibilities available to QPUs can help make up for the lack of a `COPY` instruction.

Combining QPU Operations

We now have `NOT`, `HAD`, `PHASE`, `READ`, and `WRITE` at our disposal. It's worth mentioning that, as is the case in conventional logic, these operations can be combined to realize each other, and even allow us to create entirely new operations. For example, suppose a QPU provides the `HAD` and `PHASE` instructions, but `NOT` is missing. A `PHASE(180)` operation can be combined with two `HAD`s to produce the exact equivalent of a `NOT` operation, as shown in Figure 2-14. Conversely, a `PHASE(180)` instruction can also be realized from `HAD` and `NOT` operations.

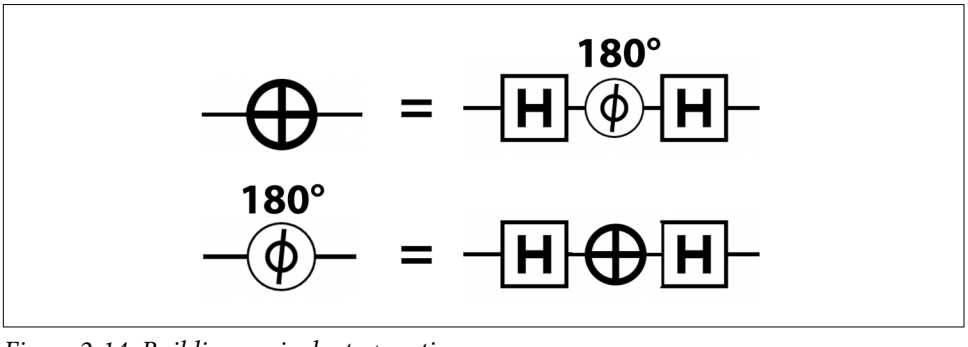


Figure 2-14. Building equivalent operations

QPU Instruction: ROOT-of-NOT

Combining instructions also lets us produce interesting new operations that do not exist at all in the world of conventional logic. The ROOT-of-NOT operation (`RNOT`) is one such example. It's quite literally the square root of the `NOT` operation, in the sense that, when applied twice, it performs a single `NOT`, as shown in Figure 2-15.

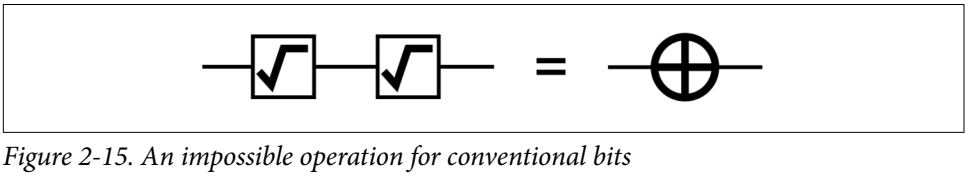


Figure 2-15. An impossible operation for conventional bits

There's more than one way to construct this operation, but Figure 2-16 shows one simple implementation.

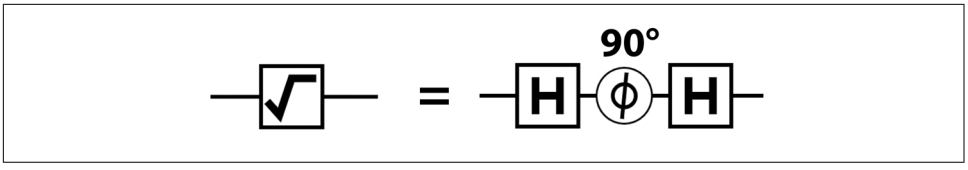


Figure 2-16. Recipe for ROOT-of-NOT

We can check that applying this set of operations twice does indeed yield the same result as a `NOT` by running a simulation, as shown in Example 2-3.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=2-3*.

Example 2-3. Showing the action of RNOT

```
qc.reset(1);
qc.write(0);

// Root-of-not
qc.had();
qc.phase(90);
qc.had();

// Root-of-not
qc.had();
qc.phase(90);
qc.had();
```

In circle notation, we can visualize each step involved in implementing an `RNOT` operation (a `PHASE(90)` between two `HADs`). The resulting operation is shown in Figure 2-17.

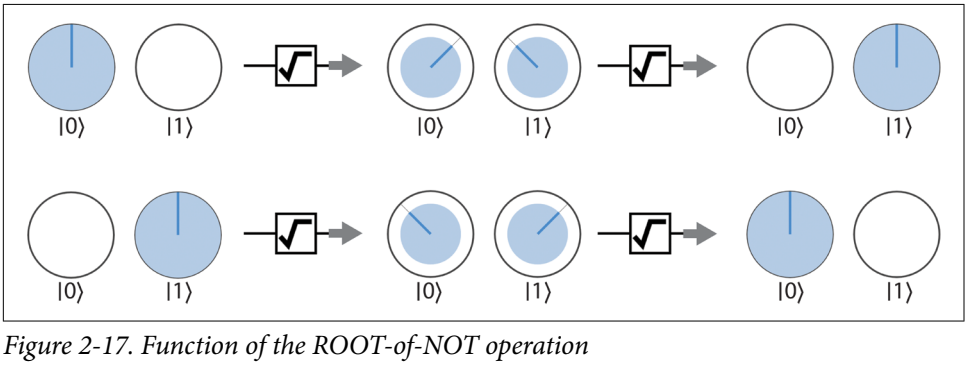


Figure 2-17. Function of the ROOT-of-NOT operation

Following the evolution of our qubit in circle notation helps us see how `RNOT` is able to get us halfway to a `NOT` operation. Recall from Figure 2-14 that if we `HAD` a qubit, then rotate its relative phase by 180°, another `HAD` will result in a `NOT` operation. `RNOT` performs half of this rotation (a `PHASE(90)`), so that two applications will result in the `HAD-PHASE(180)-HAD` sequence that is equivalent to a `NOT`. It might be a bit mind-bending at first, but see if you can piece together how the `RNOT` operation cleverly performs this feat when applied twice (it might help to remember that `HAD` is its own inverse, so a sequence of two `HAD`s is equivalent to doing nothing at all).

Reversibility: While `RNOT` operations are never their own inverse, the inverse of the operation in Figure 2-16 may be constructed by using a negative phase value, as shown in Figure 2-18.

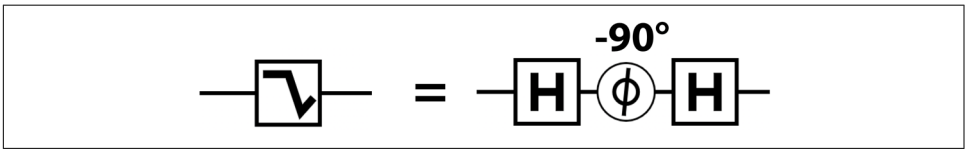


Figure 2-18. Inverse of RNOT

Although it might seem like an esoteric curiosity, the `RNOT` operation teaches us the important lesson that by carefully placing information in the relative phase of a qubit, we can perform entirely new kinds of computation.

Hands-on: Quantum Spy Hunter

For a more practical demonstration of the power in manipulating the relative phases of qubits, we finish this chapter with a more complex program. The code presented in Example 2-4 uses the simple single-qubit QPU operations introduced previously to perform a simplified version of Quantum Key Distribution (QKD). QKD is a protocol at the core of the field of quantum cryptography that allows the provably secure transmission of information.

Suppose that two QPU programmers, Alice and Bob, are sending data to each other via a communication channel capable of transmitting qubits. Once in a while, they send the specially constructed "spy hunter" qubit described in Example 2-4, which they use to test whether their communication channel has been compromised.

Any spy who tries to read one of these qubits has a 25% chance of getting caught. So even if Alice and Bob only use 50 of them in the whole transfer, the spy's chances of getting away are far less than one in a million.

Alice and Bob can detect whether their key has been compromised by exchanging some conventional digital information, which does not need to be private or encrypted. After exchanging their messages, they test a few of their qubits by reading them

out and checking that they agree in a certain expected way. If any disagree, then they know someone was listening in. This process is illustrated in Figure 2-19.

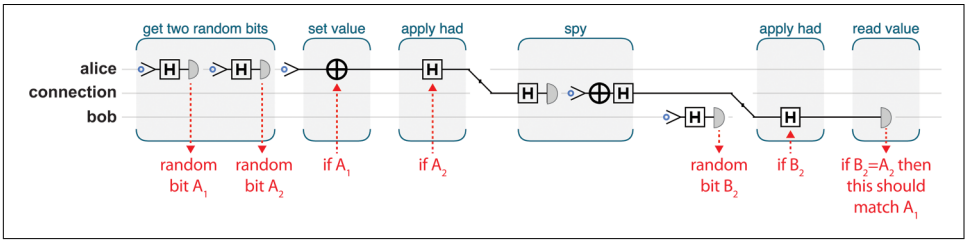


Figure 2-19. The quantum spy hunter program

Here's the code. We recommend trying out Example 2-4 on your own, and tweaking and testing like you would with any other code snippet.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=2-4*.

Example 2-4. Quantum random spy hunter

```
qc.reset(3);
qc.discard();
var a = qint.new(1, 'alice');
var fiber = qint.new(1, 'fiber');
var b = qint.new(1, 'bob');

function random_bit(q) {
    q.write(0);
    q.had();
    return q.read();
}

// Generate two random bits
var send_had = random_bit(a);
var send_val = random_bit(a);

// Prepare Alice's qubit
a.write(0);
if (send_val)  // Use a random bit to set the value
    a.not();
if (send_had)  // Use a random bit to apply HAD or not
    a.had();

// Send the qubit!
fiber.exchange(a);

// Activate the spy
var spy_is_present = true;
```

```
if (spy_is_present) {
    var spy_had = 0;
    if (spy_had)
        fiber.had();
    var stolen_data = fiber.read();
    fiber.write(stolen_data);
    if (spy_had)
        fiber.had();
}

// Receive the qubit!
var recv_had = random_bit(b);
fiber.exchange(b);
if (recv_had)
    b.had();
var recv_val = b.read();

// Now Alice emails Bob to tell
// him her choice of operations and value.
// If the choice matches and the
// value does not, there's a spy!
if (send_had == recv_had)
    if (send_val != recv_val)
        qc.print('Caught a spy!\n');
```

In Example 2-4, Alice and Bob each have access to a simple QPU containing a single qubit, and can send their qubits along a quantum communication channel. There might be a spy listening to that link; in the sample code you can control whether or not a spy is present by toggling the `spy_is_present` variable.

The fact that quantum cryptography can be performed with such relatively small QPUs is one of the reasons why it has begun to see commercial application long before more powerful general-purpose QPUs are available.

Let's walk through the code one step at a time to see how Alice and Bob's simple resources allow them to perform this feat. We'll refer to comments from the code snippet as markers:

`// Generate two random bits`
: Alice uses her one-qubit QPU as a simple QRNG, exactly as we did in Example 2-2, generating two secret random bits known only to her. We denote these `send_val` and `send_had`.

`// Prepare Alice's qubit`

Using her two random bits, Alice prepares the "spy hunter" qubit. She sets it to `value`, and then uses `send_had` to decide whether to apply a `HAD`. In effect, she is preparing her qubit randomly in one of the states $|0\rangle$, $|1\rangle$, $|+\rangle$, or $|-\rangle$, and not (yet) telling anyone *which* of the states it is. If she does decide to apply a `HAD`, then if Bob wants to extract whether she intended a 0 or 1, he will have to apply the inverse of `HAD` (another `HAD`) before performing a `READ`.

`// Send the qubit!`

Alice sends her qubit to Bob. For clarity in this example, we are using another qubit to represent the communication channel.

`// Activate the spy`

If Alice were transmitting *conventional* digital data, the spy would simply make a copy of the bit, accomplishing their mission. With qubits, that's not possible. Recall that there is no `COPY` operation, so the only thing the spy can do is `READ` the qubit Alice sent, and then try to carefully send one just like it to Bob to avoid detection. Remember, however, that reading a qubit irrevocably destroys information, so the spy will only be left with the conventional bit of the readout. The spy doesn't know whether or not Alice performed a `HAD`. As a result, he won't know whether to apply a second (*inverting*) `HAD` before performing his `READ`. If he simply performs a `READ` he won't know whether he's receiving a random value from a qubit in superposition or a value actually encoded by Alice. This means that not only will he not be able to reliably extract Alice's bit, but he also won't know what the right state is to send on to Bob to avoid detection.

`// Receive the qubit!`

Like Alice, Bob randomly generates a `recv_had` bit, and he uses that to decide whether to apply a `HAD` before applying a `READ` to Alice's qubit, resulting in his value bit. This means that sometimes Bob will (by chance) correctly decode a binary value from Alice and other times he won't.

`// If the had setting matches between sender and receiver but the val does not, there's a spy!`

Now that the qubit has been received, Alice and Bob can openly compare the cases in which their choices of applying `HAD`s (or not) correctly matched up. If they randomly happened to agree in both applying (or not applying) a `HAD` (this will be about half the time), their value bits should match; i.e., Bob will have correctly decoded Alice's message. If in these *correctly decoded messages* their values *don't* agree, they can conclude that the spy must have `READ` their message and sent on an *incorrect* replacement qubit to Bob, messing up his decoding.

Conclusion

In this chapter we introduced a way to describe single qubits, as well as a variety of QPU instructions to manipulate them. The random property of the `READ` operation was used to construct a quantum random number generator, and control over the relative phase in a qubit was used to perform basic quantum cryptography.

The circle notation used to visualize the state of a qubit is also used extensively in the chapters ahead. In Chapter 3, we will extend circle notation to deal with the behavior of multi-qubit systems, and introduce new QPU operations used to work with them.

CHAPTER 3
Multiple Qubits

As useful as single qubits can be, they're much more powerful (and intriguing) in groups. We've already seen in Chapter 2 how the distinctly quantum phenomenon of superposition introduces the new parameters of magnitude and relative phase for computation. When our QPU has access to more than one qubit, we can make use of a second powerful quantum phenomenon known as *entanglement*. Quantum entanglement is a very particular kind of interaction between qubits, and we'll see it in action within this chapter, utilizing it in complex and sophisticated ways.

But to explore the abilities of multiple qubits, we first need a way to visualize them.

Circle Notation for Multi-Qubit Registers

Can we extend our circle notation to multiple qubits? If our qubits *didn't* interact with one another, we could simply employ multiple versions of the representation we used for a single qubit. In other words, we could use a pair of circles for the $|0\rangle$ and $|1\rangle$ states of each qubit. Although this naive representation allows us to describe a superposition of any one *individual* qubit, there are superpositions of *groups* of qubits that it cannot represent.

How else might circle notation represent the state of a *register* of multiple qubits? Just as is the case with conventional bits, a register of N qubits can be used to represent one of 2^N different values. For example, a register of three qubits in the states $|0\rangle|1\rangle|1\rangle$ can represent a decimal value of 3. When talking about multi-qubit registers we'll often describe the decimal value that the register represents in the same quantum notation that we used for a single qubit, so whereas a single qubit can encode the states $|0\rangle$ and $|1\rangle$, a two-qubit register can encode the states $|0\rangle$, $|1\rangle$, $|2\rangle$, and $|3\rangle$. Making use of the quantum nature of our qubits, we can also create superpositions of these different values. To represent these kinds of superpositions of N qubits, we'll use

a separate circle for each of the 2^N different values that an N-bit number can assume, as shown in Figure 3-1.

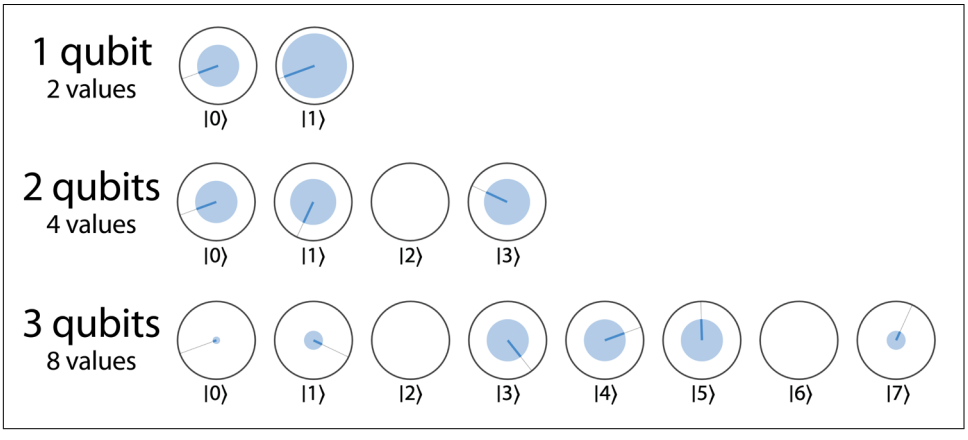


Figure 3-1. Circle notation for various numbers of qubits

In Figure 3-1, we see the familiar two-circle $|0\rangle$, $|1\rangle$ representation for a single qubit. For two qubits we have circles for $|0\rangle$, $|1\rangle$, $|2\rangle$, $|3\rangle$. This is not "one pair of circles per qubit"; instead, it is one circle for each possible two-bit number you may get by reading these qubits. For three qubits, the values of the QPU register are $|0\rangle$, $|1\rangle$, $|2\rangle$, $|3\rangle$, $|4\rangle$, $|5\rangle$, $|6\rangle$, $|7\rangle$, since upon readout we can get any three-bit value. In Figure 3-1, this means that we can now associate a magnitude and relative phase with each of these 2^N values. In the case of the three-qubit example, the magnitude of each of the circles determines the probability that a specific three-bit value will be observed when *all three* qubits are read.

You may be wondering what such a superposition of a multi-qubit register's value looks like in terms of the states of the individual qubits making it up. In some cases we can easily deduce the individual qubit states. For example, the three-qubit register superposition of the states $|0\rangle$, $|2\rangle$, $|4\rangle$, $|6\rangle$ shown in Figure 3-2 can easily be expressed in terms of each individual qubit's state.

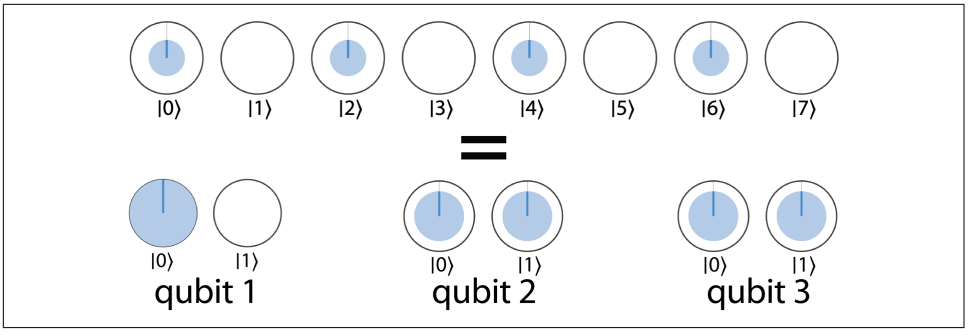


Figure 3-2. Some multi-qubit quantum states can be understood in terms of single-qubit states

To convince yourself that these single-qubit and multi-qubit representations are equivalent, write down each decimal value in the multi-qubit state in terms of the three-bit binary values. In fact, this multi-qubit state can be generated simply using two single-qubit HAD operations, as shown by Example 3-1.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=3-1*.

Example 3-1. Creating a multi-qubit state that can be expressed in terms of its qubits

```
qc.reset(3);
qc.write(0);
var qubit1 = qint.new(1, 'qubit 1');
var qubit2 = qint.new(1, 'qubit 2');
var qubit3 = qint.new(1, 'qubit 3');
qubit2.had();
qubit3.had();
```

Example 3-1 introduces some new QCEngine notation for keeping track of larger numbers of qubits. The `qint` object allows us to label our qubits and treat them more like a standard programming variable. Once we've used `qc.reset()` to set up our register with some qubits, `qint.new()` allows us to assign them to `qint` objects. The first argument to `qint.new()` specifies how many qubits to assign to this `qint` from the stack created by `qc.reset()`. The second argument takes a label that is used in the circuit visualizer. `qint` objects have many methods allowing us to apply QPU operations directly to qubit groupings. In Example 3-1, we use `qint.had()`.

Although the state in Figure 3-2 can be understood in terms of its constituent qubits, take a look at the three-qubit register state shown in Figure 3-3.

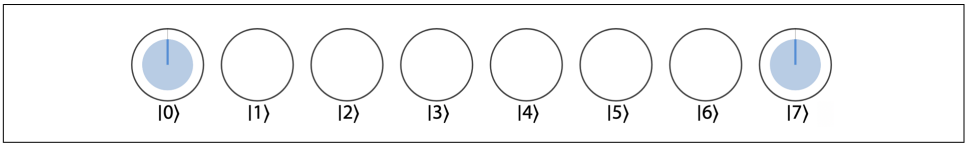


Figure 3-3. Quantum relationships between multiple qubits

This represents a state of three qubits in equal superposition of $|0\rangle$ and $|7\rangle$. Can we visualize this in terms of what each individual qubit is doing like we could in Figure 3-2? Since 0 and 7 are 000 and 111 in binary, we have a superposition of the three qubits being in the states $|0\rangle|0\rangle|0\rangle$ and $|1\rangle|1\rangle|1\rangle$. Surprisingly, in this case, there is no way to write down circle representations for the individual qubits! Notice that reading out the three qubits always results in us finding them to have the *same* values (with 50% probability that the value will be 0 and 50% probability it will be 1). So clearly there must be some kind of link between the three qubits, ensuring that their outcomes are the same.

This link is the new and powerful *entanglement* phenomenon. Entangled multi-qubit states cannot be described in terms of individual descriptions of what the constituent qubits are doing, although you're welcome to try! This entanglement link is only describable in the configuration of the *whole* multi-qubit register. It also turns out to be impossible to produce entangled states from only *single-qubit* operations. To explore entanglement in more detail, we'll need to introduce multi-qubit operations.

Drawing a Multi-Qubit Register

We now know how to describe the configuration of N qubits in circle notation using 2^N circles, but how do we draw multi-qubit quantum circuits? Our multi-qubit circle notation considers each qubit to take a position in a length N bitstring, so it's convenient to label each qubit according to its binary value.

For example, let's take another look at the random eight-qubyte circuit we introduced in Chapter 2. We can collectively refer to a register of eight qubits as a *qubyte* in analogy with a conventional eight-bit byte. In our previous encounter with a qubyte, we simply labeled the eight qubits as `qubit 1`, `qubit 2`, etc. Figure 3-4 shows how that circuit looks if we properly label each qubit with the binary value it represents.

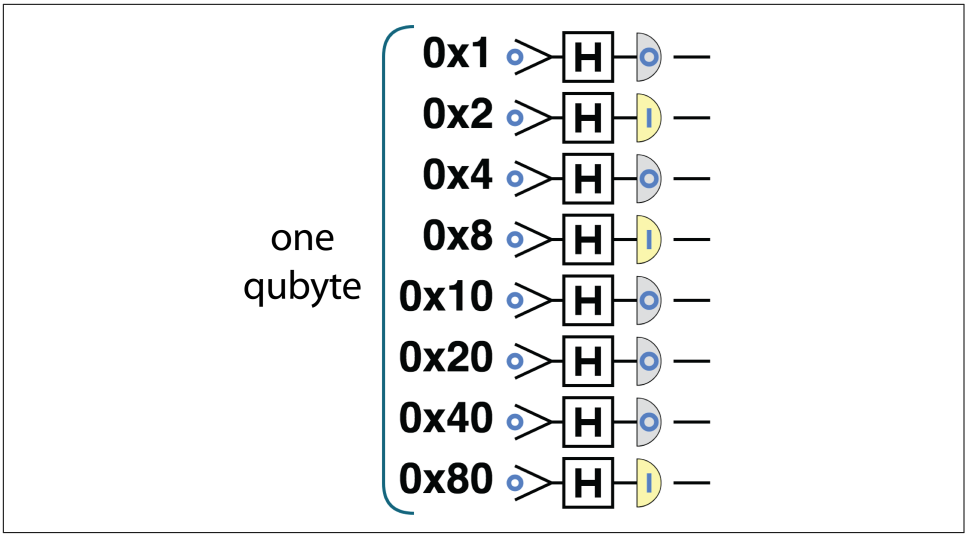


Figure 3-4. Labeling qubits in a qubyte

Naming Qubits

In Figure 3-4 we write our qubit values in hexadecimal, using notation such as `0x1` and `0x2`. This is standard programmer notation for hexadecimal values, and we'll use this throughout the book as a convenient notation to clarify when we're talking about a specific qubit—even in cases when we have large numbers of them.

Single-Qubit Operations in Multi-Qubit Registers

Now that we're able to draw multi-qubit circuits and represent them in circle notation, let's start making use of them. What happens (in circle notation) when we apply single-qubit operations such as `NOT`, `HAD`, and `PHASE` to a multi-qubit register? The only difference from the single-qubit case is that the circles are operated on in certain *operator pairs* specific to the qubit that the operation acts on.

To identify a qubit's operator pairs, match each circle with the one whose value differs by the qubit's bit-value, as shown in Figure 3-5. For example, if we are operating on qubit `0x4`, then each pair will include circles whose values differ by exactly 4.

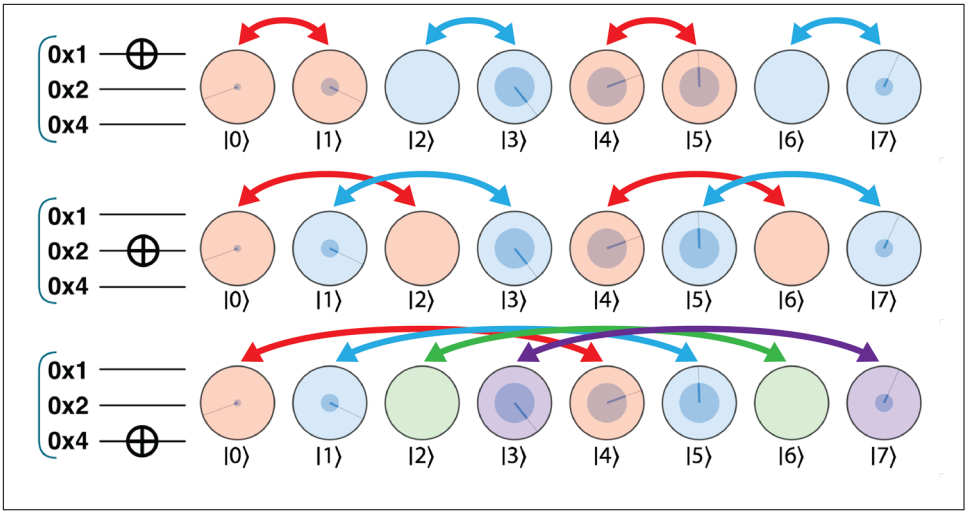


Figure 3-5. The NOT operation swaps values in each of the qubit's operator pairs; here, its action is shown on an example multi-qubit superposition

Once these operator pairs have been identified, the operation is performed on *each pair*, just as if the members of a pair were the $|0\rangle$ and $|1\rangle$ values of a single-qubit register. For a `NOT` operation, the circles in each pair are simply swapped, as in Figure 3-5.

For a single-qubit `PHASE` operation, the righthand circle of each pair is rotated by the phase angle, as in Figure 3-6.

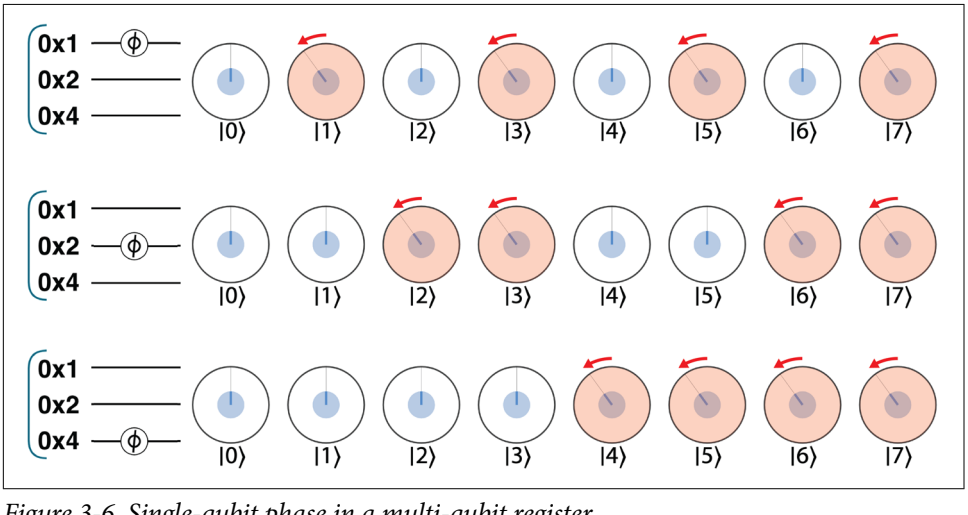


Figure 3-6. Single-qubit phase in a multi-qubit register

Thinking in terms of operator pairs is a good way to quickly visualize the action of single-qubit operations on a register. For a deeper understanding of why this works, we need to think about the effect that an operation on a given qubit has on the binary representation of the whole register. For example, the circle-swapping action of a `NOT` on the second qubit in Figure 3-5 corresponds to simply flipping the second bit in each value's binary representation. Similarly, a single-qubit `PHASE` operation acting on (for example) the third qubit rotates each circle for which the third bit is 1. A single-qubit `PHASE` will always cause exactly half of the values of the register to be rotated, and *which* half just depends on which qubit is the target of the operation.

The same kind of reasoning helps us think about the action of any other single-qubit operation on qubits from larger registers.

Occasionally we'll colorfully highlight certain circles in circle notation, such as in Figures 3-5 and 3-6. This is only to highlight which states have been involved in operations.

Reading a Qubit in a Multi-Qubit Register

What happens when we perform a `READ` operation on a single qubit from a multi-qubit register? `READ` operations also function using operator pairs. If we have a multi-qubit circle representation, we can determine the probability of obtaining a 0 outcome for one single qubit by adding the squared magnitudes of all of the circles on the $|0\rangle$ (lefthand) side of that qubit's operator pairs. Similarly, we can determine the probability of 1 by adding the squared magnitudes of all of the circles on the $|1\rangle$ (righthand) side of the qubit's operator pairs.

Following a `READ`, the state of our multi-qubit register will change to reflect which outcome occurred. All circles that do not agree with the result will be eliminated, as shown in Figure 3-7.

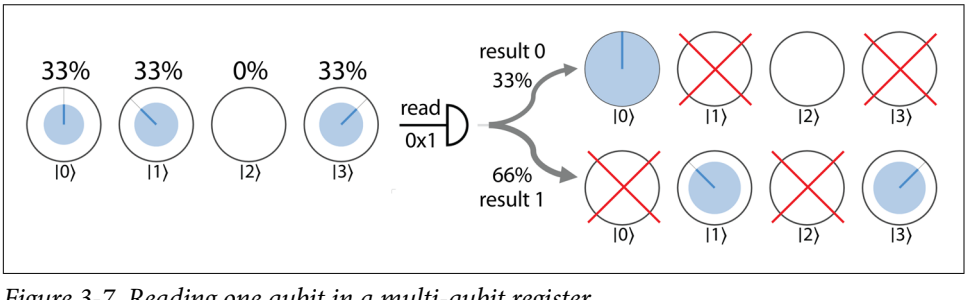


Figure 3-7. Reading one qubit in a multi-qubit register

Note that both states $|1\rangle$ and $|3\rangle$ are compatible with reading an outcome of 1 in the first (`0x1`) qubit. This is because the binary representations of 1 and 3 both contain a 1 in the first bit. Note also that following this elimination, the state has the remaining values *renormalized* so that their areas (and hence the associated probabilities) add up to 100%. To read more than one qubit, each single-qubit `READ` operation can be performed individually according to the operator pair prescription.

Visualizing Larger Numbers of Qubits

N qubits require 2^N circles in circle notation, and therefore each additional qubit we might add to our QPU doubles the number of circles we must keep track of. As Figure 3-8 shows, this number quite quickly increases to the point where our circle-notation circles become vanishingly small.

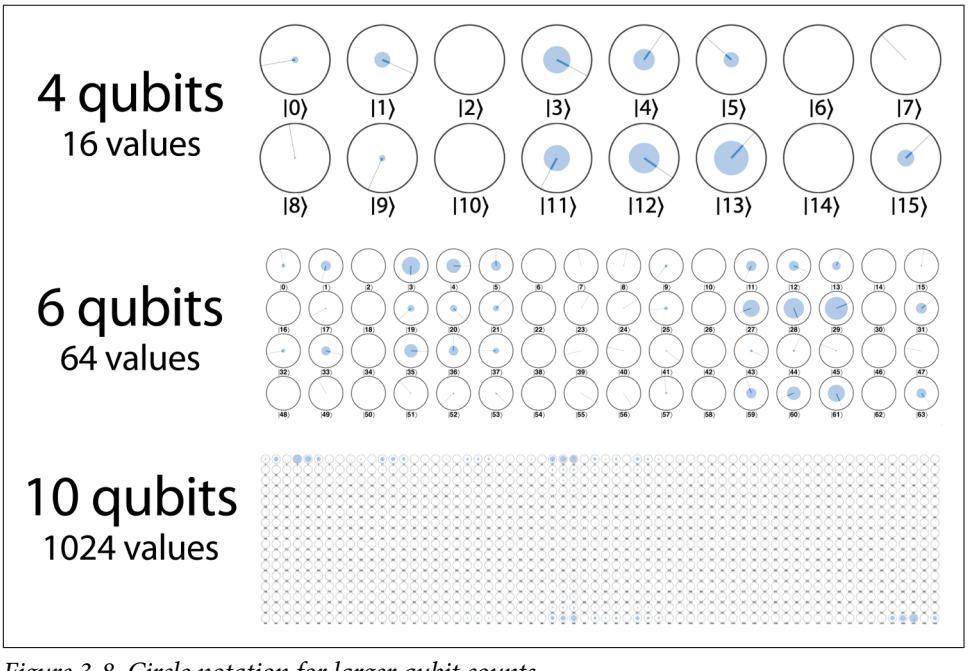


Figure 3-8. Circle notation for larger qubit counts

With so many tiny circles, the circle-notation visualization becomes useful for seeing *patterns* instead of individual values, and we can zoom in to any areas for which we want a more quantitative view. Nevertheless, we can improve clarity in these situations by exaggerating the relative phases, making the line showing a circle's rotation bold, and using differences in color or shading to emphasize differences in phase, as shown in Figure 3-9.

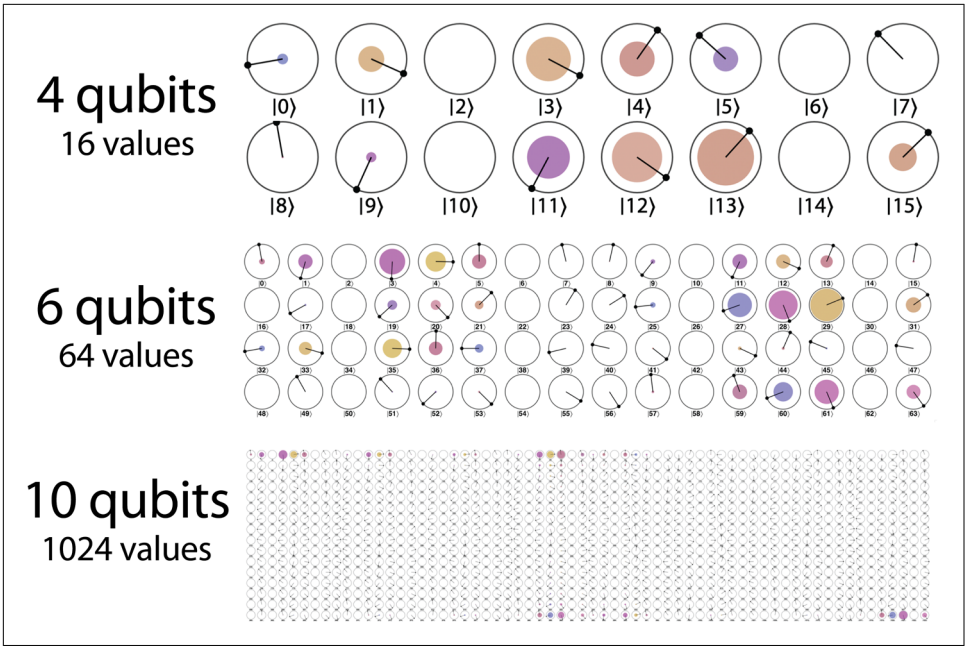
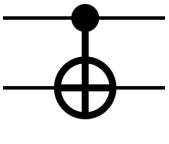


Figure 3-9. Sometimes exaggeration is warranted

In coming chapters, we'll make use of these techniques. But even these "eyeball hacks" are only useful to a point; for a 32-qubit system there are 4,294,967,296 circles—too much information for most displays and eyes alike.

In your own QCEngine programs you can add the line `qc_options.color_by_phase = true;` at the very beginning to enable the *phase coloring* shown in Figure 3-9. The *bold phase lines* can also be toggled using `qc_options.book_render = true;`.

QPU Instruction: CNOT



It's finally time to introduce some definitively multi-qubit QPU operations. By *multi-qubit* operations we mean ones that *require* more than one qubit to operate. The first we'll consider is the powerful `CNOT` operation. `CNOT` operates on *two* qubits and can be thought of as an "if" programming construct with the following condition: *"Apply the `NOT` operation to a target qubit, but only if a condition qubit has the value 1."* The circuit symbol used for `CNOT` shows this logic by

connecting two qubits with a line. A filled dot represents the control qubit, while a NOT symbol shows the target qubit to be conditionally operated on.[1]

The idea of using *condition* qubits to selectively apply actions is used in many other QPU operations, but CNOT is perhaps the prototypical example. Figure 3-10 illustrates the difference between applying a NOT operation to a qubit within a register versus applying CNOT (conditioned on some other *control* qubit).

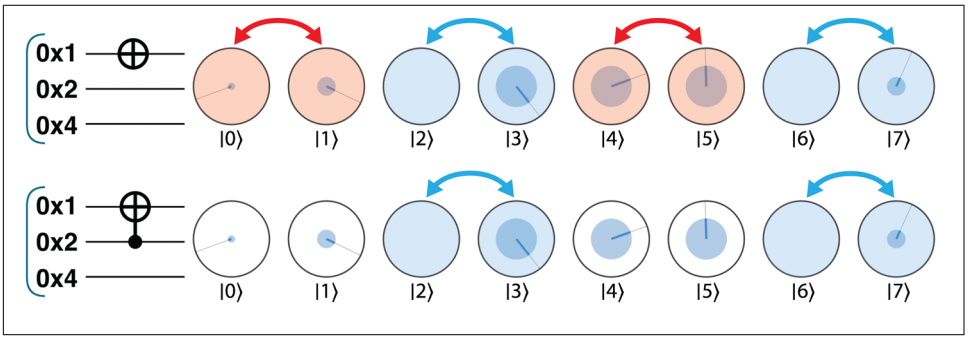


Figure 3-10. NOT versus CNOT in operation

The arrows in Figure 3-10 show which operator pairs have their circles swapped in circle notation. We can see that the essential operation of CNOT is the same as that of NOT, only more selective—applying the NOT operation only to values whose binary representations (in this example) have a 1 in the second bit (010=2, 011=3, 110=6, and 111=7).

Reversibility: Like the NOT operation, CNOT is its own inverse—applying the CNOT operation twice will return a multi-qubit register to its initial state.

On its own there's nothing particularly quantum about CNOT; conditional logic is, of course, a fundamental feature of conventional CPUs. But armed with CNOT we can now ask an interesting and distinctly quantum question. What would happen if the control qubit of a CNOT operation is in a superposition, as exemplified in Figure 3-11?

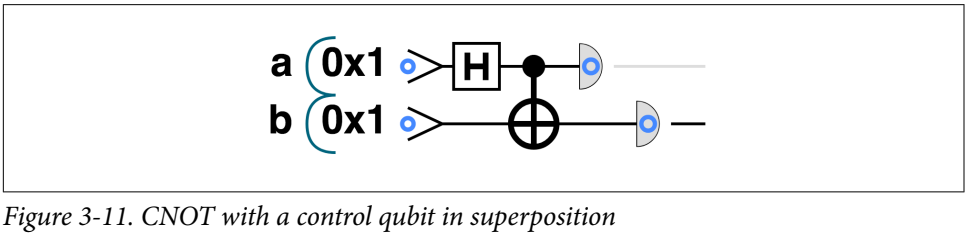


Figure 3-11. CNOT with a control qubit in superposition

1 Operations can also be controlled upon the value 0. To achieve this, we simply use a pair of NOT gates on the control register, one before and one after the operation.

For ease of reference, we've temporarily labeled our two qubits as `a` and `b` (rather than using hexadecimal). Starting with our register in the $|0\rangle$ state, let's walk through the circuit and see what happens. Figure 3-12 shows the circuit and the state at the beginning of the program, before instructions have been executed.

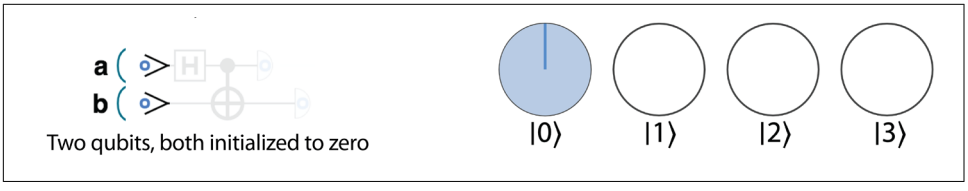


Figure 3-12. Bell pair step 1

First we apply `HAD` to qubit `a`. Since `a` is the lowest-weight qubit in the register, this creates a superposition of the values $|0\rangle$ and $|1\rangle$, visualized in Figure 3-13 in circle notation.

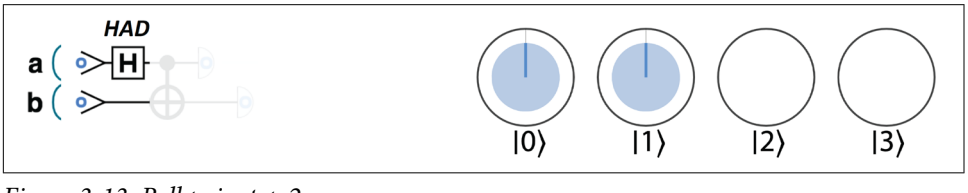


Figure 3-13. Bell pair step 2

Next we apply the `CNOT` operation such that qubit `b` is *conditionally* flipped, dependent on the state of qubit `a`, as shown in Figure 3-14.

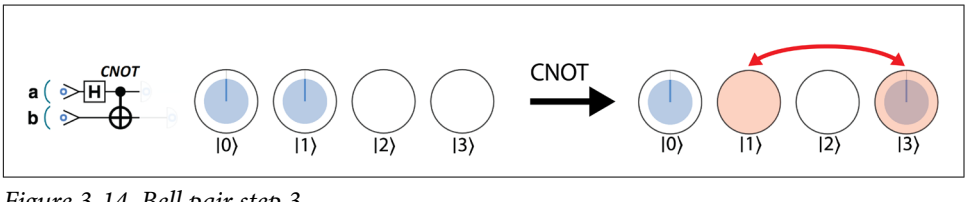


Figure 3-14. Bell pair step 3

The result is a superposition of $|0\rangle$ and $|3\rangle$. This makes sense, since *if* qubit `a` had taken value $|0\rangle$, then no action would occur on `b`, and it would remain in the state $|0\rangle$—leaving the register in a total state of $|0\rangle|0\rangle=|0\rangle$. However, *if* `a` had been in state $|1\rangle$, then a `NOT` would be applied to `b` and the register would have a value of $|1\rangle|1\rangle=|3\rangle$. Another way to understand the operation of `CNOT` in Figure 3-14 is that it simply follows the `CNOT` circle-notation rule, which implies swapping the states $|1\rangle$ and $|3\rangle$ (as is shown by the red arrow in Figure 3-14). In this case, one of these circles just so happens to be in superposition.

The result in Figure 3-14 turns out to be a very powerful resource. In fact, it's the two-qubit equivalent of the *entangled state* we first saw in Figure 3-3. We've already noted that these entangled states demonstrate a kind of interdependence between qubits—if we read out the two qubits shown in Figure 3-14, although the outcomes will be random, they will always agree (i.e., be either `00` or `11`, with 50% chance for each).

The one exception we've made so far to our mantra of "avoid physics at all costs" was to give a slightly deeper insight into superposition. The phenomenon of entanglement is equally important, so—for one final time—we'll indulge ourselves in a few sentences of physics to give you a better feel for *why* entanglement is so powerful.[2]

If you prefer code samples over physics insight, you can happily skip the next couple of paragraphs without any adverse side effects.

Entanglement might not necessarily strike you as strange. Agreement between the values of conventional bits certainly isn't cause for concern, even if they *randomly* assume correlated values. In conventional computing, if two otherwise random bits are always found to agree on readout, then there are two entirely unremarkable possibilities:

1. Some mechanism in the past has coerced their values to be equal, giving them a common cause. If this is the case, their randomness is actually illusory.
2. The two bits truly assume random values at the very moment of readout, but they are able to communicate to ensure their values correlate.

In fact, with a bit of thought it is possible to see that these are the *only* two ways we could explain random agreement between two conventional bits.

Yet through a clever experiment initially proposed by the great Irish physicist John Bell, it's possible to conclusively demonstrate that entanglement allows such agreement without either of these two reasonable explanations being responsible! This is the sense in which you may hear it said that entanglement is a kind of distinctly quantum link between qubits that is *stronger than could ever be conventionally possible*. As we start programming more complex QPU applications, entanglement will begin popping up everywhere. You won't need these kinds of philosophical insights to make practical use of entanglement, but it never hurts to have a little insight into what's going on under the hood.

2 Honestly, this is for *reals* the absolute, 100%, last mention of physics. Pinky promise.

Hands-on: Using Bell Pairs for Shared Randomness

The entangled state we created in Figure 3-14 is commonly referred to as a Bell pair.[3] Let's see a quick and easy way to put the powerful link within these states to use.

In the previous chapter we noted that measuring a single qubit's superposition provides us with a Quantum Random Number Generator. Similarly, the reading out of a Bell pair acts like a QRNG, only now we will obtain *agreeing* random values on two qubits.

A surprising fact about entanglement is that the qubits involved remain entangled no matter how far apart we may move them. Thus, we can easily use Bell pairs to generate *correlated random bits* at different locations. Such bits can be the basis for establishing secure shared randomness—something that critically underlies the modern internet.

The code snippet in Example 3-2 implements this idea, generating shared randomness by creating a Bell pair and then reading out a value from each qubit, as shown in Figure 3-15.

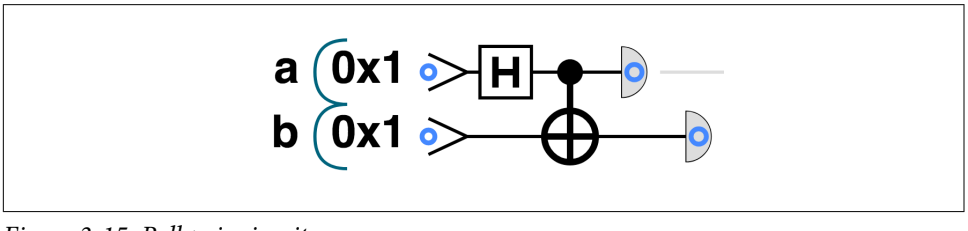


Figure 3-15. Bell pair circuit

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=3-2*.

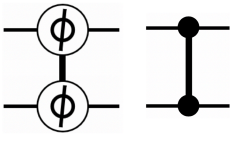
Example 3-2. Make a Bell pair

```
qc.reset(2);
var a = qint.new(1, 'a');
var b = qint.new(1, 'b');
qc.write(0);
a.had();           // Place into superposition
b.cnot(a);         // Entangle
var a_result = a.read();
var b_result = b.read();
```

3 This and several other states carry that name because they were the states used by John Bell in his demonstration of the inexplicable correlations of entangled states.

```
qc.print(a_result);
qc.print(b_result);
```

QPU Instructions: CPHASE and CZ



Another very common two-qubit operation is `CPHASE(`θ`)`. Like the `CNOT` operation, `CPHASE` employs a kind of entanglement-generating conditional logic. Recall from Figure 3-6 that the single-qubit `PHASE(`θ`)` operation acts on a register to rotate (by angle θ) the $|1\rangle$ values in that qubit's operator pairs. As `CNOT` did for `NOT`, `CPHASE` restricts this action on some target qubit to occur only when another control qubit assumes the value $|1\rangle$. Note that `CPHASE` only acts when its control qubit is $|1\rangle$, and when it does act, it only affects target qubit states having value $|1\rangle$. This means that a `CPHASE(`θ`)` applied to, say, qubits `0x1` and `0x4` results in the rotation (by θ) of all circles for which *both* these two qubits have a value of $|1\rangle$. Because of this particular property, `CPHASE` has a symmetry between its inputs not shared by `CNOT`. Unlike with most other controlled operations, it's irrelevant which qubit we consider to be the target and which we consider to be the control for `CPHASE`.

In Figure 3-16 we compare the operation of a `CPHASE` between the `0x1` and `0x4` qubits with individual `PHASE` operations on these qubits.

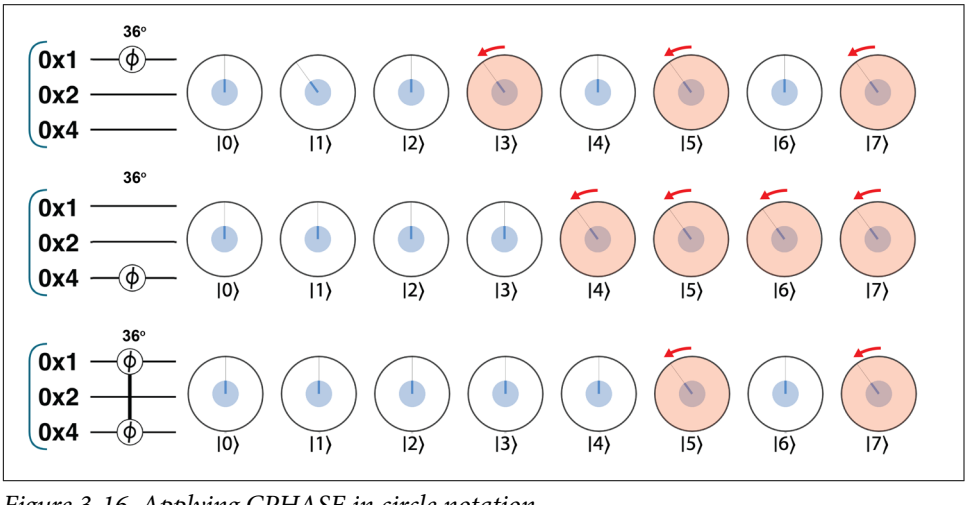


Figure 3-16. Applying CPHASE in circle notation

On their own, single-qubit `PHASE` operations will rotate the relative phases of *half* of the circles associated with a QPU register. Adding a condition further cuts the

number of rotated circles in half. We can continue to add conditional qubits to our `CPHASE`, each time halving the number of values we act on. In general, the more we *condition* QPU operations, the more selective we can be with which values in a QPU register we manipulate.

QPU programs frequently employ the `CPHASE(`θ`)` operation with a phase of $\theta = 180°$, and consequently this particular implementation of `CPHASE` is given its own name, `CZ`, along with its own simplified symbol, shown in Figure 3-17. Interestingly, `CZ` can be constructed from `HAD` and `CNOT` very easily. Recall from Figure 2-14 that the `phase(180)` (`Z`) operation can be made from two `HAD` operations and a `NOT`. Similarly, `CZ` can be made from two `HAD` operations and a `CNOT`, as shown in Figure 3-17.

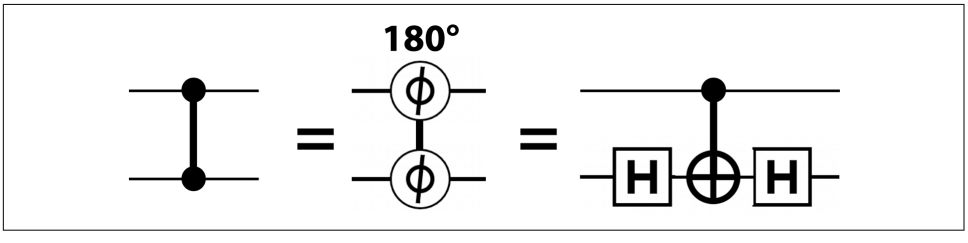


Figure 3-17. Three representations of CPHASE(180)

QPU Trick: Phase Kickback

Once we start thinking about altering the phase of one QPU register *conditioned* on the values of qubits in some other register, we can produce a surprising and useful effect known as *phase kickback*. Take a look at the circuit in Figure 3-18.

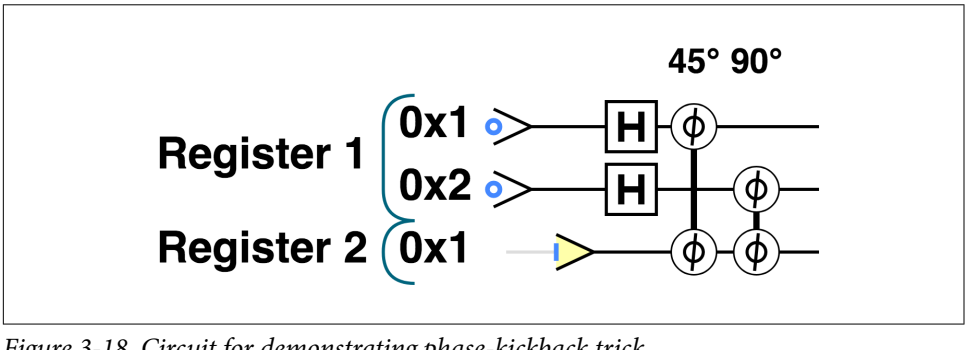


Figure 3-18. Circuit for demonstrating phase-kickback trick

One way to think of this circuit is that, after placing register 1 in a superposition of all of its $2^2 = 4$ possible values, we've rotated the phase of register 2, conditional on the values taken by the qubits of register 1. However, looking at the resulting individual states of *both* registers in circle notation, we see that something interesting has also happened to the state of Register 1, as shown in Figure 3-19.

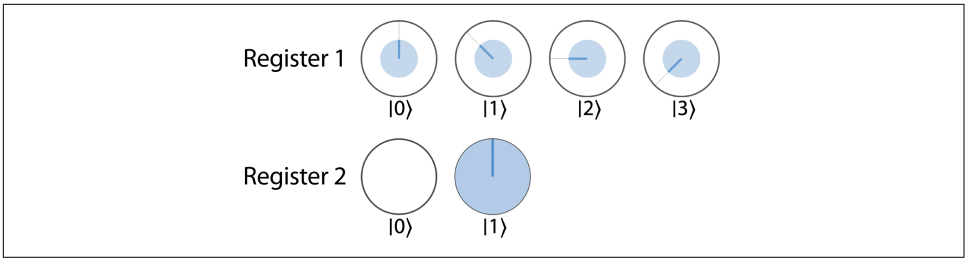


Figure 3-19. States of both registers involved in phase kickback

The phase rotations we tried to apply to the second register (conditioned on qubits from the first) have also affected different values from the first register! More specifically, here's what we're seeing happen in the preceding circle-notation representations:

- The 45° rotation we performed on register 2 conditioned on the lowest-weight qubit from register 1 has *also* rotated every value in register 1 for which this lowest-weight qubit is activated (the $|1\rangle$ and $|3\rangle$ values).
- The 90° rotation we performed on register 2 conditioned on the highest-weight qubit from register 1 has also been *kicked back* onto all values in register 1 having this highest-weight qubit activated (the $|2\rangle$ and $|3\rangle$ values).

The net result that we see on register 1 is the combination of these phase rotations that have been kicked back onto it from our intended target of register 2. Note that since register 2 is *not* in superposition, its (global) phase remains unchanged.

Phase kickback is a very useful idea, as we can use it to apply phase rotations to specific values in a register (register 1 in the preceding example). We can do this by performing a phase rotation on some *other* register conditioned on qubits from the register we really care about. We can choose these qubits to specifically pick out the values we want to rotate.

For phase kickback to work, we always need to initialize the second register in the $|1\rangle$ value. Notice that although we are applying two-qubit operations between the two registers, we are not creating any entanglement; hence, we can fully represent the state as separate registers. Two-qubit gates do not always generate entanglement between registers; we will see why in Chapter 14.

If this phase-kickback trick initially strikes you as a little mind-bending, you're in good company—it can take a while to get used to. The easiest way to get a feel for it is to play around with some examples to build intuition. To get you started,

Example 3-3 contains QCEngine code for reproducing the two-register example described previously.

Sample Code

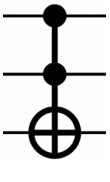
Run this sample online at *http://oreilly-qc.github.io?p=3-3*.

Example 3-3. Phase kickback

```
qc.reset(3);
// Create two registers
var reg1 = qint.new(2, 'Register 1');
var reg2 = qint.new(1, 'Register 2');
reg1.write(0);
reg2.write(1);
// Place the first register in superposition
reg1.had();
// Perform phase rotations on second register,
// conditioned on qubits from the first
qc.phase(45, 0x4, 0x1);
qc.phase(90, 0x4, 0x2);
```

Phase kickback will be of great use in Chapter 8 to understand the inner workings of the *quantum phase estimation* QPU primitive, and again in Chapter 13 to explain how a QPU can help us solve systems of linear equations. The wide utility of phase kickback stems from the fact that it doesn't only work for `CPHASE` operations, but any conditional operation that generates a change in a register's phase. This is as good a reason as any to understand how we might construct more general conditional operations.

QPU Instruction: CCNOT (Toffoli)



We've previously noted that multi-qubit conditional operations can be made more selective by performing operations conditioned on more than one qubit. Let's see this in action and generalize the `CNOT` operation by adding multiple conditions. A `CNOT` with two condition qubits is commonly referred to as a `CCNOT` operation. The `CCNOT` is also sometimes called a Toffoli gate, after the identically titled equivalent gate from conventional computing.

With each condition added, the `NOT` operation stays the same, but the number of operator pairs affected in the register's circle notation is reduced by half. We show this in Figure 3-20 by comparing a `NOT` operation on the first qubit in a three-qubit register with the associated `CNOT` and `CCNOT` operations.

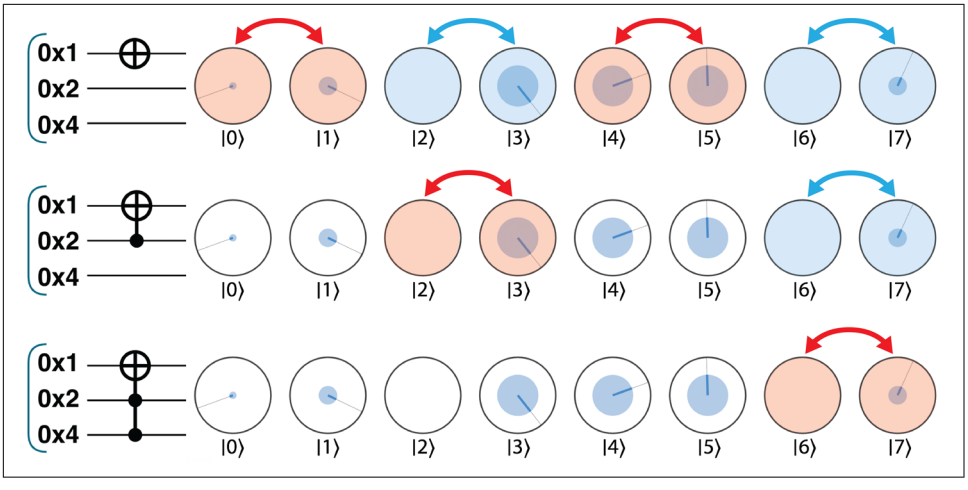
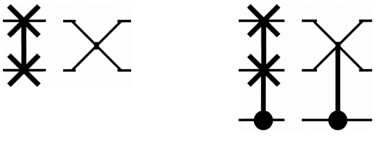


Figure 3-20. Adding conditions makes NOT operations more selective

In a sense, `CCNOT` can be interpreted as an operation implementing "if A AND B then flip C." For performing basic logic, the `CCNOT` gate is arguably the single most useful QPU operation. Multiple `CCNOT` gates can be combined and cascaded to produce a wide variety of logic functions, as we will explore in Chapter 5.

QPU Instructions: SWAP and CSWAP



Another very common operation in quantum computation is `SWAP` (also called *exchange*), which simply exchanges two qubits. If the architecture of a QPU allows it, `SWAP` may be a truly fundamental operation in which the physical objects representing qubits are actually moved to swap their positions. Alternatively, a `SWAP` can be performed by exchanging the *information* contained in two qubits (rather than the qubits themselves) using three `CNOT` operations, as shown in Figure 3-21.

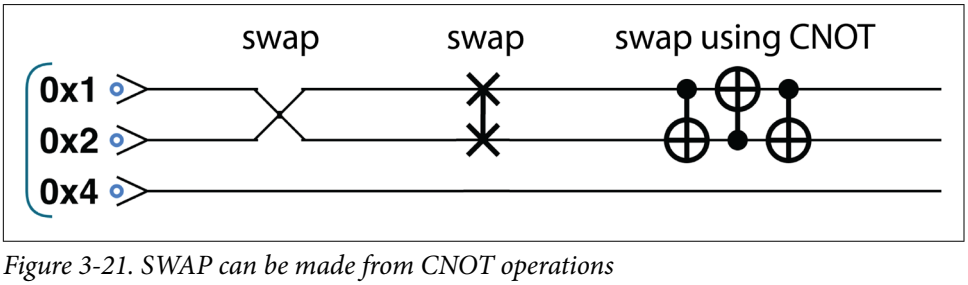


Figure 3-21. SWAP can be made from CNOT operations

In this book, we have two ways to indicate SWAP operations. In the general case we use a pair of connected Xs. When the swapped qubits are adjacent to one another, it is often simpler and more intuitive to cross the qubit lines. The operation is identical in both cases.

You may be wondering why SWAP is a useful operation. Why not simply rename our qubits instead? On a QPU, SWAP comes into its own when we consider generalizing it to a conditional operation called CSWAP, or *conditional exchange*. CSWAP can be implemented using three CCNOT gates, as shown in Figure 3-22.

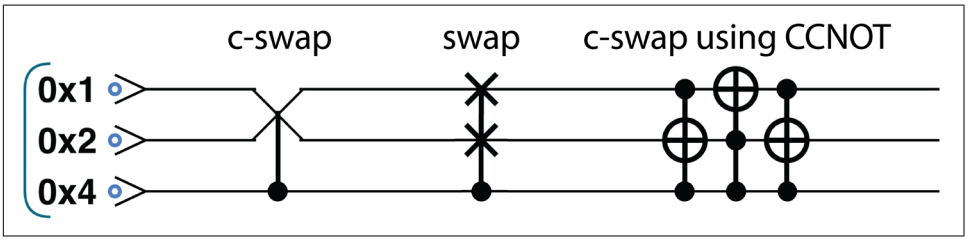


Figure 3-22. CSWAP constructed from CCNOT gates

If the condition qubit for a CSWAP operation is in superposition, we end up with a superposition of our two qubits being exchanged, and also being not exchanged. In Chapters 5 and 12, we'll see how this feature of CSWAP allows us to perform multiplication-by-2 in quantum superposition.

The Swap Test

SWAP operations allow us to build a very useful circuit known as a *swap test*. A swap test circuit solves the following problem: if you're given two qubit registers, how do you tell if they are in the *same* state? By now we know only too well that in general (if either register is in superposition), we can't use the destructive READ operation to completely learn the state of each register in order to make the comparison. The SWAP operation does something a little sneakier. Without telling us what either state is, it simply lets us determine whether or not they're equal.

In a world where we can't necessarily learn precisely what's in an output register, the swap test can be an invaluable tool. Figure 3-23 shows a circuit for implementing a swap test, as demonstrated in Example 3-4.

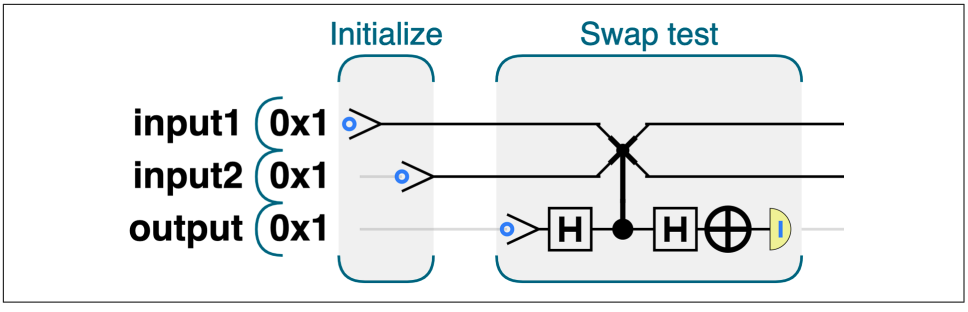


Figure 3-23. Using the swap test to determine whether two registers are in the same state

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=3-4*.

Example 3-4. The swap test

```
// In this example the swap test should reveal
// the equality of the two input states
qc.reset(3);
var input1 = qint.new(1, 'input1');
var input2 = qint.new(1, 'input2');
var output = qint.new(1, 'output');

// Initialize to any states we want to test
input1.write(0);
input2.write(0);

// The swap test itself
output.write(0);
output.had();
// Now exchange the two inputs conditional on the output qubits
input1.exchange(input2, 0x1, output.bits());
output.had();
output.not();
var result = output.read();
// result is 1 if inputs are equal
```

Example 3-4 uses the swap test to compare the states of two single-qubit registers, but the same circuit can easily be extended to compare multi-qubit registers. The result of the swap test is found when we `READ` the extra single-qubit output register that we introduced (no matter how large the input registers are, the output remains a single qubit). By changing the lines `input1.write(0)` and `input2.write(0)`, you can experiment with inputs that differ to varying extents. You should find that if the two input states are equal, the output register always results in a state of $|1\rangle$, so we

definitely obtain a 1 outcome when applying a `READ` to this register. However, as the two inputs become increasingly more different, the probability of `READ`ing a 1 outcome in the output register decreases, eventually becoming 50% in the case where `input1` is $|0\rangle$ and `input2` is $|1\rangle$. Figure 3-24 shows precisely how the outcome probability in the `output` register changes as the similarity between the two input registers is increased.

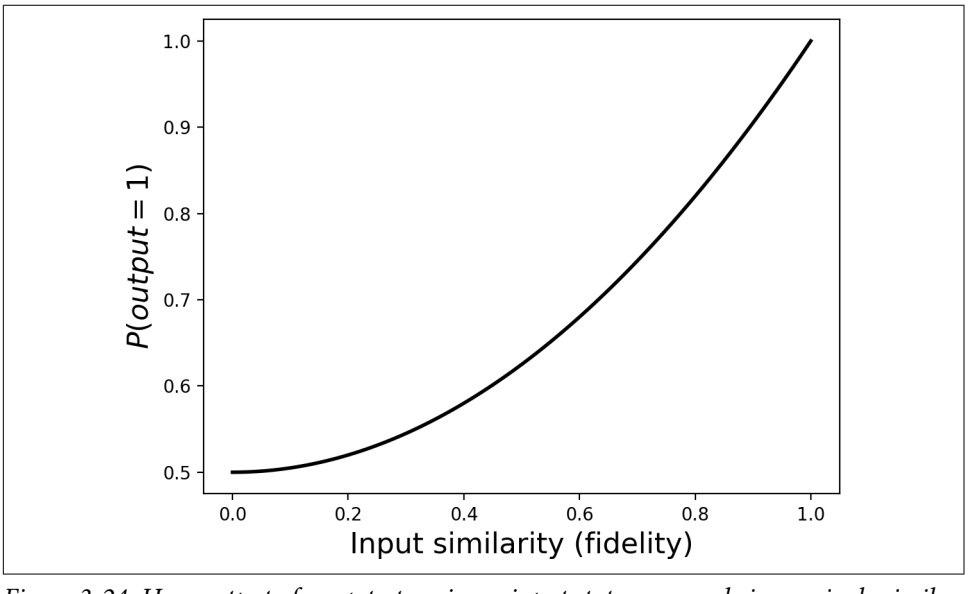


Figure 3-24. How output of swap test varies as input states are made increasingly similar

The x-axis on this plot uses a numerical measure of the difference between two register states known as the *fidelity*. We won't go into the mathematical detail of how the fidelity is calculated between QPU register states, but it mathematically encapsulates a way of comparing two superpositions.

By running the swap test circuit multiple times we can keep track of the outcomes we obtain. The more runs for which we observe a 1 outcome, the more convinced we can be that the two input states were identical. Precisely how many times we would need to repeat the swap test depends on how confident we want to be that the two inputs are identical, and how close we would allow them to be in order to be called identical. Figure 3-25 shows a lower bound for the number of swap tests where we would need to observe 1 outcomes to be 99% confident that our inputs are identical. The plot's y-axis shows how this number changes as we relax the requirement for inputs being

identical (varied along the x-axis). Note that the moment we obtain a 0 outcome in a swap test, we know for sure that the two input states are not identical.[4]

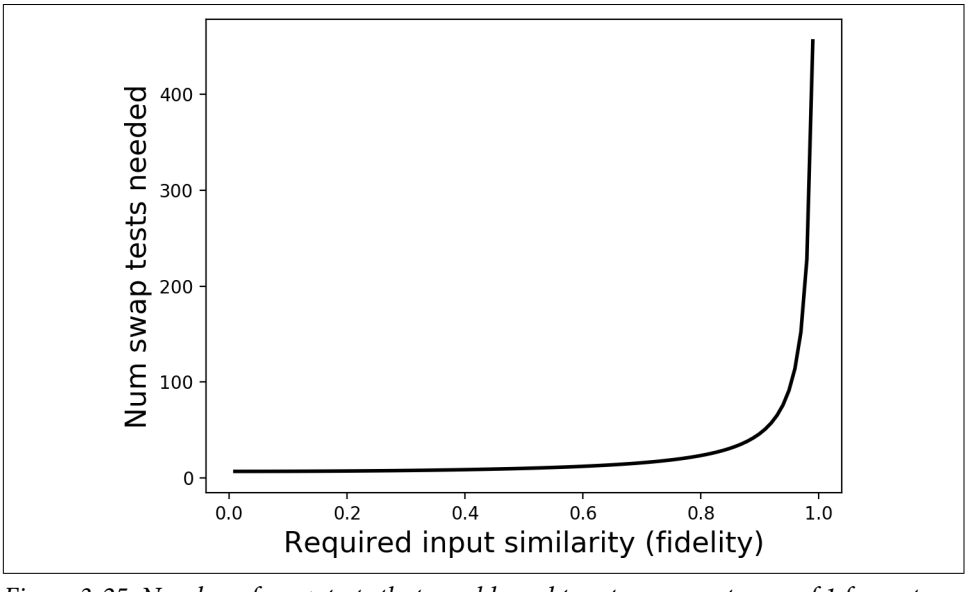


Figure 3-25. Number of swap tests that would need to return an outcome of 1 for us to be 99% confident inputs are identical

Rather than looking at the swap test as a yes/no way of ascertaining whether two states are equal, another useful interpretation is to note that Figure 3-25 shows us that the *probability* that we get a 1 outcome is a measure of just *how* identical the two inputs are (their fidelity). If we repeat the swap test enough times, we can estimate this probability and therefore the fidelity—a more quantitative measure of how close the two states are. Estimating precisely how close two quantum states are in this way is something we'll find useful in quantum machine-learning applications.

Constructing Any Conditional Operation

We've introduced `CNOT` and `CPHASE` operations, but is there such thing as a `CHAD` (conditional `HAD`), or a `CRNOT` (conditional `RNOT`)? Indeed there is! Even if a conditional version of some single-qubit operation is missing from the instruction set of a particular QPU, there's a process by which we can "convert" single-qubit operations into multi-qubit conditional ones.

4 In reality we would want to allow for *some* 0 occurrences if we were genuinely content with close but not identical states—and also perhaps to allow for the possibility of errors in the computation. For this simple analysis we have ignored such considerations.

The general prescription for *conditioning* a single-qubit operation involves a little more mathematics than we want to cover here, but seeing an example will help you feel more comfortable with it. The key idea is that we break our single-qubit operation into smaller steps. It turns out that it's always possible to break a single-qubit operation into a set of steps such that we can use `CNOT`s to conditionally *undo* the operation. The net result is that we can conditionally choose whether or not to effect its action.

This is much easier to see with a simple example. Suppose we are writing software for a QPU that can perform `CNOT`, `CZ`, and `PHASE` operations, but has no instruction to perform `CPHASE`. This is actually a very common case with current QPUs, but fortunately we can easily create our own `CPHASE` from these ingredients.

Recall that the desired effect for a two-qubit `CPHASE` is to rotate the phase for any value of the register for which *both* qubits take the value $|1\rangle$. This is shown in Figure 3-26 for the case of `PHASE(90)`.

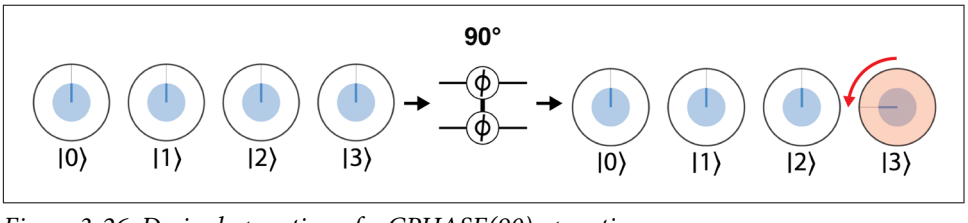


Figure 3-26. Desired operation of a CPHASE(90) operation

We can easily break down a `PHASE(90)` operation into smaller pieces by rotating through smaller angles. For example, `PHASE(90)=PHASE(45)PHASE(45)`. We can also "undo" a rotation by rotating in the opposite direction; for example, the operation `PHASE(45)PHASE(-45)` is the same as doing nothing to our qubit. With these facts in mind, we can construct the `CPHASE(90)` operation described in Figure 3-26 using the operations shown in Figure 3-27 and Example 3-5.

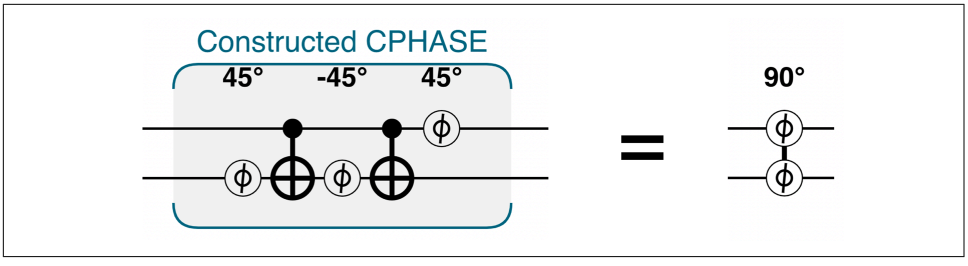


Figure 3-27. Constructing a CPHASE operation

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=3-5*.

Example 3-5. Custom conditional phase

```
var theta = 90;

qc.reset(2);
qc.write(0);
qc.hadamard();

// Using two CNOTs and three PHASEs...
qc.phase( theta / 2, 0x2);
qc.cnot(0x2, 0x1);
qc.phase(-theta / 2, 0x2);
qc.cnot(0x2, 0x1);
qc.phase( theta / 2, 0x1);

// Builds the same operation as a 2-qubit CPHASE
qc.phase(theta, 0x1, 0x2);
```

Following this circuit's operation on different possible inputs we see that it works to apply `PHASE(90)` only when both qubits are $|1\rangle$ (an input of $|1\rangle|1\rangle$). To see this, it helps to recall that `PHASE` has no effect on a qubit in the state $|0\rangle$. Alternatively, we can follow the action of Figure 3-27 using circle notation, as shown in Figure 3-28.

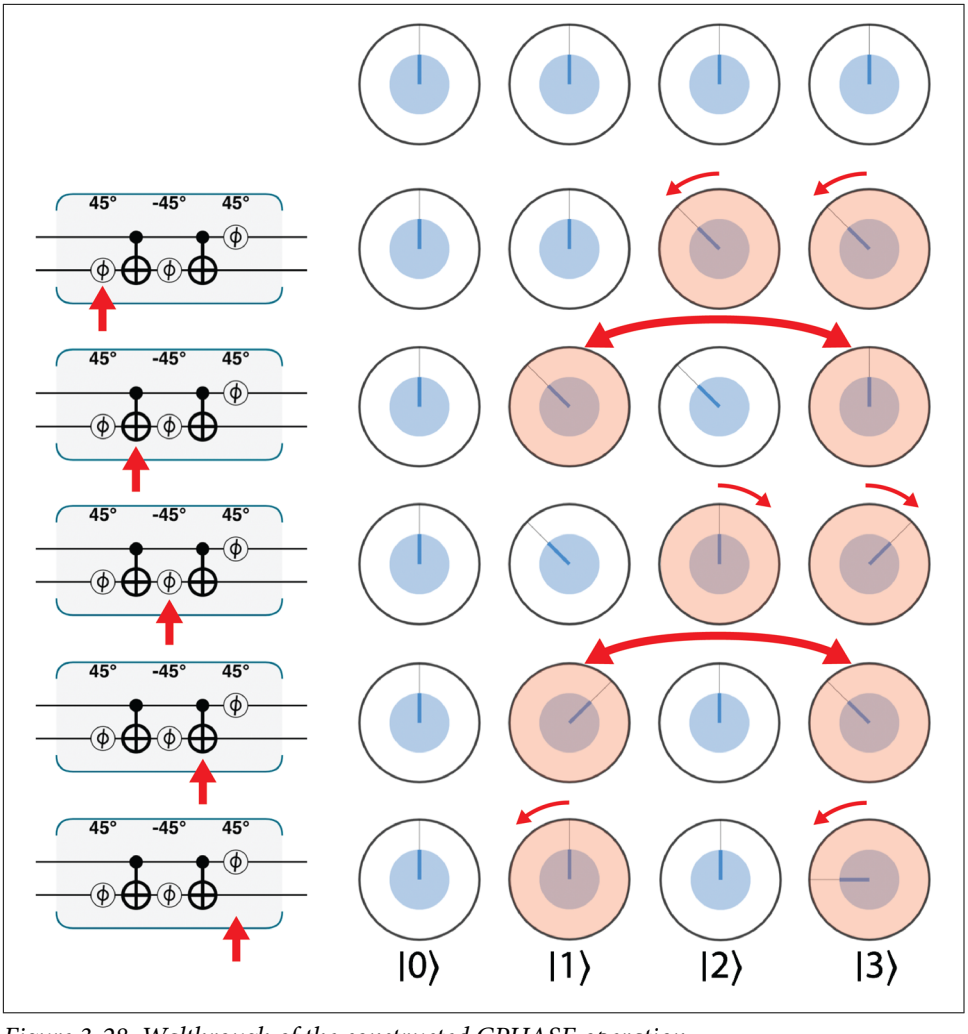


Figure 3-28. Walthrough of the constructed CPHASE operation

While this circuit works correctly to build a `CPHASE(90)` gate, the general case of conditioning an arbitrary operation is slightly more complicated. For a complete recipe and explanation, see Chapter 14.

Hands-on: Remote-Controlled Randomness

Armed with multi-qubit operations, we can explore some interesting and nonobvious properties of entanglement using a small QPU program for *remote-controlled* random number generation. This program will generate two qubits, such that reading out one

instantly affects the probabilities for obtaining a random bit READ from the other. Moreover, this effect occurs over any distance of space or time. This seemingly impossible task, enabled by a QPU, is surprisingly simple to implement.

Here's how the remote control works. We manipulate a pair of qubits such that reading one qubit (either one) returns a 50/50 random bit telling us the "modified" probability of the other. If the result is 0, the other qubit will have 15% probability of being READ to be 1. Otherwise, if the qubit we READ returns 1, the other will have 85% probability of being READ as 1.

The sample code in Example 3-6 shows how to implement this remote-controlled random number generator. As in Example 3-1, we make use of QCEngine `qint` objects to be able to easily address groups of qubits.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=3-6*.

Example 3-6. Remote-controlled randomness

```
qc.reset(2);
var a = qint.new(1, 'a');
var b = qint.new(1, 'b');
qc.write(0);
a.had();
// now prob of a is 50%
b.had();
b.phase(45);
b.had();
// now prob of b is 15%
b.cnot(a);
// Now, you can read *either*
// qubit and get 50% prob.
// If the result is 0, then
// the prob of the *remaining*
// qubit is 15%, else it's 85%.
var a_result = a.read();
var b_result = b.read();
qc.print(a_result + ' ');
qc.print(b_result + '\n');
```

We follow the effect of each operation within this program using circle notation in Figure 3-29.

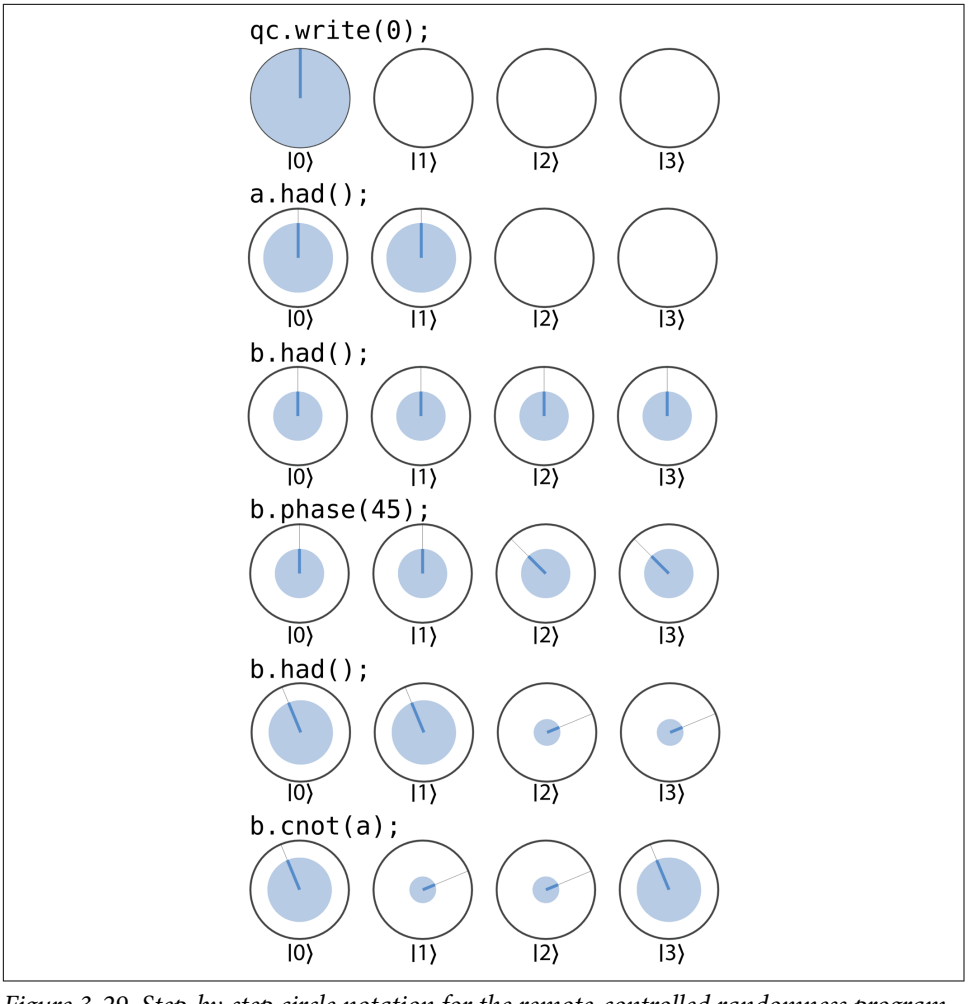


Figure 3-29. Step-by-step circle notation for the remote-controlled randomness program

After these steps are completed we're left with an entangled state of two qubits. Let's consider what would happen if we were to `READ` qubit `a` from the state at the end of Figure 3-29. If qubit `a` were `READ` to have value 0 (as it will with 50% probability), then only the circles compatible with this state of affairs would remain. These are the values for $|0\rangle|0\rangle = |0\rangle$ and $|1\rangle|0\rangle = |2\rangle$, and so the state of the two qubits becomes the one shown in Figure 3-30.

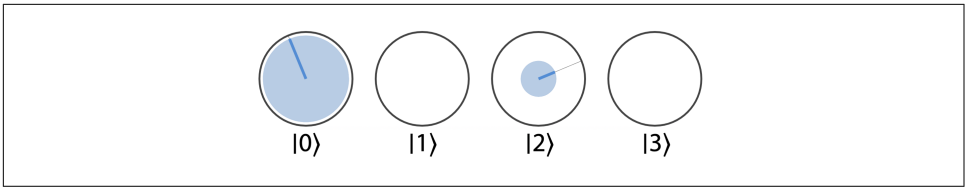


Figure 3-30. The state of one qubit in the remote control after the other is READ to be 0

The two values in this state have nonzero probabilities, both for obtaining 0 and for obtaining 1 when reading out qubit `b`. In particular, qubit `b` has 70%/30% probabilities of reading out 0/1.

However, suppose that our initial measurement on qubit `a` had yielded 1 (which also occurs with 50% probability). Then only the $|0\rangle|1\rangle=|1\rangle$ and $|1\rangle|1\rangle=|3\rangle$ values will remain, as shown in Figure 3-31.

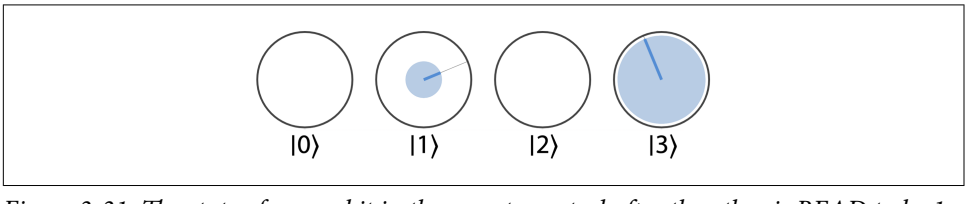


Figure 3-31. The state of one qubit in the remote control after the other is READ to be 1

Now the probabilities for reading out the 0/1 outcomes on qubit `b` are 30%/70%.

Although we're able to change the probability distribution of readout values for qubit `b` instantaneously, we're unable to do so with any intent—we can't choose whether to cause the 70%/30% or the 30%/70% distribution—since the measured outcome of qubit `a` is obtained randomly. It's a good job, too, because if we could make this change deterministically we could send *signals* instantaneously using entanglement; i.e., faster than light. Although sending signals faster than light sounds like fun, were this possible, bad things would happen.[5] In fact, one of the strangest things about entanglement is that it allows us to change the states of qubits instantaneously across arbitrary distances, but always conspires to do so in a way that precludes us from sending intelligible, predetermined information. It seems that the universe isn't a big fan of sci-fi.

5 Sending-information-back-in-time-causality-violating kinds of bad things. The venerated work of Dr. Emmett Brown attests to the dangers of such hijinks.

Conclusion

We've seen how single- and multi-qubit operations can allow us to manipulate the properties of superposition and entanglement within a QPU. Having these at our disposal, we're ready to see how they allow us to *compute* with a QPU in new and powerful ways. In Chapter 5 we'll show how fundamental digital logic can be reimagined, but first in the next chapter we conduct a hands-on exploration of quantum teleportation. Not only is quantum teleportation a fundamental component of many quantum applications, but exploring it also consolidates everything we've covered so far about describing and manipulating qubits.

CHAPTER 4

Quantum Teleportation

In this chapter we introduce a QPU program allowing us to immediately teleport an object across a distance of 3.1 millimeters! The same code would work over interstellar distances, given the right equipment.

Although teleportation might conjure up images of magician's parlor tricks, we'll see that the kind of *quantum* teleportation we can perform with a QPU is equally impressive, yet far more practical—and is, in fact, an essential conceptual component of QPU programming.

Hands-on: Let's Teleport Something

The best way to learn about teleportation is to try to do it. Keep in mind that throughout all of human history up until the time of writing, only a few thousand people have actually performed physical teleportation of any kind, so just running the following code makes you a pioneer.

For this example, rather than a simulator, we will use IBM's five-qubit actual QPU, as seen in Figure 4-1. You'll be able to paste the sample code from Example 4-1 into the IBM Q Experience website, click a button, and confirm that your teleportation was successful.

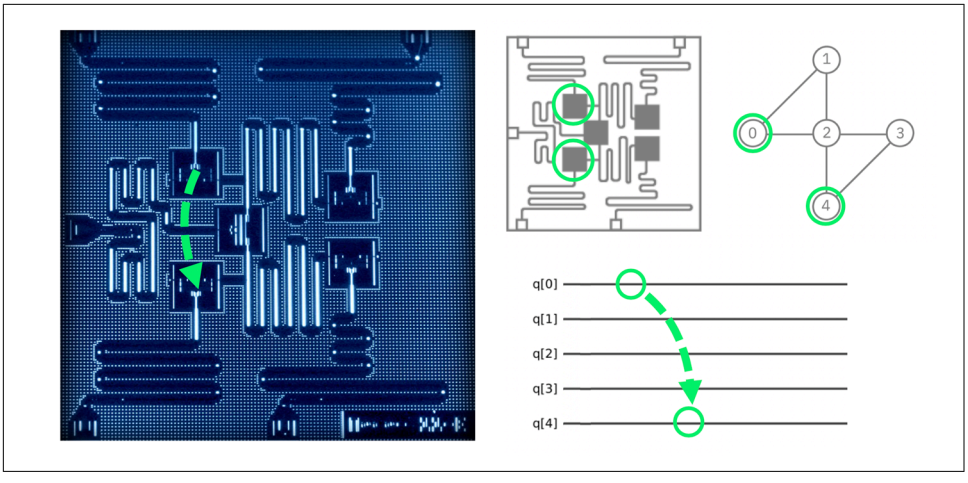


Figure 4-1. The IBM chip is very small, so the qubit does not have far to go; the image and schematics show the regions on the QPU we will teleport between[1]

The IBM Q Experience can be programmed using OpenQASM,[2] and also Qiskit.[3] Note that the code in Example 4-1 is *not* JavaScript to be run on QCEngine, but rather OpenQASM code to be run online through IBM's cloud interface, shown in Figure 4-2. Doing so allows you to not just simulate, but actually *perform* the teleportation of a qubit currently at IBM's research center in Yorktown Heights, New York. We'll walk you through how to do this. Following through the code in detail will also help you understand precisely how quantum teleportation works.

1 Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

2 OpenQASM is the quantum assembly language supported by IBM Q Experience.

3 Qiskit is an open-source software development kit for working with the IBM Q quantum processors.

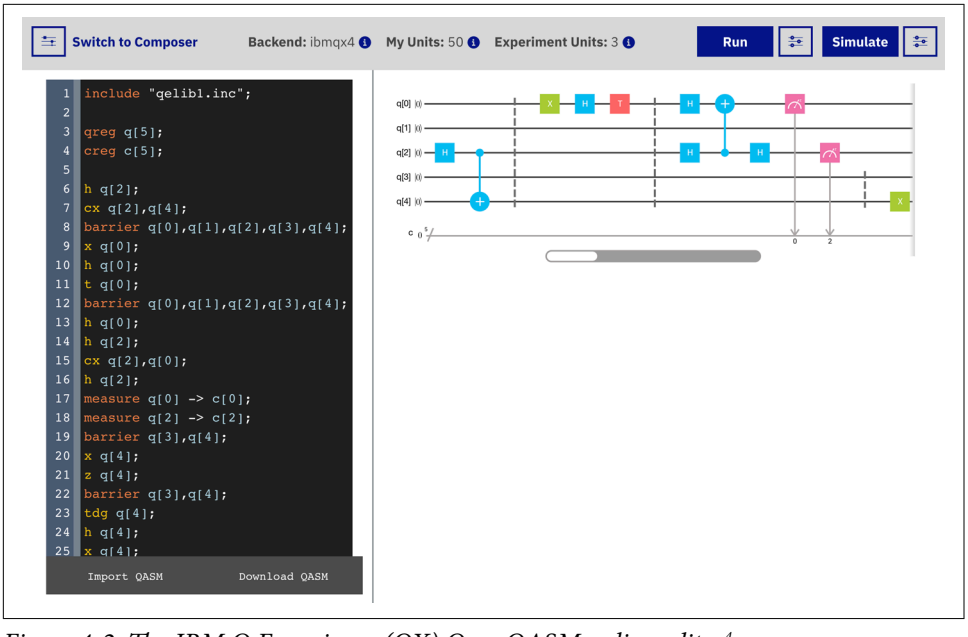


Figure 4-2. The IBM Q Experience (QX) OpenQASM online editor[4]

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=4-1*.

Example 4-1. Teleport and verify

```
include "qelib1.inc";
qreg q[5];
creg c[5];

// Step 1: Create an entangled pair
h q[2];
cx q[2],q[4];
barrier q[0],q[1],q[2],q[3],q[4];

// Step 2: Prepare a payload
x q[0];
h q[0];
t q[0];
barrier q[0],q[1],q[2],q[3],q[4];

// Step 3: Send
h q[0];
```

4 Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

```
h q[2];
cx q[2],q[0];
h q[2];
measure q[0] -> c[0];
measure q[2] -> c[2];
barrier q[3],q[4];

// Step 4: Receive
x q[4];
z q[4];
barrier q[3],q[4];

// Step 5: Verify
tdg q[4];
h q[4];
x q[4];
measure q[4] -> c[4];
```

Before getting to the details, first some clarifying points. By *quantum teleportation* we mean the ability to transport the precise state (i.e., magnitudes and relative phase) of one qubit to another. Our intention is to take all the information contained in the first qubit and put it in the second qubit. Recall that quantum information cannot be replicated; hence the information on the first qubit is necessarily destroyed when we teleport it to the second one. Since a quantum description is the most complete description of a physical object, this is actually precisely what you might colloquially think of as teleportation—only at the quantum level.[5]

With that out of the way, let's teleport! The textbook introduction to quantum teleportation begins with a story that goes something like this: A pair of qubits in an *entangled* state are shared between two parties, Alice and Bob (physicists have an obsession with anthropomorphizing the alphabet). These entangled qubits are a resource that Alice will use to teleport the state of some *other* qubit to Bob. So teleportation involves three qubits: the *payload* qubit that Alice wants to teleport, and an entangled pair of qubits that she shares with Bob (and that acts a bit like a quantum Ethernet cable). Alice prepares her payload and then, using `HAD` and `CNOT` operations, she entangles this payload qubit with her other qubit (which is, in turn, *already* entangled with Bob's qubit). She then destroys both her payload and the entangled qubit using `READ` operations. The results from these `READ` operations yield two conventional bits of information that she sends to Bob. Since these are *bits*, rather than qubits, she can use a conventional Ethernet cable for this part. Using those two bits,

5 The caveat, of course, is that humans are made up of many, many quantum states, and so teleporting qubit states is far removed from any idea of teleportation as a mode of transportation. In other words, although teleportation is an accurate description, Lt. Reginald Barclay doesn't have anything to worry about just yet.

Bob performs some single-qubit operations on his half of the entangled pair originally shared with Alice, and lo and behold, it *becomes* the payload qubit that Alice intended to send.

Before we walk through the detailed protocol of quantum operations described here,[6] you may have a concern in need of addressing. "Hang on…" (we imagine you saying), "if Alice is sending Bob conventional information through an Ethernet cable…" (you continue), "then surely this isn't that impressive at all." Excellent observation! It is certainly true that quantum teleportation crucially relies on the transmission of conventional (digital) bits for its success. We've already seen that the magnitudes and relative phases needed to fully describe an arbitrary qubit state can take on a continuum of values. Crucially, the teleportation protocol works even in the case when Alice does not know the state of her qubit. This is particularly important since it is impossible to determine the magnitude and relative phase of a single qubit in an unknown state. And yet—with the help of an entangled pair of qubits—only two conventional bits were needed to effectively transmit the precise configuration of Alice's qubit (whatever its amplitudes were). This configuration will be correct to a potentially infinite number of bits of precision!

So how do we engineer this magic? Here's the full protocol. Figure 4-3 shows the operations that we must employ on the three involved qubits. All these operations were introduced in Chapters 2 and 3.

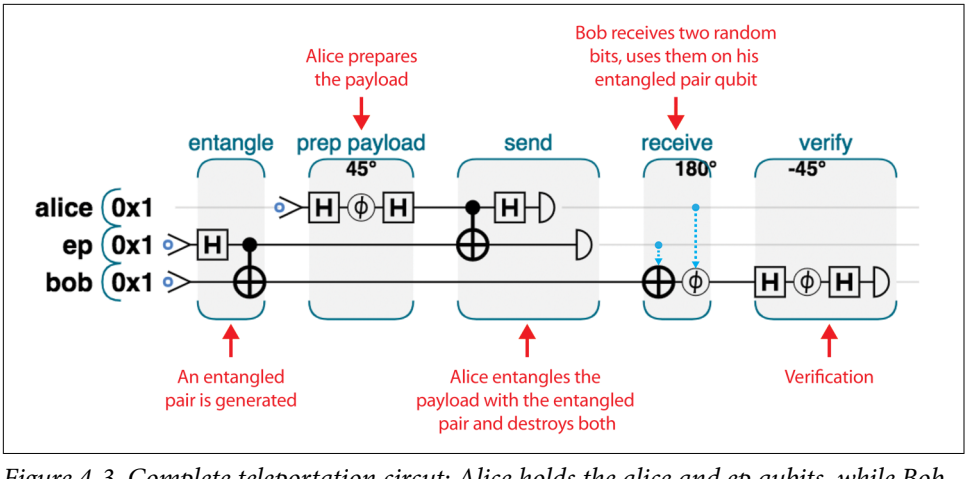


Figure 4-3. Complete teleportation circut: Alice holds the alice and ep qubits, while Bob holds bob

6 The complete source code for both the QASM and QCEngine versions of this example can be seen at *http://oreilly-qc.github.io?p=4-1*.

If you paste the code for this circuit from Example 4-1 into the IBM QX system, the IBM user interface will display the circuit shown in Figure 4-4. Note that this is exactly the same program as we've shown in Figure 4-3, just displayed slightly differently. The quantum gate notation we've been using is standardized, so you can expect to find it used outside this book.[7]

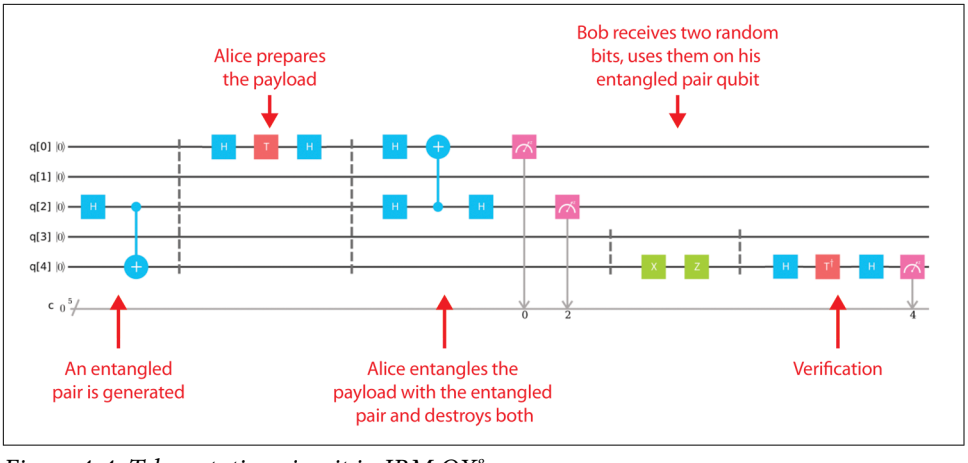


Figure 4-4. Teleportation circuit in IBM QX[8]

When you click Run, IBM will run your program 1,024 times (this number is adjustable) and then display statistics regarding all of these runs. After running the program, you can expect to find something similar (although not identical) to the bar chart shown in Figure 4-5.

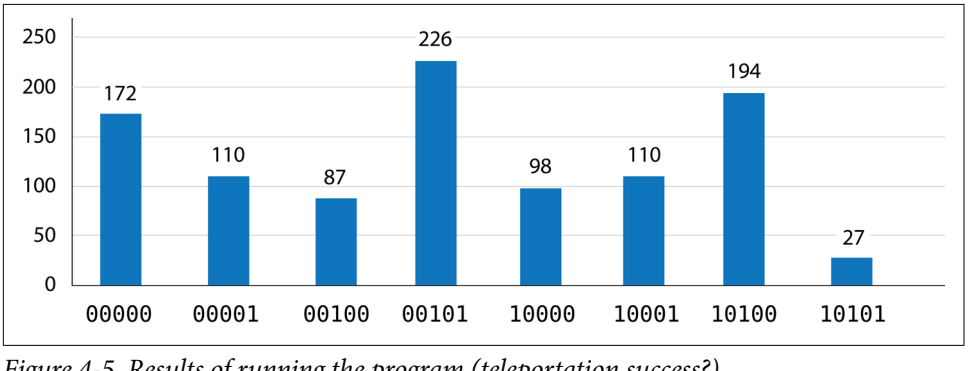


Figure 4-5. Results of running the program (teleportation success?)

7 Gates such as CNOTs and HADs can be combined in different ways to produce the same result. Some operations have different decompositions in IBM QX and QCEngine.

8 Courtesy of International Business Machines Corporation, © International Business Machines Corporation.

Success? Maybe! To demonstrate how to read and interpret these results, let's walk through each step in the QPU program in a little more detail, using circle notation to visualize what's happening to our qubits.[9]

At the time of writing, the circuits and results displayed by IBM QX show what's happening with all *five* qubits available in the QPU, even if we're not using them all. This is why there are two empty qubit lines in the circuit shown in Figure 4-4, and why the bars showing results for each output in Figure 4-5 are labeled with a five-bit (rather than three-bit) binary number—even though only the bars corresponding to the $2^3 = 8$ possible configurations of the three qubits we used are actually shown.

Program Walkthrough

Since we use three qubits in our teleportation example, their full description needs $2^3 = 8$ circles (one for each possible combination of the 3 bits). We'll arrange these eight circles in two rows, which helps us to visualize how operations affect the three constituent qubits. In Figure 4-6 we've labeled each row and column of circles corresponding to a given qubit having a particular value. You can check that these labels are correct by considering the binary value of the register that each circle corresponds to.

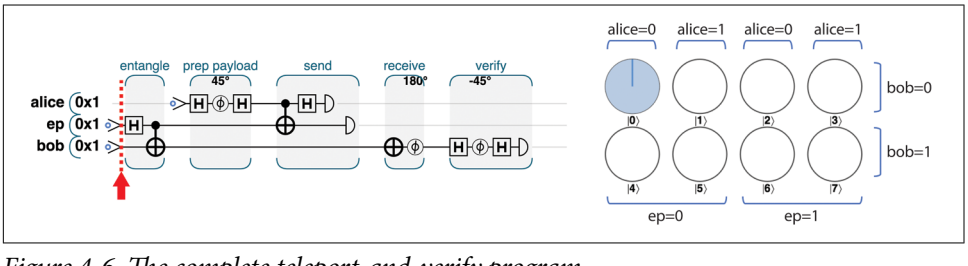


Figure 4-6. The complete teleport-and-verify program

When dealing with multi-qubit registers we'll often arrange our circle notation in rows and columns as we do in Figure 4-6. This is always a useful way of more quickly spotting the behavior of individual qubits: it's easy to pick out the relevant rows or columns.

9 The complete source code for this example can be seen at *http://oreilly-qc.github.io?p=4-1*.

At the beginning of the program, all three qubits are initialized in state $|0\rangle$, as indicated in Figure 4-6—the only possible value is the one where `alice=0` and `ep=0` and `bob=0`.

Step 1: Create an Entangled Pair

The first task for teleportation is establishing an entangled link. The `HAD` and `CNOT` combination achieving this is the same process we used in Chapter 3 to create the specially named *Bell pair* entangled state of two qubits. We can readily see from the circle notation in Figure 4-7 that if we read `bob` and `ep`, the values are 50/50 random, but guaranteed to match each other, à la entanglement.

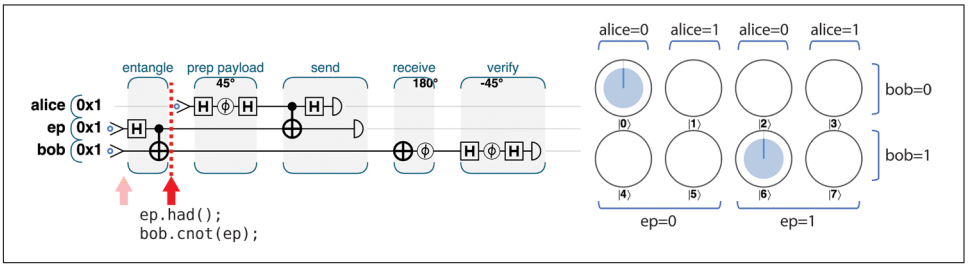


Figure 4-7. Step 1: Create an entangled pair

Step 2: Prepare the Payload

Having established an entanglement link, Alice can prepare the payload to be sent. How she prepares it depends, of course, on the nature of the (quantum) information that she wants to send to Bob. She might write a value to the payload qubit, entangle it with some other QPU data, or even receive it from a previous computation in some entirely separate part of her QPU.

For our example here we'll disappoint Alice by asking her to prepare a particularly simple payload qubit, using only `HAD` and `PHASE` operations. This has the benefit of producing a payload with a readily decipherable circle-notation pattern, as shown in Figure 4-8.

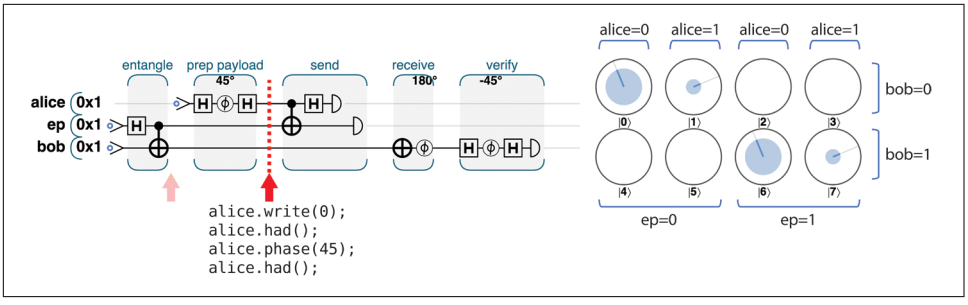


Figure 4-8. Step 2: Prepare the payload

We can see that the `bob` and `ep` qubits are still dependent on one another (only the circles corresponding to the `bob` and `ep` qubits possessing equal values have nonzero magnitudes). We can also see that the value of `alice` is not dependent on either of the other two qubits, and furthermore that her payload preparation produced a qubit that is 85.4% $|0\rangle$ and 14.6% $|1\rangle$, with a relative phase of –90° (the circles corresponding to `alice`=1 are at 90° *clockwise* of the `alice`=0 circles, which is negative in our convention).

Step 3.1: Link the Payload to the Entangled Pair

In Chapter 2 we saw that the conditional nature of the `CNOT` operation can entangle the states of two qubits. Alice now uses this fact to entangle her payload qubit with her half of the entangled pair she already shares with Bob. In terms of circle notation, this action swaps circles around as shown in Figure 4-9.

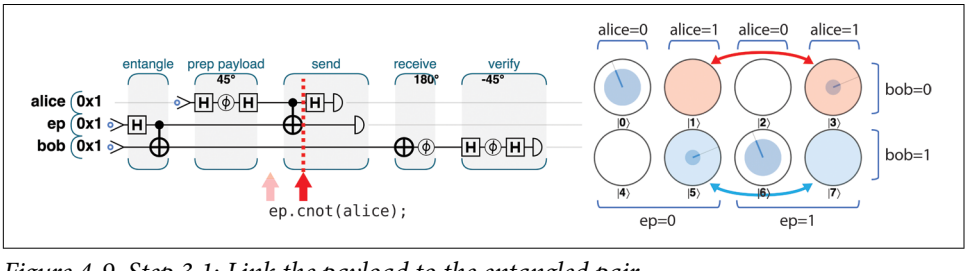


Figure 4-9. Step 3.1: Link the payload to the entangled pair

Now that we have *multiple* entangled states, there's the potential for a little confusion—so let's be clear. Alice and Bob *already* each held one of two entangled qubits (produced in step 1). Now Alice has entangled *another* (payload) qubit onto her half of this (already entangled) pair. Intuitively we can see that in some sense Alice has, by proxy, now linked her payload to Bob's half of the entangled pair—although her payload qubit is still unchanged. `READ` results on her payload will now be logically linked with those of the other two qubits. We can see this link in circle notation since the

QPU register state in Figure 4-9 only contains entries where the XOR of all three qubits is 0. Formerly this was true of `ep` and `bob`, but now it is true for all three qubits forming a *three-qubit* entangled group.

Step 3.2: Put the Payload into a Superposition

To make the link that Alice has created for her payload actually useful, she needs to finish by performing a `HAD` operation on her payload, as shown in Figure 4-10.

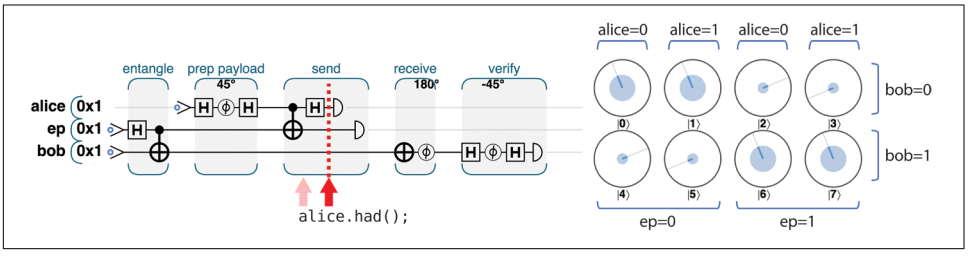


Figure 4-10. Step 3.2: Put the payload into a superposition

To see why Alice needed this `HAD`, take a look at the circle-notation representation of the state of the three qubits shown in Figure 4-10. In each *column* there is a pair of circles, showing a qubit that Bob might receive (we'll see shortly that precisely which one he *would* receive depends on the results of the `READ`s that Alice performs). Interestingly, the four potential states Bob could receive are all different variations on Alice's original payload:

- In the first column (where `alice=0` and `ep=0`), we have Alice's payload, exactly as she prepared it.
- In the second column, we have the same thing, except with a `PHASE(180)` applied.
- In the third column, we see the correct payload, but with a `NOT` having been applied to it ($|0\rangle$ and $|1\rangle$ are flipped).
- Finally, the last column is both phase-shifted and flipped (i.e., a `PHASE(180)` followed by a `NOT`).

If Alice hadn't applied `HAD`, she would have destroyed magnitude and phase information when applying her `READ` operations that she will shortly use (try it!). By applying the `HAD` operation, Alice was able to maneuver the state of Bob's qubit closer to that of her payload.

Step 3.3: READ Both of Alice's Qubits

Next, Alice performs a `READ` operation on her two qubits (the payload and her half of the entangled pair she shares with Bob). This `READ` irrevocably destroys both these

qubits. You may wonder why Alice bothers to do this. As we'll see, it turns out that the results of this unavoidably destructive `READ` operation are crucial for the teleportation protocol to work. Copying quantum states is not possible, even when using entanglement. The only option to communicate quantum states is to teleport them, and when teleporting, we *must* destroy the original.

In Figure 4-11, Alice performs the prescribed `READ` operations on her payload and her half of the entangled pair. This operation returns two bits.

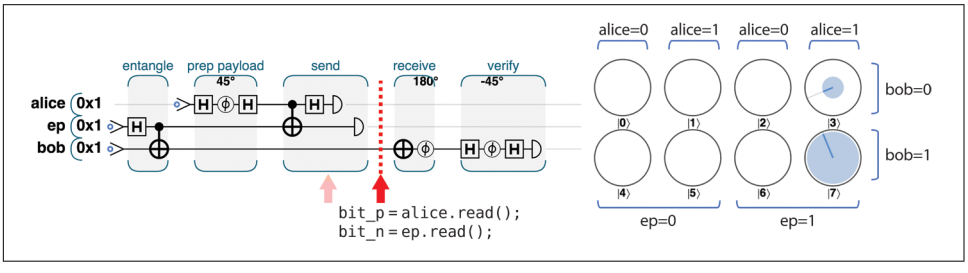


Figure 4-11. Step 3.3: READ both of Alice's qubits

In terms of circle notation, Figure 4-11 shows that by reading the values of her qubits Alice has selected one column of circles (which one she gets being dependent on the random `READ` results), resulting in the circles outside this column having zero magnitude.

Step 4: Receive and Transform

In "Step 3.2: Put the Payload into a Superposition" on page 76 we saw that Bob's qubit could end up in one of four states—each of which is simply related to Alice's payload by `HAD` and/or `PHASE(180)` operations. If Bob could learn which of these four states he possessed, he could apply the necessary inverse operations to convert it back to Alice's original payload. And the two bits Alice has from her `READ` operations are precisely the information that Bob needs! So at this stage, *Alice picks up the phone and transmits two bits of conventional information to Bob*.

Based on which two bits he receives, Bob knows which column from our circle-notation view represents his qubit. If the first bit he receives from Alice is 1, he performs a `NOT` operation on the qubit. Then, if the second bit is 1 he also performs a `PHASE(180)`, as illustrated in Figure 4-12.

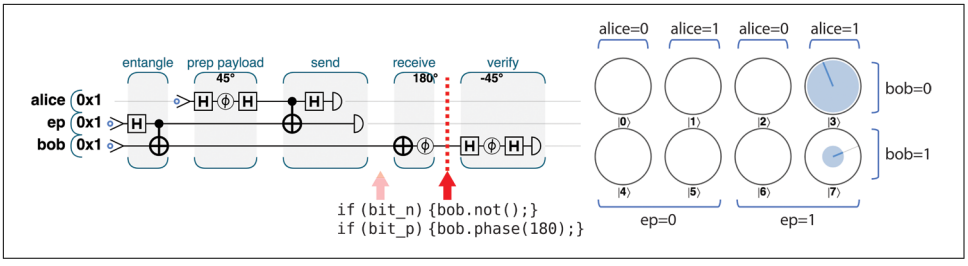


Figure 4-12. Step 4: Receive and transform

This completes the teleportation protocol—Bob now holds a qubit indistinguishable from Alice's initial payload.

The current IBM QX hardware does not support the kind of *feed-forward* operation we need to allow the (completely random) bits from Alice's `READ` to control Bob's actions. This shortcoming can be circumvented by using *post-selection*—we have Bob perform the same operation no matter what Alice sends. This behavior may exasperate Alice, as shown in Figure 4-13. Then we look at all of the outputs, and discard all but the results where Bob did what he would have with the benefit of information from Alice.

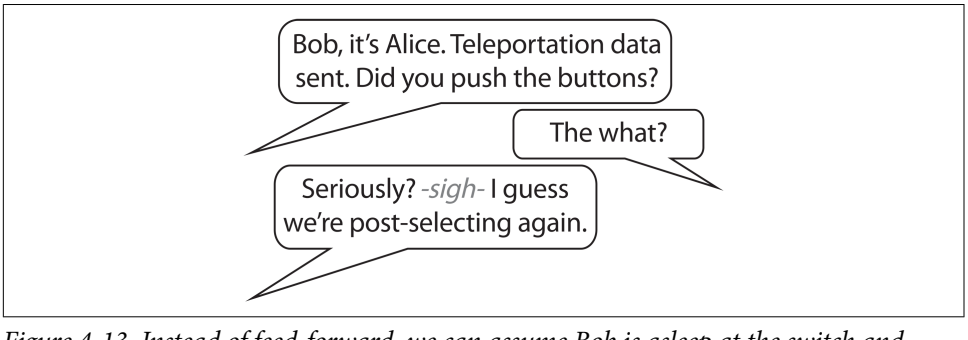


Figure 4-13. Instead of feed-forward, we can assume Bob is asleep at the switch and throw away cases where he makes wrong decisions

Step 5: Verify the Result

If Alice and Bob were using this teleportation in serious work, they'd be finished. Bob would take the teleported qubit from Alice and continue to use it in whatever larger quantum application they were working on. So long as they trust their QPU hardware, they can rest assured that Bob has the qubit Alice intended.

But, what about cases where we'd like to *verify* that the hardware has teleported a qubit correctly (even if we don't mind destroying the teleported qubit in the process)?

Our only option is to READ Bob's final qubit. Of course, we can never expect to learn (and therefore verify) the state of his qubit from a single READ, but by repeating the whole teleportation process and doing multiple READs we can start to build up a picture of Bob's state.

In fact, the easiest way for us to verify the teleportation protocol's success on a physical device would be for Bob to run the "prep the payload" steps that Alice performs on a $|0\rangle$ state to create her payload, on his final qubit, only in reverse. If the qubit Bob has truly matches the one Alice sent, this should leave Bob with a $|0\rangle$ state, and if Bob then performs a final verification READ, it should only ever return a 0. If Bob ever READs this test qubit as nonzero, the teleportation has failed. This additional step for verification is shown in Figure 4-14.

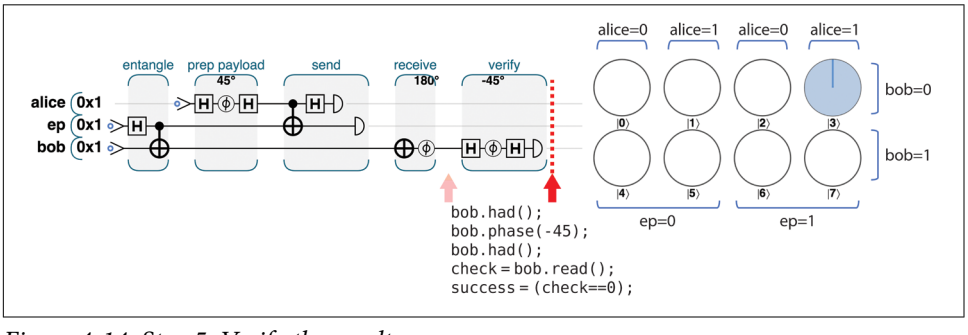


Figure 4-14. Step 5: Verify the result

Even if Alice and Bob are doing serious teleporter work, they will probably want to intersperse their actual teleportations with many verification tests such as these, just to be reassured that their QPU is working correctly.

Interpreting the Results

Armed with this fuller understanding of the teleportation protocol and its subtleties, let's return to the results obtained from our physical teleportation experiment on the IBM QX. We now have the necessary knowledge to decode them.

There are three READ operations performed in the entire protocol: two by Alice as part of the teleportation, and one by Bob for verification purposes. The bar chart in Figure 4-5 enumerates the number of times each of the 8 possible combinations of these outcomes occurred during 1,024 teleportation attempts.

As noted earlier, we'll be *post-selecting* on the IBM QX for instances where Bob happens to perform the right operations (as if he had acted on Alice's READ results). In the example circuit we gave the IBM QX in Figure 4-4, Bob always performs both a HAD and a PHASE(180) on his qubit, so we need to post-select on cases where Alice's two

READ operations gave 11. This leaves two sets of actually useful results where Bob's actions happened to correctly match up with Alice's READ results, as shown in Figure 4-15.

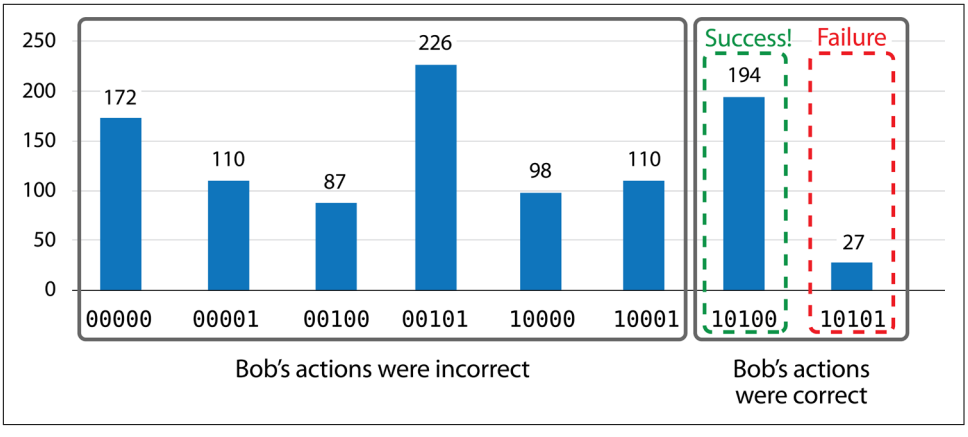


Figure 4-15. Interpreted teleportation results

Of the 221 times where Bob did the correct thing, the teleportation succeeded when Bob's verification READ gave a value of 0 (since he uses the verification step we discussed previously). This means the teleportation succeeded 194 times, and failed 27 times. A success rate of 87.8% is not bad, considering Alice and Bob are using the best available equipment in 2019. Still, they may think twice before sending anything important.

If Bob receives an erroneously teleported qubit that is *almost* like the one Alice sent, the verification is likely to report success. Only by running this test many times can we gain confidence that the device is working well.

How Is Teleportation Actually Used?

Teleportation is a surprisingly fundamental part of the operation of a QPU—even in straight computational applications that have no obvious "communication" aspect at all. It allows us to shuttle information between qubits while working around the *"no-copying"* constraint. In fact, most practical uses of teleportation are over very short distances within a QPU as an integral part of quantum applications. In the upcoming chapters, you'll see that most quantum operations for two or more qubits function by forming various types of entanglement. The use of such quantum links to perform computation can usually be seen as an application of the general concept of teleportation. Though we may not explicitly acknowledge teleportation in the algorithms and applications we cover, it plays a fundamental role.

Fun with Famous Teleporter Accidents

As science fiction enthusiasts, our personal favorite use of teleportation is the classic film *The Fly*. Both the 1958 original and the more modern 1986 Jeff Goldblum version feature an error-prone teleportation experiment. After the protagonist's cat fails to teleport correctly, he decides that the next logical step is to try it himself, but without his knowledge a fly has gotten into the chamber with him.

We feel a little guilty for bringing the crushing news that actual quantum teleportation doesn't live up to what's promised in *The Fly*. To try to make it up to you, we've put together some sample code to teleport a quantum *image* of a fly—even including the small amount of error called for in the movie's plot. The code sample can be run online at *http://oreilly-qc.github.io?p=4-2*, and the horrifying result it produces is shown in Figure 4-16.

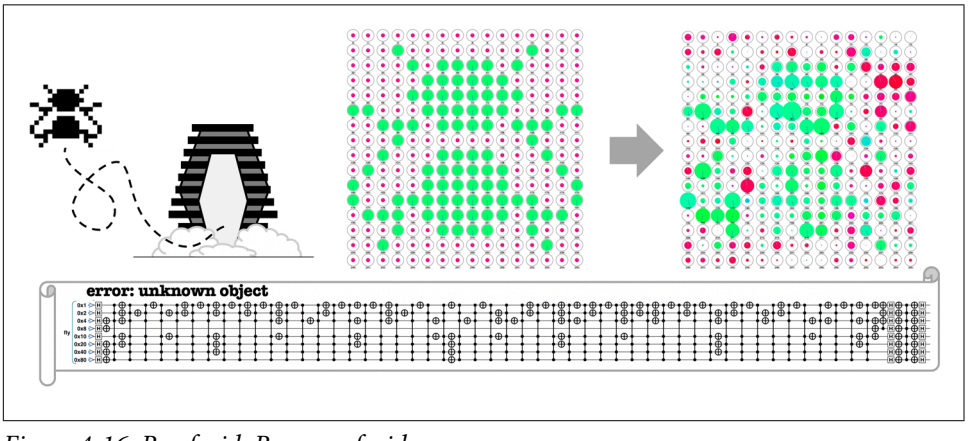


Figure 4-16. Be afraid. Be very afraid.

PART II
QPU Primitives

Now that you know how to describe and manipulate qubits at a basic level, we can introduce some higher-level *QPU primitives*. These primitives form a toolbox, ultimately allowing us to build full-blown applications.

Very roughly speaking, quantum applications tend to have the structure shown in Figure II-1.

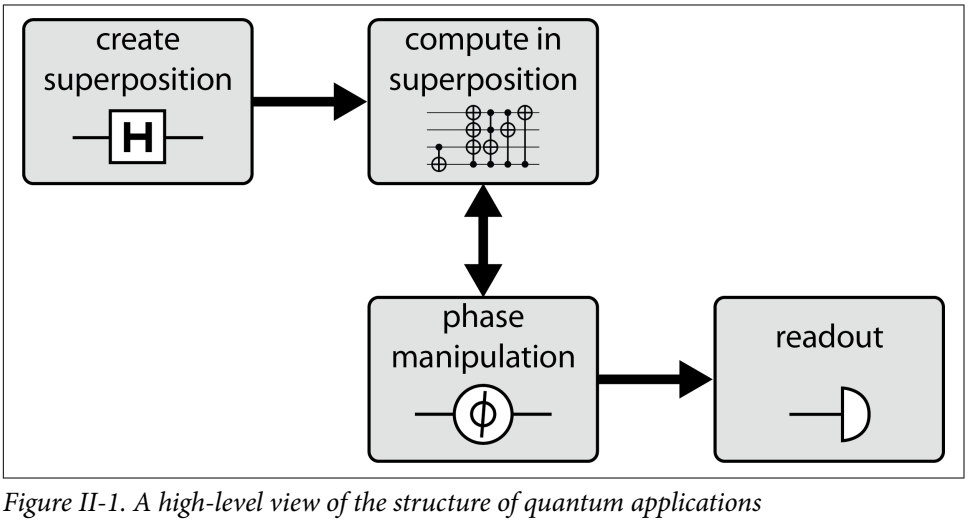


Figure II-1. A high-level view of the structure of quantum applications

QPU primitives help us to fill out this structure. Primitives associated with the second of these four steps (*compute in superposition*) allow us to compute using the

implicit parallelism of superposition, while primitives realizing the third step (*phase manipulation*) ensure that our results can actually be `READ` in a practical way.

These steps are usually implemented together and many times in iteration, in a manner dependent on the particular application. Rather than there being one all-purpose primitive for each step, we'll actually need an arsenal. The next five chapters introduce the primitives listed in Table II-1.

Table II-1. Quantum primitives covered

Primitive	Type	Chapter
Digital logic	Compute in superposition	5
Amplitude amplification	Phase manipulation	6
Quantum Fourier Transform	Phase manipulation	7
Phase estimation	Phase manipulation	8
Quantum data types	Superposition creation	9

In each chapter we first give a "hands-on" introduction to the primitive, then outline ways of *using* that primitive in practical applications. Finally, each chapter ends with a section entitled "Inside the QPU" giving more *intuition* into how the primitive works, often with a breakdown in terms of the fundamental QPU operations we've seen in Chapters 2 and 3.

The art of programming a QPU is to determine which combination of primitives from Table II-1 works to construct a structure like Figure II-1 for a given application. In Part III of the book we show some examples of such constructions.

Now that we know where we're headed, let's start collecting our QPU primitives!

CHAPTER 5

Quantum Arithmetic and Logic

QPU applications often gain an advantage over their conventional counterparts by performing a large number of logical operations *in superposition*.[1]

A key aspect of this is the ability to apply simple arithmetic operations on a qubit register in superposition. In this chapter, we will look in detail at how to do this. Initially, we'll discuss arithmetic operations at the more abstracted level we're used to in conventional programming, dealing with integers and variables rather than qubits and operations. But toward the end of the chapter we'll also take a closer look at the logical operations making these up (akin to the elementary gates of digital logic).

Strangely Different

Conventional digital logic has plenty of well-optimized approaches for performing arithmetical operations—why can't we just replace bits with qubits and use those in our QPU?

The problem, of course, is that conventional operations are expected to operate on a *single* set of inputs at a time, whereas we will often have input registers that are in superposition, for which we want *quantum* arithmetic operations to affect *all* values in the superposition.

Let's start with a simple example demonstrating how we want logic to work on a superposition. Suppose we have three single-qubit QPU registers, `a`, `b`, and `c`, and we want to implement the following logic: `if (a and b) then invert c`. So if `a` is $|1\rangle$

1 As we noted in the introduction to this part of the book, superposition alone is not enough; a critical second step must ensure that the (effectively) parallel computations we perform can be read out, rather than remaining hidden in the quantum states of our qubits.

and b is |1⟩, then the value of c will be flipped. We saw in Figure 3-20 that we can implement this logic directly with a single Toffoli operation. Visualizing this in circle notation, the Toffoli operation swaps the |3⟩ and |7⟩ values, regardless of what's in them. If b is |0⟩ and a is |1⟩, as in Figure 5-1, the exchanged circles are both empty, so the gate has no effect on c.

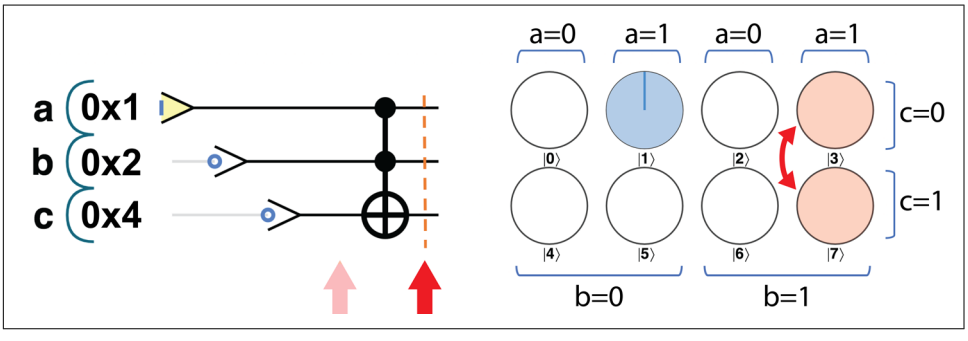


Figure 5-1. When b=0, this operation has no effect

Now suppose that we use a HAD operation (introduced in Chapter 2) to prepare b in a superposition of |0⟩ and |1⟩—what do we want c to do? Properly implemented, this operation should *both* invert and not invert c in superposition, as in Figure 5-2. Toffoli gates also exist for conventional computation, but they certainly couldn't pull off this trick. We need a definitively quantum implementation of the Toffoli gate acting on quantum registers to make this work. The circle notation in Figure 5-2 helps us see what a properly quantum Toffoli gate should do to our superposition input.

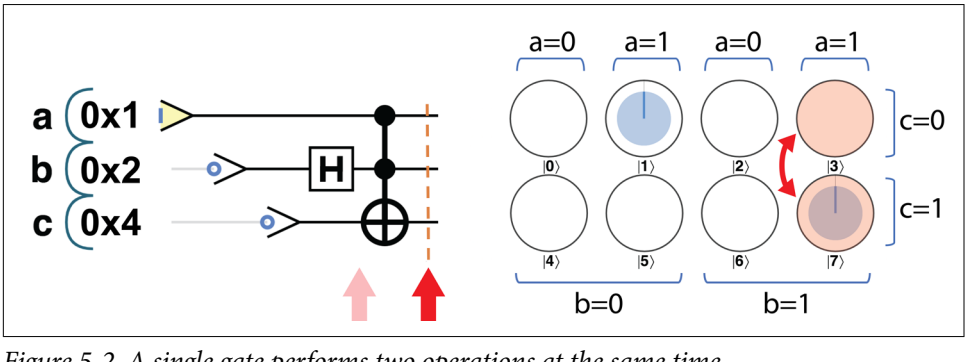


Figure 5-2. A single gate performs two operations at the same time

There are also a few other requirements we will have of operations in order for them to be able to properly operate on qubits within a QPU. Although some of these might not be as obviously necessary as our preceding example, they are worth bearing in mind as we construct a variety of arithmetic and logical operations for QPUs:

Moving and copying data
: As we have already learned, qubits cannot be copied. This is a *huge* difference between quantum and conventional logic. Moving or copying bits is the most common operation a conventional CPU performs. A QPU can move qubits using *exchange* instructions to swap them with other qubits, but no QPU will ever implement a `COPY` instruction. As a result, the = operator, so common for manipulating digital values, cannot be used to assign one qubit-based value to another.

Reversibility and data loss
: Unlike many conventional logic operations, our basic non-`READ` QPU operations *are* reversible (due to details of the laws of quantum mechanics). This imposes significant constraints on the logic and arithmetic we can perform with a QPU, and often drives us to think creatively when trying to reproduce conventional arithmetic operations. `READ` is the only irreversible operation, and you might be tempted to make heavy use of it to build nonreversible operations. Beware! This will make your computation so conventional that it will most likely rob you of any quantum advantage. The simplest way of implementing *any* conventional circuit in our QPU is to replace it with an equivalent conventional circuit only using *reversible* operations, such as Toffoli. We can then implement it virtually as is on a quantum register.[2]

Arithmetic on a QPU

In conventional programming, we rarely use individual logic gates to write programs. Instead, we trust the compiler and CPU to convert our program into the gates needed to perform our desired operations.

Quantum computation is no different. In order to write serious QPU software, we primarily need to learn how to work with *qubytes* and *quantum integers* rather than *qubits*. In this section we'll lay out the intricacies of performing arithmetical operations on a QPU. Just as classical digital logic can be built up from `NAND` gates (a single gate which performs `NOT(b AND b)`), the quantum integer operations we need can be built from the elementary QPU operations we covered in Chapters 2 and 3.

For simplicity, we will diagram and demonstrate the arithmetic operations that follow using four-qubit integers (quantum nibbles, or *qunibbles*, for those who remember the early microcomputer days). However, all of these examples can be extended to larger QPU registers. The size of integers that our arithmetic can handle will depend on the number of qubits available in our QPU or simulator.

2 Any conventional operation implemented with reversible logic using N Toffoli gates can be implemented using $O(N)$ single- and two-qubit operations.

Hands-on: Building Increment and Decrement Operators

Two of the simplest useful integer arithmetic operations we can perform are those for incrementing and decrementing a number. Try running Example 5-1, and stepping through the gates one by one to observe the operation shown in Figure 5-3.

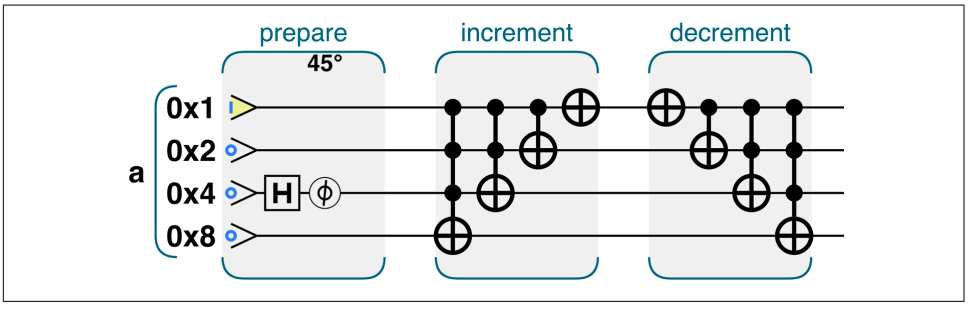


Figure 5-3. Operations performing increment-by-1 and decrement-by-1

The prepare step and initialization value in Figure 5-3 are not part of the actual increment operation, but chosen simply to provide a nonzero input state, so that we can follow the action of operations.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=5-1*.

Example 5-1. Integer increment-by-one operation

```
// Initialize
var num_qubits = 4;
qc.reset(num_qubits);
var a = qint.new(num_qubits, 'a');

// prepare
a.write(1);
a.hadamard(0x4);
a.phase(45, 0x4);

// increment
a.add(1);

// decrement
a.subtract(1);
```

In Example 5-1, we've implemented the increment and decrement operations using an argument of `1` in the QCEngine `add()` and `subtract()` functions.

These implementations satisfy all the requirements we have for them being truly quantum—in particular:

Reversibility
: First, and most obviously, the decrement operation is simply the increment with its constituent operations reversed. This makes sense, but may not be obvious if you're used to conventional logic. In conventional logic devices gates tend to have dedicated inputs and outputs, and simply running a device in reverse is likely to damage it, or at least fail to provide a useful result. As we've noted, for quantum operations reversibility is a critical requirement.

Superposition operation
: Crucially, this implementation of increment also works on inputs in superposition. In Example 5-1, the preparation instructions write the value $|1\rangle$ to a quantum integer and then call `HAD` and `PHASE(45)` on the `0x4` qubit, resulting in our register containing a superposition[3] of $|1\rangle$ and $|5\rangle$, as shown in Figure 5-4.

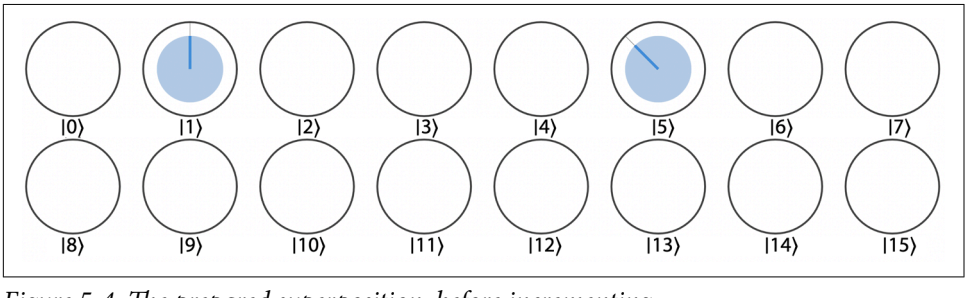


Figure 5-4. The prepared superposition, before incrementing

Now let's try running the increment operation on this input. Doing so transforms the state into a superposition of $|2\rangle$ and $|6\rangle$, where the phase of each value matches its pre-increment counterpart. This is shown in Figure 5-5.

3 We include the 45° phase difference between the two values just so that we can tell them apart.

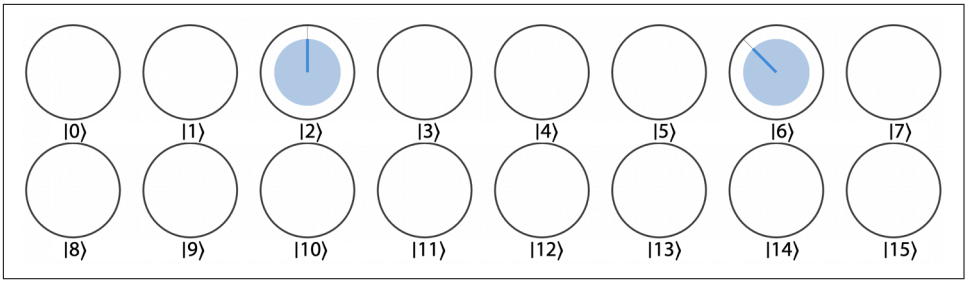


Figure 5-5. The resulting superposition, after incrementing

How does incrementing work? Looking carefully at the operations involved, we can see that it starts by using a three-condition `CNOT` gate to apply "if *all* of the lower bits in the integer are 1, then flip the top bit." This is essentially the same as a conventional arithmetic carry operation. We then repeat the process for each bit in the integer, resulting in a complete "add and carry" operation on all qubits, performed using only multicondition `CNOT` gates.

We can go beyond simple incrementing and decrementing. Try changing the integer values passed to `add()` and `subtract()` in Example 5-1. Any integer will work, though different choices result in different configurations of QPU operations. For example, `add(12)` produces the circuit in Figure 5-6.

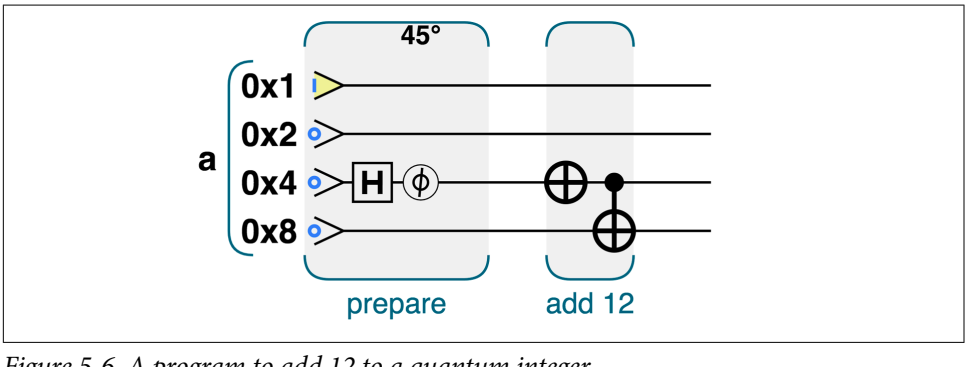


Figure 5-6. A program to add 12 to a quantum integer

In this case, Figure 5-7 shows that the input values |1⟩ and |5⟩ will become |13⟩ and |1⟩, since this program will wrap on overflow (just like conventional integer math).

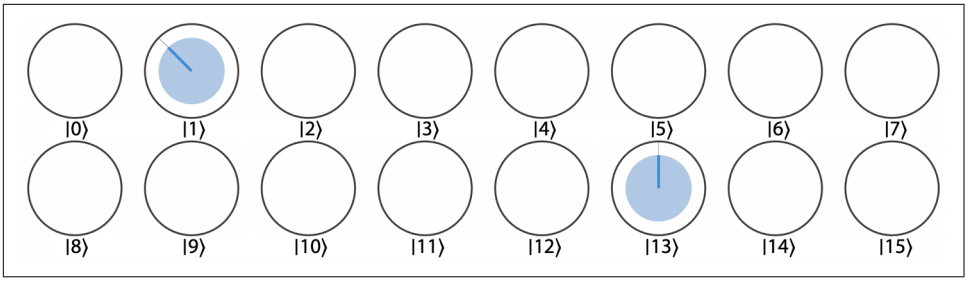


Figure 5-7. add(12) applied to a superposition of 1 and 5 states

The fact that these functions will take any integer brings up an interesting point: this program is performing arithmetic on a conventional digital and a quantum value. We're always adding a *fixed* integer to a quantum register, and have to change the set of gates used to be specific to the particular integer we want to add. What about going a step further—can we perform addition between two *quantum* values?

Adding Two Quantum Integers

Suppose we have two QPU registers `a` and `b` (bearing in mind that each of these could potentially store a superposition of integer values), and we ask for a simple + operation that would store the result of their addition in a new register `c`. This is analogous to how a CPU performs addition with conventional digital registers. However, there's a problem—this approach violates *both* the reversibility and the no-copying restrictions on QPU logic:

- Reversibility is violated by `c = a + b` because the prior contents of `c` are lost.
- No copying is violated because we could be sneaky and perform `b = c - a` to ultimately end up with two copies of whatever superposition might have been in `a`.

To get around this, we will instead implement the `+=` operator, adding one number directly onto another. The code sample in Example 5-2, shown in Figure 5-8, adds two quantum integers together in a reversible manner, whatever superposition they happen to be in. Unlike the previous approach for adding a conventional digital integer to a quantum register, here the gates don't need to be reconfigured every time the input values change.

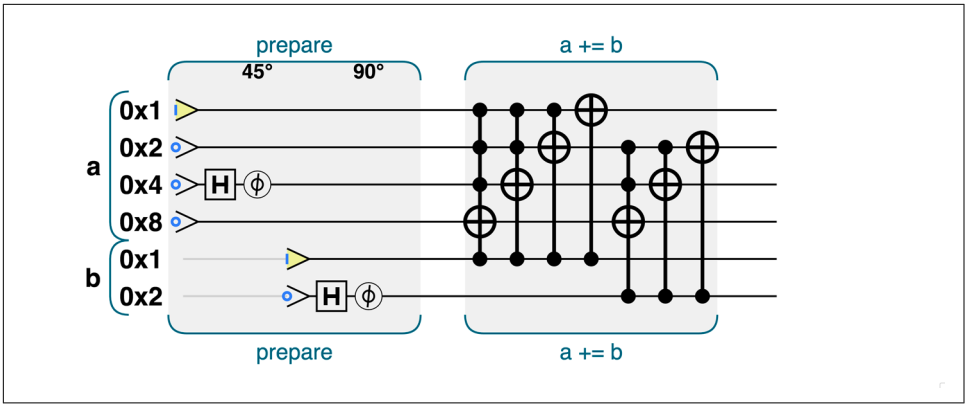


Figure 5-8. Operations assembled to perform a += operation

How does the circuit in Figure 5-8 work? A close look at the gates of this program will reveal that they are simply the integer addition operations from Figure 5-3 and Figure 5-6 applied to `a`, but performed *conditional* on the corresponding qubits of `b`. This allows the values in `b`, even in superposition, to determine the effect of the addition.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=5-2*.

Example 5-2. Adding two quantum integers

```
// a += b
a.add(b);
```

As was the case in Example 5-1, the prepare step in Figure 5-8 is just there to provide test input, and is not part of the operation. Also, we can implement `a -= b` by simply running the `a += b` gates in reverse order.

Negative Integers

So far we've dealt only with adding and subtracting positive integers. How can we represent and manipulate negative integers in a QPU register? Fortunately, we can employ a *two's-complement* binary representation, as used by all modern CPUs and programming languages. We provide a quick review of two's complement here, with qubits specifically in mind.

For a given number of bits, we simply associate half the values with negative numbers, and half with positive numbers. For example, a three-bit register allows us to represent the integers –4, –3, –2, –1, 0, +1, +2, and +3 as shown in Table 5-1.

Table 5-1. Two's complement for binary representation of negative integers

0	1	2	3	–4	–3	–2	–1
000	001	010	011	100	101	110	111

If you've never encountered two's complement before, the association between negative and binary values may seem a bit haphazard, but this particular choice has the surprising benefit that methods for performing elementary arithmetic developed for positive integers will also work out of the box with the two's-complement representations. We can also see from Table 5-1 that the highest-order bit conveniently functions as an indicator of an integer's sign.

A two's-complement encoding works just as well for a register of qubits as it does for a register of bits. Thus, all of the examples in this chapter work equally well with negative values represented using two's complement. We must, of course, be diligent in keeping track of whether or not we're encoding data in QPU registers with two's complement, so that we interpret their binary values correctly.

To negate a number in two's complement, we simply flip all of the bits, and then add 1.[4] The quantum operations for performing this are shown in Figure 5-9, which bears a very strong resemblance to the increment operator presented back in Figure 5-3.

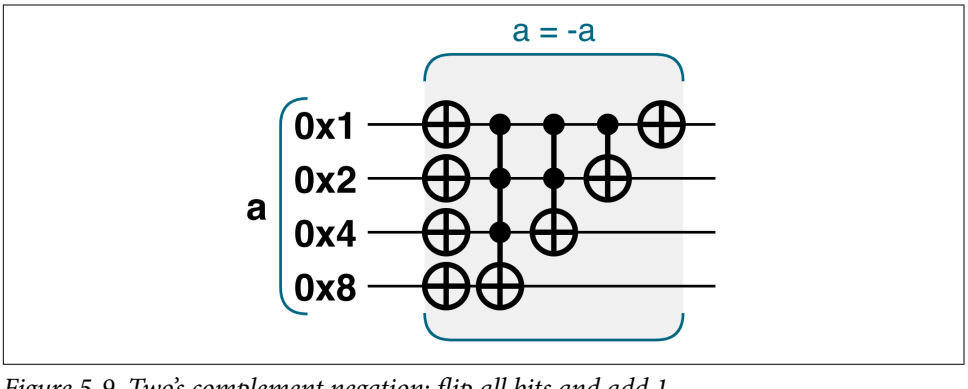


Figure 5-9. Two's complement negation: flip all bits and add 1

4 In this three-bit example, the negation works for all values except –4. As seen in Table 5-1, there is no representation for 4, so the negation leaves it unchanged.

Hands-on: More Complicated Math

Not all arithmetic operations will readily lend themselves to the requirements we demand for QPU operations, such as reversibility and no copying. For example, multiplication is hard to perform reversibly. The code in Example 5-3, shown in Figure 5-10, illustrates a related operation that *can* be constructed to be reversible. Specifically, we square one value and add the result to another.

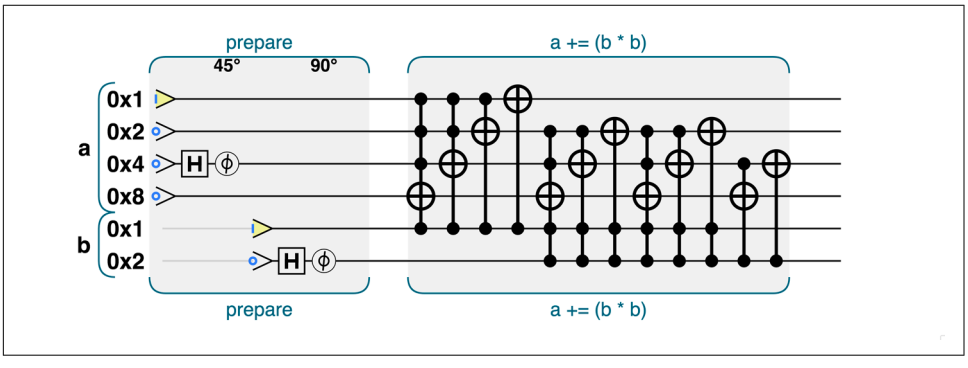


Figure 5-10. Add the square of b to a

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=5-3*.

Example 5-3. Some interesting arithmetic

```
// a += b * b
a.addSquared(b);
```

Performing `a += b * b` as in Figure 5-10 will add the squared value of `b` to `a`.

How does the circuit in Figure 5-10 work? This implementation performs multiplication by using repeated addition, conditional on bits of `b`.

Reversibility is an especially interesting consideration for this example. If we try to naively implement `b = b * b`, we'll quickly discover that there's no suitable combination of reversible operations, as we always end up losing the sign bit. Implementing `a += b * b`, however, is fine, as reversing it simply gives us `a -= b * b`.

Getting Really Quantum

Now that we're equipped with quantum versions of arithmetic circuits, there are other entirely new tricks we can play.

Quantum-Conditional Execution

On a conventional computer, conditional execution causes logic to be performed *if* a value is set. During execution, the CPU will read the value and decide whether to execute the logic. With a QPU, a value can be both *set* and *not set* in superposition, so an operation conditional on that value will both *execute* and *not execute* in superposition. We can use this idea at a higher level, and conditionally execute large pieces of digital logic in superposition.

For example, consider the program in Example 5-4, shown in Figure 5-11, which increments a register of three qubits labeled `b`—but only if another three-qubit register, `a`, holds an integer value in a certain range. By initializing `a` in a superposition of $|1\rangle$ and $|5\rangle$, `b` ends up in a superposition of being *both* incremented and not incremented.

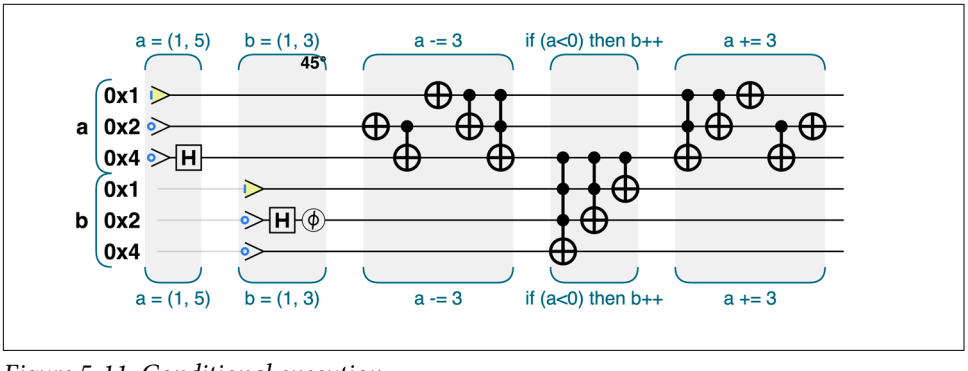


Figure 5-11. Conditional execution

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=5-4*.

Example 5-4. Conditional execution

```
a.subtract(3);
// if high bit of a is set then b += 1
b.add(1, a.bits(0x4));
a.add(3);
```

Note that for some values of a, subtracting 3 from a will cause its lowest-weight qubit to be set. We can use this top bit as a condition for our increment operation. After incrementing b *conditional* on that qubit's value we need to add 3 back to a, in order to restore it to its original state.

Running Example 5-4, circle notation reveals that only parts of the quantum state where a is less than 3 or greater than 6 are incremented.

Only circles in columns 0, 1, 2, *and* 7 in Figure 5-12 are affected by the incrementing logic.

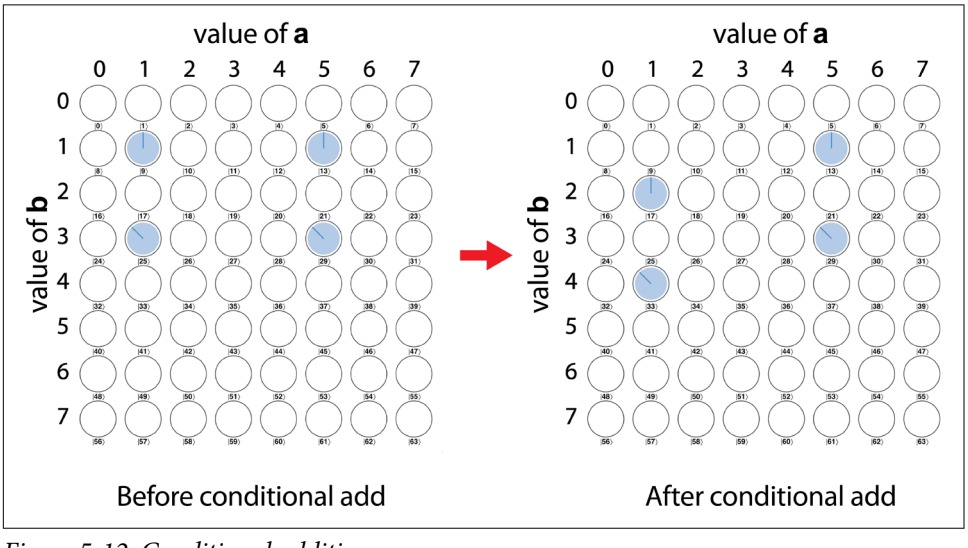


Figure 5-12. Conditional addition

Phase-Encoded Results

Our QPU versions of arithmetic operations can also be modified to encode the output of a calculation in the *relative phases* of the original input qubit register—something completely impossible using conventional bits. Encoding calculation results in a register's relative phases is a crucial skill for programming a QPU, and can help us produce answers able to survive `READ` operations.

Example 5-5, shown in Figure 5-13, modifies Example 5-4 to flip the *phase* in an input register if a is less than 3 and b is equal to 1.

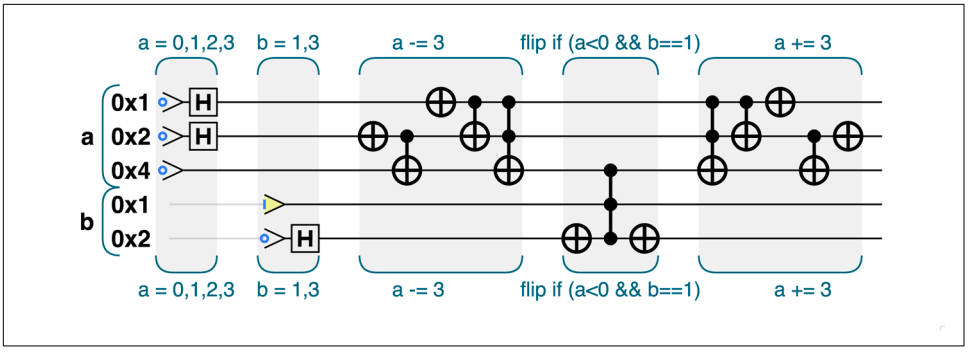


Figure 5-13. Phase-encoding the result

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=5-5*.

Example 5-5. Phase-encoding the result

```
// Flip the phase if a is less than 3 and b is equal to 1
a.subtract(3);
b.not(~1);
qc.phase(180, a.bits(0x4), b.bits());
b.not(~1);
a.add(3);
```

When this operation is complete, the *magnitudes* of the register are unchanged. The probabilities of READing each value remain just as they were, unable to show that this operation was ever performed. However, Figure 5-14 shows that we have used the phases in our input register to "mark" specific states in the superposition as a result of a calculation. We can see this effect most readily in circle notation.

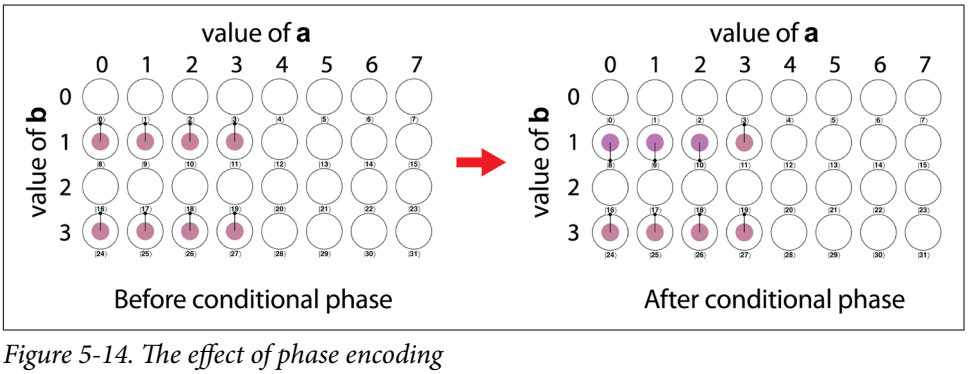


Figure 5-14. The effect of phase encoding

We'll make heavy use of this ability to compute in a register's phases in Chapter 10.

Reversibility and Scratch Qubits

During this chapter we've noted again and again that QPU operations need to be reversible. Naturally, you might ask, "How can I make sure that the arithmetic operations I want to perform are reversible?" Although there's no hard-and-fast way to convert an arithmetical operation into a form that is reversible (and therefore more likely to work on a QPU), a helpful technique is the use of *scratch qubits*.

Scratch qubits are not necessary for encoding the input or output we're interested in, but rather play a temporary role enabling the quantum logic connecting them.

Here's a specific example of an otherwise irreversible operation that can be made reversible with a scratch qubit. Consider trying to find a QPU implementation of `abs(a)`, a function computing the absolute value of a signed integer, as shown in Figure 5-15.

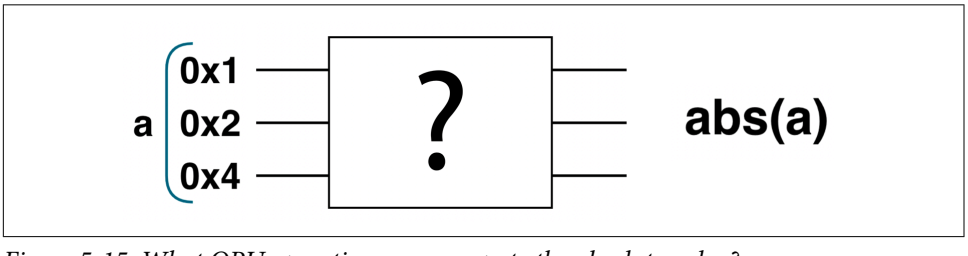


Figure 5-15. What QPU operations can compute the absolute value?

Since we've already seen in Figure 5-9 how to easily negate integers in QPU registers, you might think that implementing `abs(a)` is simple—negate our QPU register dependent on its own sign bit. But any attempt to do this will fail to be reversible (as we might expect, since the mathematical `abs(a)` function itself destroys information about the input's sign). It's not that we'll receive a QPU compile or runtime error; the problem is that no matter how hard we try, we won't ever find a configuration of reversible QPU operations able to produce the desired results.

Enter our scratch qubit! This is used to stash the sign of the integer in `a`. We begin with it initialized to $|0\rangle$, and then use a `CNOT` to flip the scratch qubit dependent on the highest-weight qubit in register `a`. We then perform the negation conditional on the scratch qubit's value (rather than directly on the sign of the `a` register itself), as shown in Figure 5-16.

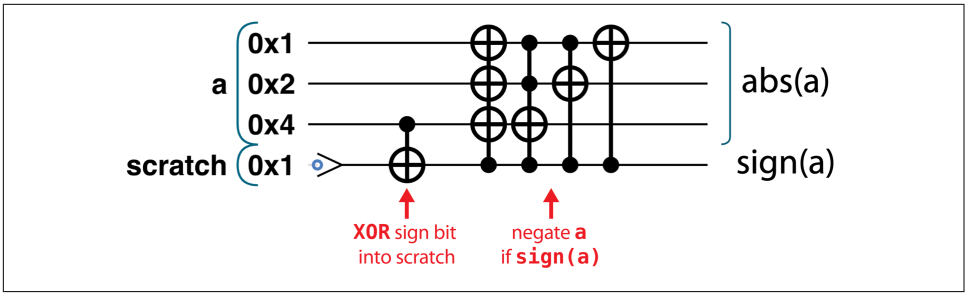


Figure 5-16. Scratch qubits can make an irreversible operation reversible

With the scratch qubit involved, we now *can* find a set of gates that will ultimately implement `abs` on our `a` register. But before we verify in detail that the circuit in Figure 5-16 achieves this, note that the `CNOT` operation we use between our scratch qubit and the `0x4` qubit of `a` (which tells us the sign of its integer value in two's complement) does not *copy* the sign qubit exactly. Instead, the `CNOT` *entangles* the scratch qubit with the sign qubit. This is an important distinction to note, since we also know that QPU operations cannot copy qubits.

Figure 5-17 follows the progression of these operations using an illustrative sample state that has varying relative phases, simply so that we can easily keep track of which circle moves where during the various QPU operations.

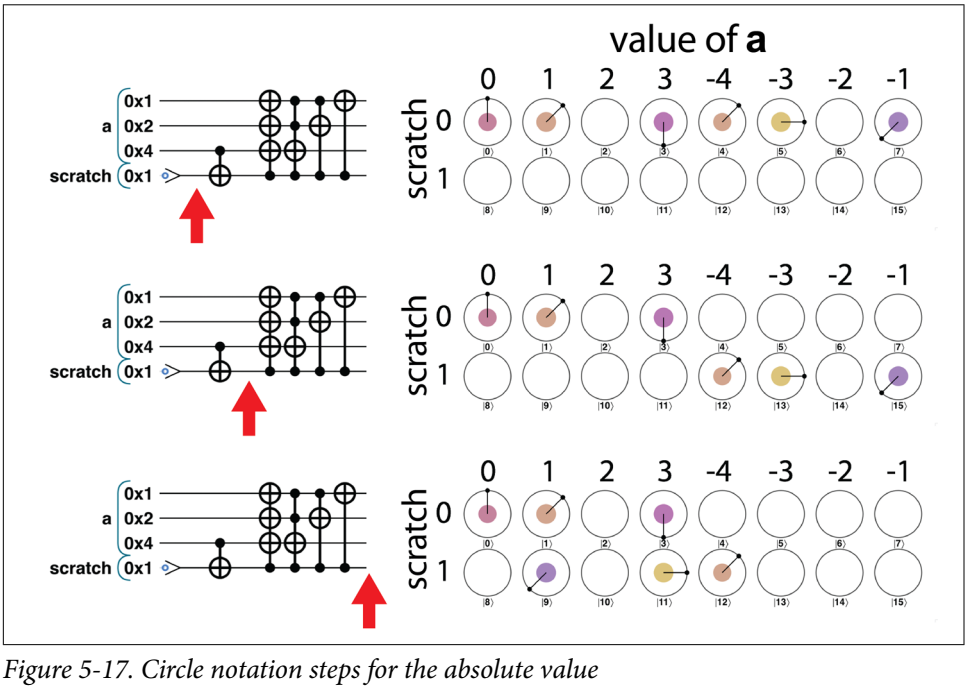


Figure 5-17. Circle notation steps for the absolute value

Note that each value in the superposition that corresponds to `a` starting with a positive or zero value is left unchanged throughout. However, values for which `a` has a negative value are moved first into the second row of circles (corresponding to the scratch qubit having been conditionally flipped), and then into the corresponding positive values (as required by `abs`), *while remaining in the "scratch = 1" row*.[5]

Following the action of the scratch qubit in circle notation visually demonstrates how it solves our irreversibility issues. If we tried to perform `abs` *without* a scratch qubit, our circle notation in Figure 5-17 would only have a single row, and trying to move the negative-value circles into the corresponding positive values would obliterate the magnitudes and phases already held there—such that we could never later figure out what the original state was (i.e., we would have an irreversible operation). The scratch qubit gives us an extra row we can move into and then across in, leaving everything from the original state untouched and recoverable.[6]

Uncomputing

Although scratch qubits are frequently a necessity, they do tend to get tangled up with our QPU registers. More precisely, they tend to get *entangled*. This introduces two related problems. First, scratch qubits rarely end up back in an all-zero state. This is bad news, since we now need to reset these scratch qubits to make them usable again further down the line in our QPU.

"No problem!" you may think, "I'll just `READ` and `NOT` them as necessary, like we learned in Chapter 1." But this would lead you to encounter the second problem. Because using scratch qubits almost always results in them becoming entangled with the qubits in our output register (this is certainly the case in Figure 5-17), performing a `READ` on them can have a disastrous effect on the output state they helped us create. Recall from Chapter 3 that when qubits are entangled, any operations you perform on one have unavoidable effects on the others. In the case of Figure 5-17, attempting to reset the scratch qubit to $|0\rangle$ (as we must if we ever want to use it again in our QPU) destroys half of the quantum state of `a`!

Fortunately, there's a trick that solves this problem: it's called *uncomputing*. The idea of uncomputing is to reverse the operations that entangled the scratch qubit, returning it to its initial, disentangled $|0\rangle$ state. In our `abs` example, this means reversing all the `abs(a)` logic involving `a` and the scratch qubit. Brilliant! We have our scratch

5 Note that in this case –4 is unchanged, which is correct for a three-bit number in a two's-complement encoding.

6 This is quite a remarkable feat. Keeping track of the original state using conventional bits would require an exponentially vast number of bits, but just adding one scratch qubit allows us to do so perfectly.

qubit back in a $|0\rangle$ state. Unfortunately, we will, of course, also have completely undone all the hard work of our absolute-value calculation.

What good is getting back our scratch qubit if we've also undone our answer? Thanks to the no-copying constraints faced by qubits, we can't even copy the value stored in `a` to another register before uncomputing it. However, the reason uncomputing is not completely self-defeating is that *we will often make use of our answer within register a before we uncompute*. In most cases, functions such as `abs` are used as part of a larger arithmetic computation. For example, we might *actually* want to implement "add the absolute value of `a` to `b`." We can make this operation reversible and save our scratch qubit with the circuit shown in Figure 5-18.

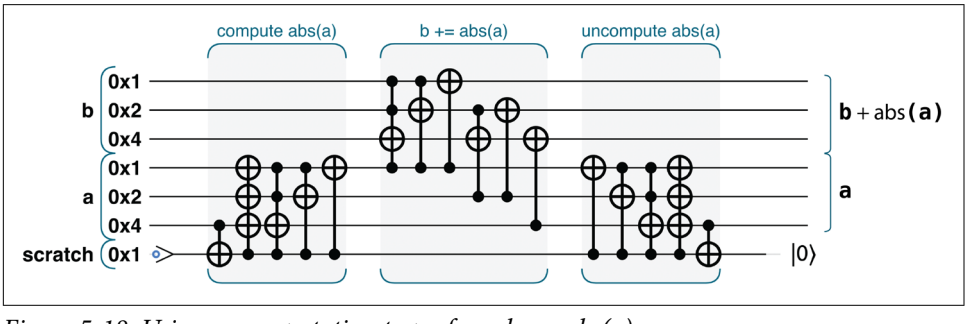


Figure 5-18. Using uncomputation to perform b += abs(a)

After these operations, `a` and `b` are likely entangled with one another; however, the scratch qubit has been returned to its disentangled $|0\rangle$ state, ready to be used in some other operation. The larger operation *is* reversible, even though `abs` itself is not. In quantum computation, this trick is extremely common: use temporary scratch qubits to perform an otherwise-irreversible computation, perform other operations conditional on the result, and then uncompute.

In fact, although we cannot just simply "copy" the absolute value into another register before uncomputing its action, we can do something similar by initializing `b` to 0 before the addition in Figure 5-18. We can actually achieve the same result more simply using a `CNOT` operation instead of addition, as shown in Figure 5-19.

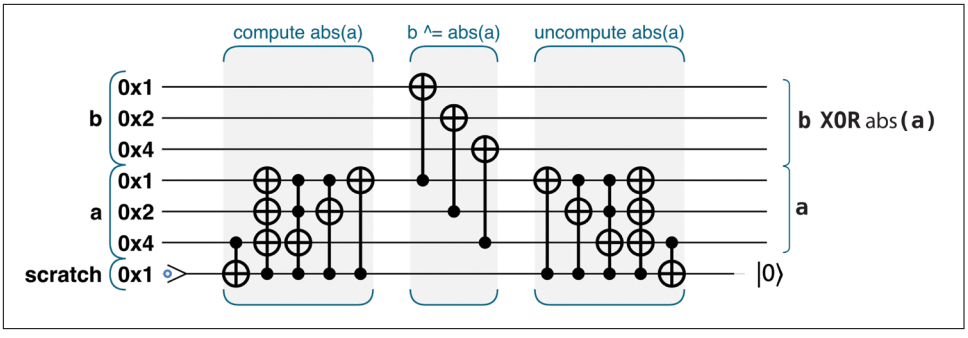


Figure 5-19. Using uncomputation to produce b xor abs(a)

Another extremely common application of uncomputation involves performing a computation (potentially using scratch qubits), storing the results in the relative phases of the output register, and then uncomputing the result. So long as the initial computation (and thus also the final uncomputation step) does not interfere with the relative phases of the output registers, they will survive the process intact. We make use of this trick in Chapter 6.

As an example, the circuit in Figure 5-20 performs the operation "flip the phase of any value where `abs(a)` equals 1." We compute the absolute value using a scratch qubit, flip the phase of the output register only if the value is 1, and then uncompute.

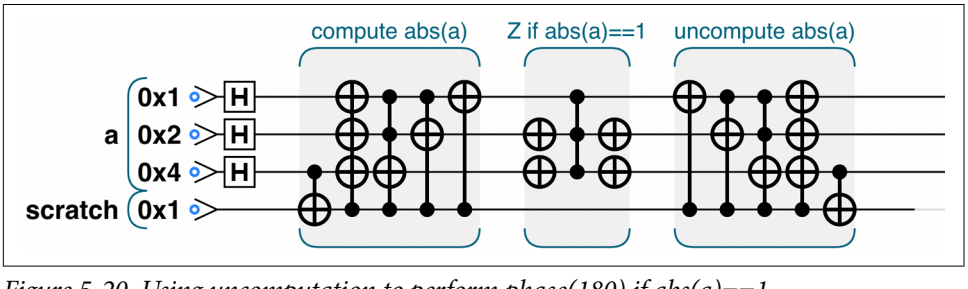


Figure 5-20. Using uncomputation to perform phase(180) if abs(a)==1

In Figure 5-21 we follow this program step by step in circle notation.

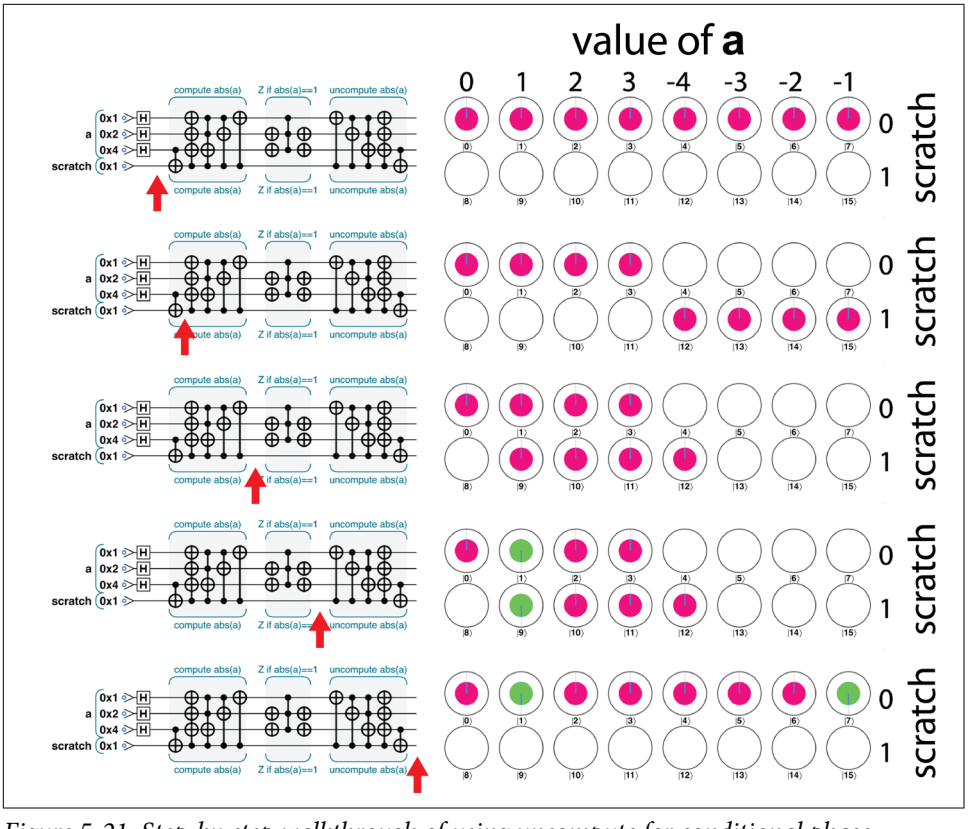


Figure 5-21. Step-by-step walkthrough of using uncompute for conditional phase inversion

Mapping Boolean Logic to QPU Operations

Just as digital arithmetic is built from digital logic gates, to see in detail how the circuits for our basic QPU arithmetic work we rely on a toolbox of programmable quantum logic. In this section we'll highlight the quantum analogs of some low-level digital logic gates.

Basic Quantum Logic

In digital logic, there are a few basic logic gates that can be used to build all of the others. For example, if you have just a `NAND` gate, you can use it to create `AND`, `OR`, `NOT`, and `XOR`, which can be combined into any logical function you wish.

Note that the `NAND` gates in Figure 5-22 can have any number of inputs. With a single input (the simplest case), `NAND` performs a `NOT`.

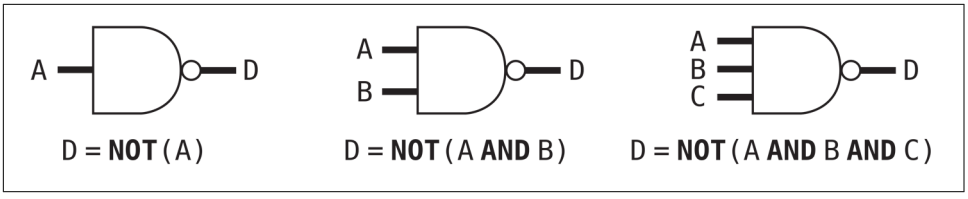


Figure 5-22. Digital NAND gates in different sizes

In quantum computation we can similarly start with one versatile gate, as in Figure 5-23, and build our quantum digital logic from it. To accomplish this, we'll use a multiple-condition `CNOT` gate: the Toffoli gate. As with the `NAND` gate, we can vary the number of condition inputs to expand the logic performed, as shown in Example 5-6.

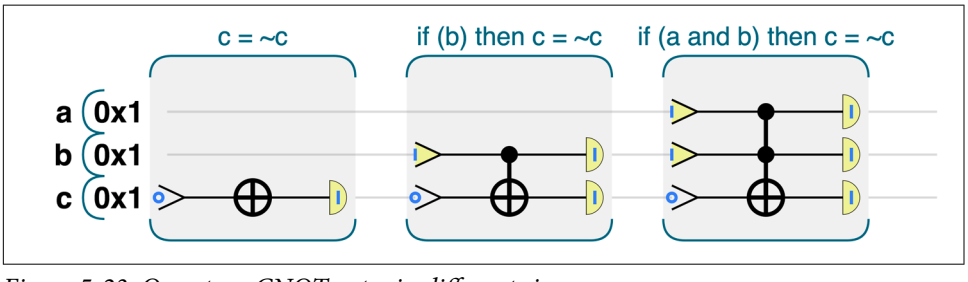


Figure 5-23. Quantum CNOT gates in different sizes

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=5-6*.

Example 5-6. Logic using CNOT gates

```
// c = ~c
c.write(0);
c.not();
c.read();

// if (b) then c = ~c
qc.write(2, 2|4);
c.cnot(b);
qc.read(2|4);

// if (a and b) then c = ~c
qc.write(1|2);
qc.cnot(4, 1|2);
qc.read(1|2|4);
```

Note that this performs *almost* the same operation as `NAND`. We can express its digital-logic equivalent using `AND` and `XOR`, as shown in Figure 5-24.

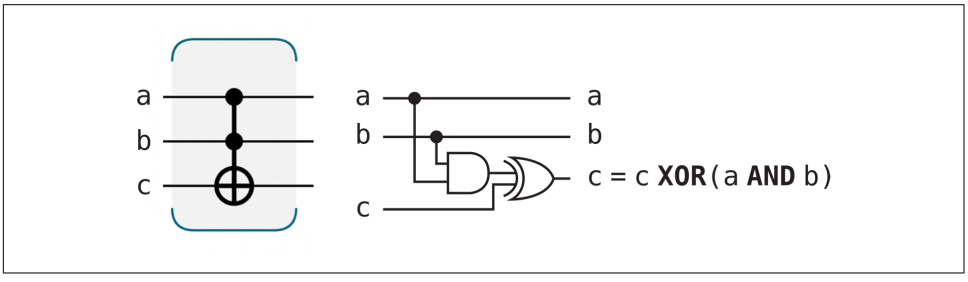


Figure 5-24. The exact digital logic equivalent of our multicondition CNOT gate

Armed with this gate, we can produce QPU equivalents of a wide variety of logical functions, as shown in Figure 5-25.

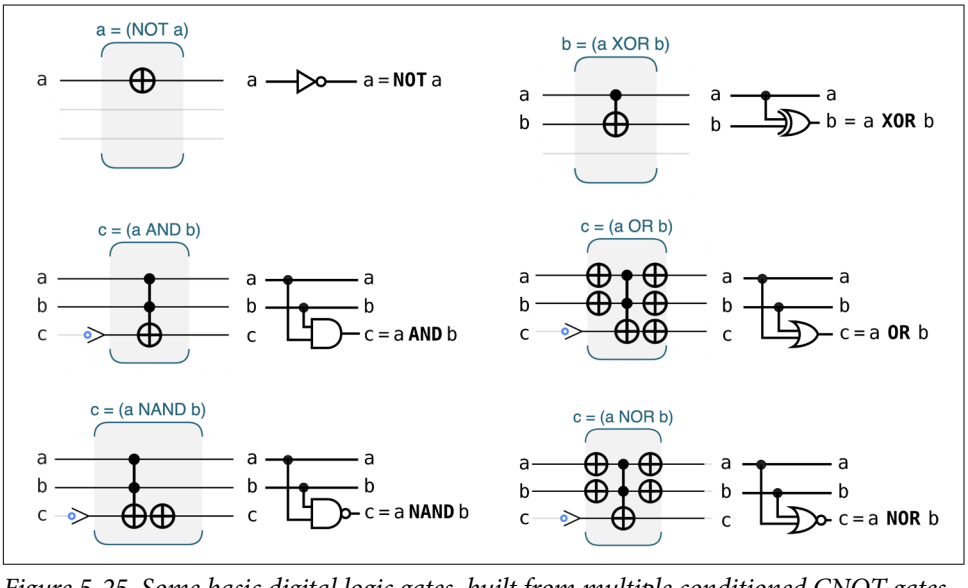


Figure 5-25. Some basic digital logic gates, built from multiple conditioned CNOT gates

Notice that in some cases in order to get the desired logic function we need to add an extra scratch qubit, initialized to $|0\rangle$. One substantial challenge in quantum computation is reducing the number of scratch qubits needed to perform a specific computation.

Conclusion

The ability to perform digital logic in superposition is a core part of most QPU algorithms. In this chapter, we have taken a close look at ways to manipulate quantum data and even to perform conditional operations within superpositions.

Performing digital logic in superposition is of limited use unless we can extract information from the resulting state in a useful way. Recall that if we try to `READ` a superposition of solutions to an arithmetic problem, we'll randomly obtain just one of them. In the next chapter, we will explore a QPU primitive allowing us to reliably extract output from quantum superpositions, known as *amplitude amplification*.

CHAPTER 6

Amplitude Amplification

In the previous chapter we showed how to build conventional arithmetic and logical operations that utilize the power of superposition. But when using a QPU, being able to perform computations in superposition is useless if we don't have something clever up our sleeves to make sure that we're actually able to `READ` out a solution.

In this chapter we introduce the first *quantum primitive* allowing us to manipulate superpositions into a form that we can reliably `READ`. We've already noted that there are many such primitives, each suited to different kinds of problems. The first we will cover is *amplitude amplification*.[1]

Hands-on: Converting Between Phase and Magnitude

Very simply, amplitude amplification is a tool that converts inaccessible *phase* differences inside a QPU register into `READ`able *magnitude* differences (and vice versa). As a QPU tool, it's simple, elegant, powerful, and very useful.

Given that amplitude amplification converts phase differences into magnitudes, you might think that *magnitude* amplification would be a better name. However, "amplitude amplification" is commonly used in the wider literature to refer to the kind of primitive we describe here.

For example, suppose we have a four-qubit QPU register containing one of the three quantum states (A, B, or C), shown in Figure 6-1, but we don't know which one.

1 In this book, we use the term *amplitude amplification* in a slightly different way than the term is used in the academic literature. We cover the exact difference in Chapter 14.

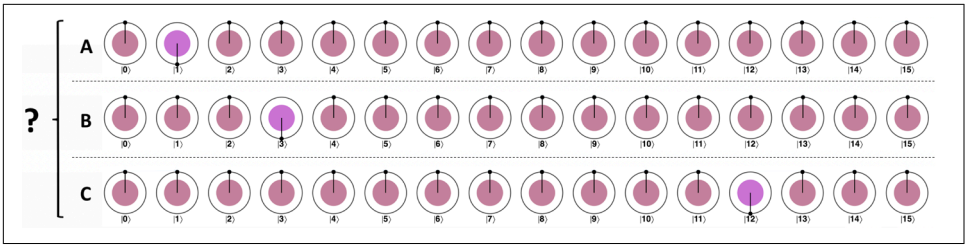


Figure 6-1. Each state has a single flipped value

These three states are clearly distinct in that each one has a different certain value phase-flipped. We'll call that the *marked value*. However, as all of the values in the register have the same magnitude, reading a QPU register in any of these states will return an evenly distributed random number, revealing nothing about which of the three states we started with. At the same time, such `READ`s will destroy the phase information in the register.

But with a single QPU subroutine we can reveal the hidden phase information. We call this subroutine the `mirror` operation (we'll see why later), and you can see its action for yourself by running the sample code in Example 6-1, shown in Figure 6-2.

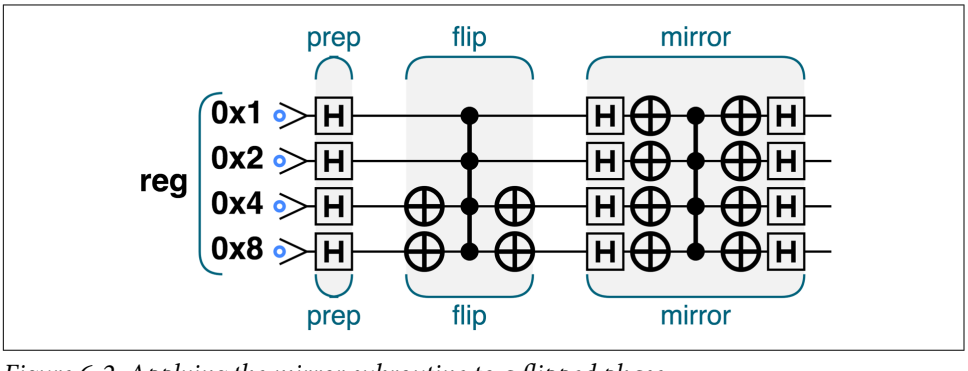


Figure 6-2. Applying the mirror subroutine to a flipped phase

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=6-1*.

Example 6-1. Applying the mirror subroutine to a flipped phase

```
var number_to_flip = 3; // The 'marked' value

var num_qubits = 4;
qc.reset(num_qubits);
var reg = qint.new(num_qubits, 'reg')
reg.write(0);
```

```
reg.hadamard();

// Flip the marked value
reg.not(~number_to_flip);
reg.cphase(180);
reg.not(~number_to_flip);

reg.Grover();
```

Note that before applying the `mirror` subroutine we first `flip`, which takes our register initially in state $|0\rangle$ and *marks* one of its values. You can alter which value is flipped in the preceding code sample by changing the `number_to_flip` variable.

Applying the `mirror` subroutine to the A, B, and C states from Figure 6-1, we obtain the results in Figure 6-3.

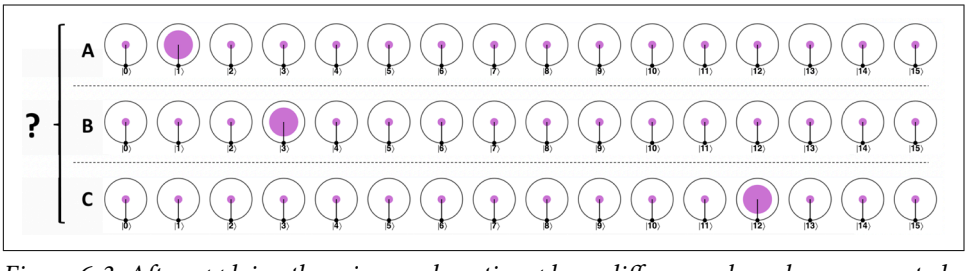


Figure 6-3. After applying the mirror subroutine, phase differences have been converted into magnitude differences

The magnitudes within each state are now very different, and performing a `READ` on the QPU register is very likely (though not certain) to reveal which value had its phase flipped, and hence which of the three states our register had been in. Instead of all values having the same probability of 6.25%, the marked value now has a `READ` probability of about 47.3%, with the nonmarked values being at about 3.5%. At this point, `READ`ing the register gives us an almost 50% chance of obtaining the value that had its phase flipped. That's still not great.

Notice that although the `mirror` subroutine changed the magnitude of the marked value, its *phase* is now the same as the rest of the register. In a sense, `mirror` has *converted* the phase difference into a magnitude difference.

The `mirror` operation is commonly called the "Grover iteration" in the quantum computing literature. Grover's algorithm for an unstructured database search was the first algorithm implementing the `flip-mirror` routine, and in fact, the amplitude amplification primitive that we cover here is a generalization of Grover's original algorithm. We've chosen to call the operation `mirror` to make it easier for the reader to recall its effect.

Can we repeat the operation again to try to further improve our probability of success? Suppose we have the B state from Figure 6-1 (i.e., `flip` acted on the $|3\rangle$ value). Applying the `mirror` subroutine again simply leaves us where we started, converting the magnitude differences back into differences of phase. However, suppose that before reapplying `mirror` we also reapply the `flip` subroutine (to re-flip the marked value). This starts us out with another phase difference before our second `mirror` application. Figure 6-4 shows what we get if we apply the *whole* `flip-mirror` combination twice.

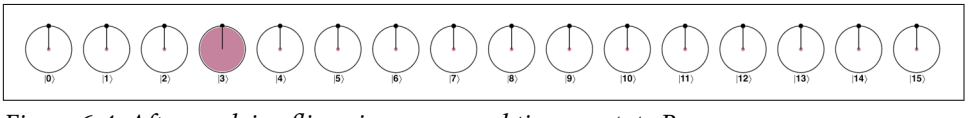


Figure 6-4. After applying flip-mirror a second time on state B

Following two applications of `flip-mirror`, the probability of finding our marked value has jumped from 47.3% to 90.8%!

The Amplitude Amplification Iteration

Together, the `flip` and `mirror` subroutines are a powerful combination. `flip` allows us to target a value of the register and distinguish its phase from the others. `mirror` then converts this phase difference into a magnitude difference. We'll refer to this combined operation, shown in Figure 6-5, as an amplitude amplification (AA) iteration.

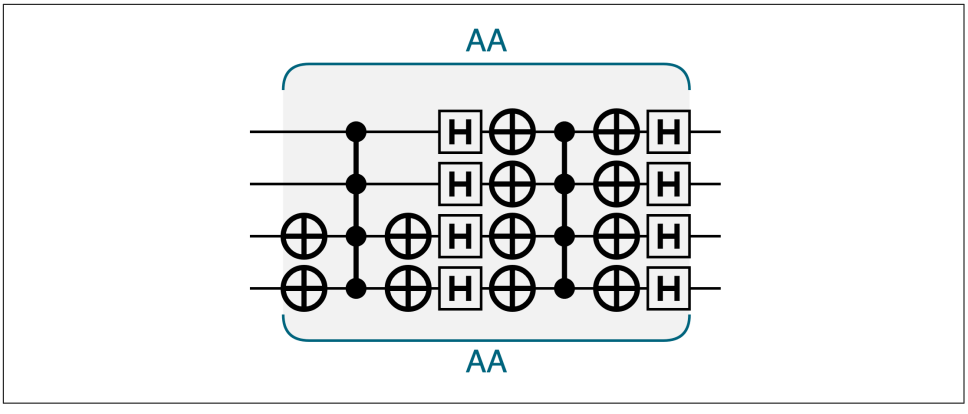


Figure 6-5. A single AA iteration

You've probably noticed that the AA operation assumes that we know which value we want to amplify—it's hardwired into which value the `flip` subroutine affects. This may seem to defeat the whole point of amplitude amplification—if we already know which values we should amplify, why do we need to find them? We've used the `flip` subroutine as the simplest possible example of a subroutine flipping the phase of selected values within our QPU register. In a real application, `flip` would be replaced with some more complex subroutine performing a combination of phase flips representing logic specific to that application. In Chapter 10 we'll show in more detail how computations can be performed purely on the phases of a QPU register. When applications make use of this kind of *phase logic* they can take the place of `flip`, and amplitude amplification becomes an extremely useful tool.

The key point is that although we discuss using `flip` alongside the `mirror` subroutine, compounding more complex phase-altering subroutines with `mirror` still converts phase alterations into magnitude differences.

More Iterations?

In Figure 6-4 we've applied two AA iterations to the B state, leaving us with a 90.8% success probability of observing the marked value. Can we continue applying AA iterations to bring that even closer to 100%? This is easy to try. The sample code in Example 6-2 repeats the AA steps a specified number of times. By varying the `number_of_iterations` variable in the sample code, we can run as many AA iterations as we like.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=6-2*.

Example 6-2. Repeated amplitude amplification iterations

```
var number_to_flip = 3;
var number_of_iterations = 4;

var num_qubits = 4;
qc.reset(num_qubits);
var reg = qint.new(num_qubits, 'reg')
reg.write(0);
reg.hadamard();

for (var i = 0; i < number_of_iterations; ++i)
{
    // Flip the marked value
    reg.not(~number_to_flip);
    reg.cphase(180);
    reg.not(~number_to_flip);

    reg.Grover();

    // Peek at the probability
    var prob = reg.peekProbability(number_to_flip);
}
```

Figure 6-6 shows the result of running this code with `number_of_iterations=4`, so that we `flip` and then `mirror` our register four consecutive times.

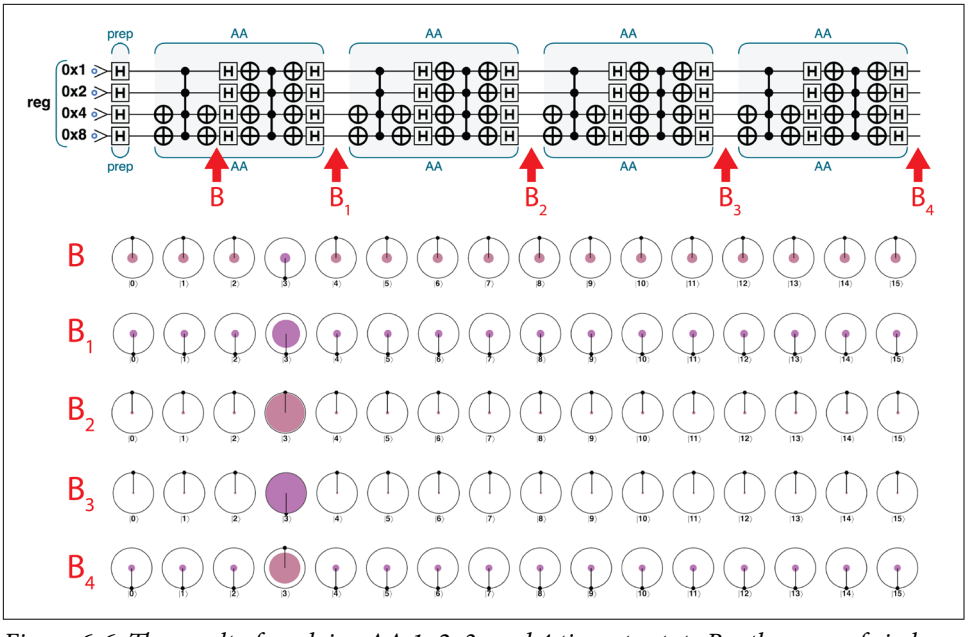


Figure 6-6. The result of applying AA 1, 2, 3, and 4 times to state B—the rows of circles show the state of our QPU register at the positions marked in the circuit

Let's follow the action of each successive AA iteration. After the `flip` of the first AA we begin with the state B from Figure 6-1. As we've already seen, B_1 has a success probability of 47.3%, while after two iterations, B_2 has a probability of 90.8%.

The third iteration brings us to 96.1%, but notice that in B_3 the marked state is out of phase with the others, so our next `flip` subroutine will cause all phases to be *in alignment*. At that point, we will have a magnitude difference but no phase difference, so further AA iterations will start diminishing magnitude differences until we end up with the original state.

Sure enough, by the time we get to B_4 our chances of successfully reading out the marked state are way down to 58.2%, and they'll continue to drop if we apply more AA iterations.

So how many AA iterations should we apply to maximize the chances of `READ`ing out our marked value correctly?

The plot in Figure 6-7 shows that as we continually loop through our iterations, the probability of reading the marked value oscillates in a predictable way. We can see that to maximize the chance of getting the correct result, we're best off waiting for the 9th or 15th iteration, giving a probability of finding the marked value of 99.9563%.

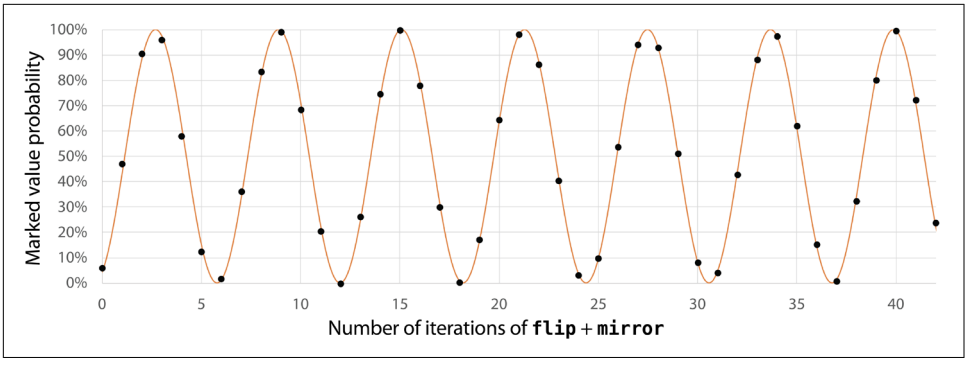


Figure 6-7. Probability of reading the marked value versus the number of AA iterations

On the other hand, if each iteration is expensive to perform (we'll see some cases of this later), we can stop after three and attempt to harvest the answer at 96.1%. Even if we fail and need to repeat the entire QPU program, we'll have a 99.848% chance that one of the two attempts will succeed, only costing us at most six total iterations.

In general, there's a simple and useful equation allowing us to determine the number of AA iterations, N_{AA}, that we should perform to get the highest probability *within the first oscillation* made by the success probability (this would be the 96.1% success probability at $N_{AA} = 3$ in our previous example). This is shown in Equation 6-1, where n is the number of qubits.

Equation 6-1. Optimal number of iterations in amplitude amplification

$$N_{AA} = \left\lfloor \frac{\pi\sqrt{2^n}}{4} \right\rfloor$$

We now have a tool that can convert a single phase difference within a QPU register into a detectable magnitude difference. But what if our register has several values with different phases? It's easy to imagine that subroutines more complicated than `flip` might alter the phases of multiple values in the register. Fortunately, our AA iterations can handle this more general case.

Multiple Flipped Entries

With a small modification to the circuit in Example 6-1 we can try running multiple AA iterations on a register having any number of phase-flipped values. In Example 6-3 you can use the `n2f` variable to set which values in the register should be flipped by the `flip` subroutine inside each AA operation. As before, you can also adjust the number of AA iterations performed using the `number_of_iterations` variable.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=6-3*.

Example 6-3. Amplitude amplification iterations with multiple values flipped

```
var n2f = [0,1,2];              // Which values to flip
var number_of_iterations = 50; // The number of Grover iterations

var num_qubits = 4;
qc.reset(num_qubits);
var reg = qint.new(num_qubits, 'reg')
reg.write(0);
reg.hadamard();

for (var i = 0; i < number_of_iterations; ++i)
{
    // Flip the marked value
    for (var j = 0; j < n2f.length; ++j)
    {
        var marked_term = n2f[j];
        reg.not(~marked_term);
        reg.cphase(180);
        reg.not(~marked_term);
    }

    reg.Grover();

    var prob = 0;
    for (var j = 0; j < n2f.length; ++j)
    {
        var marked_term = n2f[j];
        prob += reg.peekProbability(marked_term);
    }
    qc.print('iters: '+i+' prob: '+prob);
}
```

By running this sample with a single value flipped (e.g., `n2f = [4]`, as shown in Figure 6-8), we can reproduce our earlier results, where increasing the number of AA iterations causes the probability of `READ`ing the "marked" value to vary sinusoidally, as illustrated in Figure 6-9.

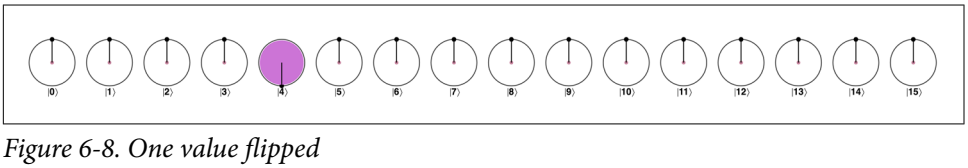


Figure 6-8. One value flipped

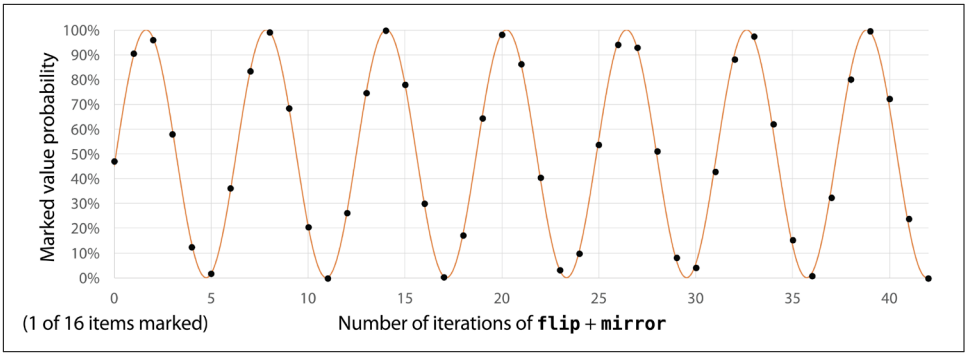


Figure 6-9. Repeated AA iterations with 1 value of 16 flipped

But now let's instead flip *two* values (e.g., `n2f=[4,7]`, as shown in Figure 6-10). In this case we ideally want to end up with the QPU register configured so that we will `READ` either of the two phase-flipped values, with zero possibility of `READ`ing any others. Applying multiple AA iterations just like we did for one marked state (albeit with two `flip` subroutines run during each iteration—one for each marked state) yields the results shown in Figure 6-11.

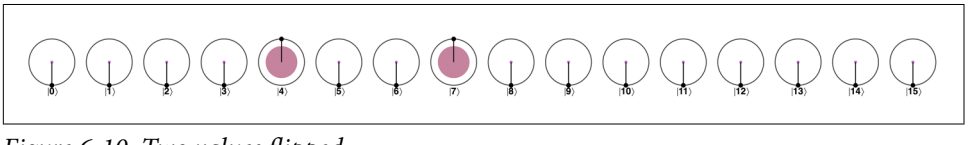


Figure 6-10. Two values flipped

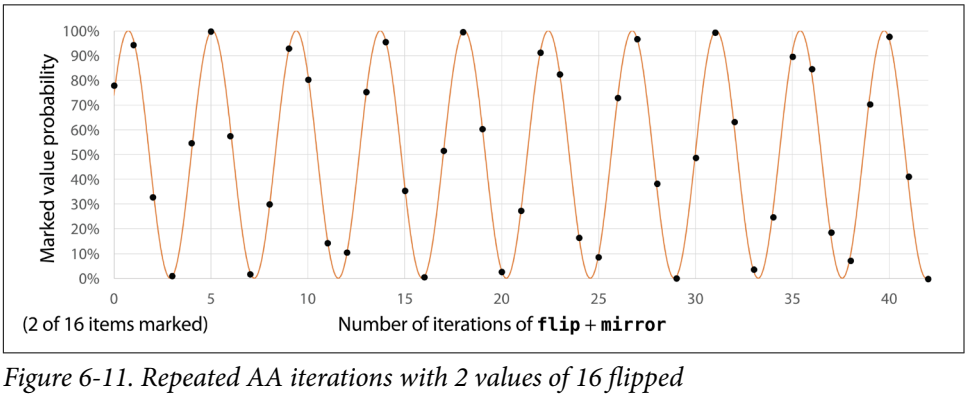


Figure 6-11. Repeated AA iterations with 2 values of 16 flipped

Note that in Figure 6-11 the probability shown on the y-axis is the probability of obtaining *either one of the (two) marked values* if we were to `READ` our register.

Although we still get a sinusoidally varying chance of success, comparing this with the similar plot for only *one* phase-flipped value in Figure 6-7 you'll notice that the frequency of the sinusoidal wave has increased.

With three values flipped (e.g., `n2f=[4,7,8]`, as shown in Figure 6-12), the wave's frequency continues to increase, as you can see in Figure 6-13.

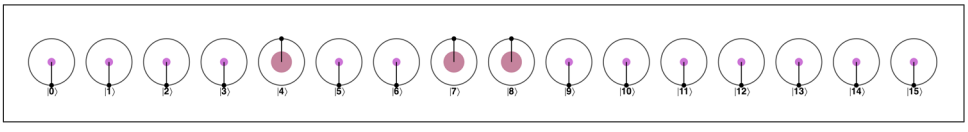


Figure 6-12. Three values flipped

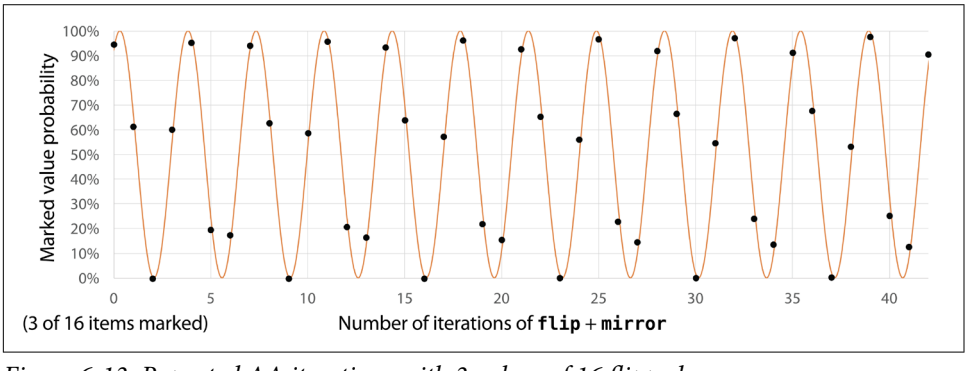


Figure 6-13. Repeated AA iterations with 3 values of 16 flipped

When we have 4 of the 16 values flipped, as shown in Figure 6-14, something interesting happens. As you can see in Figure 6-15, the wave's frequency becomes such that the probability of us `READ`ing one of the marked values repeats with every third AA iteration that we apply. This ends up meaning that the very first iteration gives us 100% probability of success.

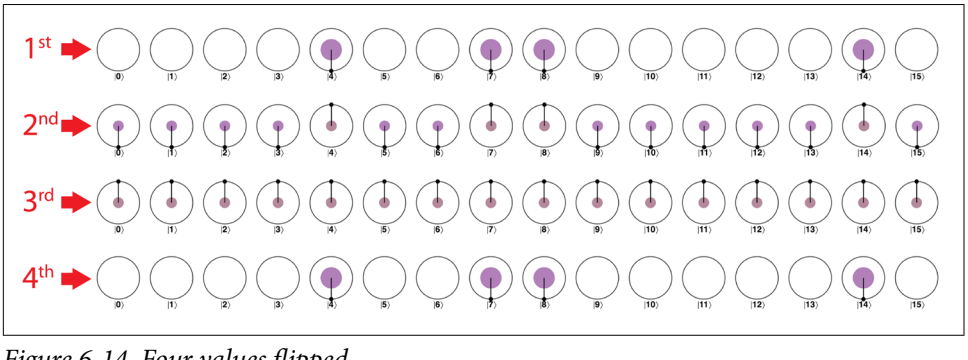


Figure 6-14. Four values flipped

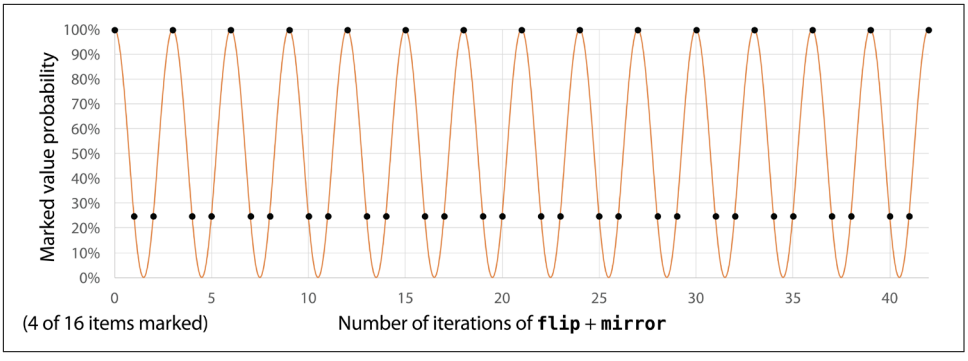


Figure 6-15. Repeated AA iterations with 4 values of 16 flipped

This trend continues for up to seven flipped values, as illustrated in Figures 6-16 and 6-17.

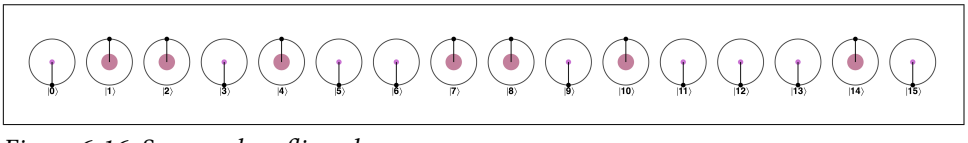


Figure 6-16. Seven values flipped

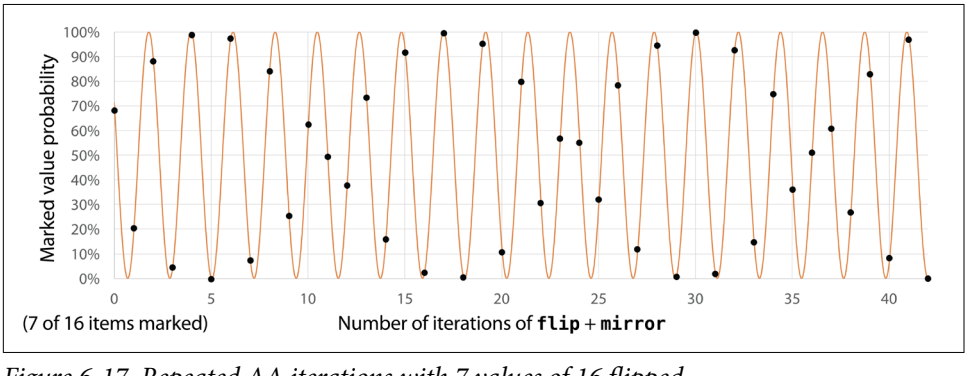


Figure 6-17. Repeated AA iterations with 7 values of 16 flipped

Of course, with 7 of our 16 values being marked, even getting a correct `READ` value might not provide us with much useful information.

Everything comes to a halt in Figure 6-19, where we have 8 of our 16 values flipped as shown in Figure 6-18. As has been mentioned in previous chapters, only the *relative* phase matters for quantum states. Because of this, flipping half of the values to mark them is physically the same as flipping the other half. The AA iterations fail completely here, and we're left with just as much chance of `READ`ing out any value in the register.

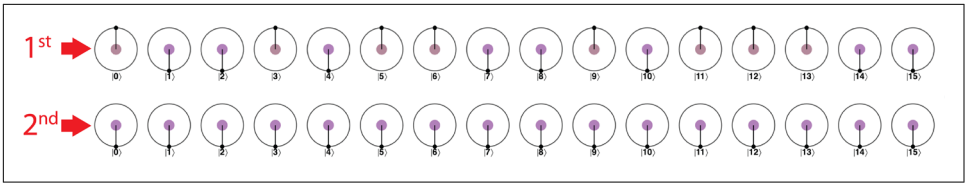


Figure 6-18. Eight values flipped

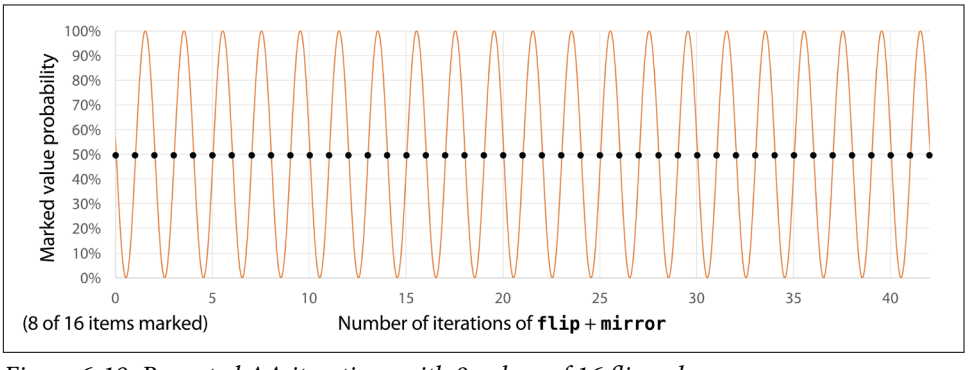


Figure 6-19. Repeated AA iterations with 8 values of 16 flipped

When 50% of the register values are marked, we could continue applying AA operations as much as we liked and we'd never do better than `READ`ing out random numbers.

Note that by symmetry we don't really need to consider what happens if we have more than 50% of our register values phase-flipped. For example, if 12 of our 16 values are marked, attempting to read "success" is identical to what we observed when marking 4 of them, but with what we consider success and failure flipped.

Interestingly, our exploration shows that the frequency with which our chance of success oscillates depends only on the *number* of flipped values, not *which* values are flipped. In fact, we can extend Equation 6-1 to also hold for when we have multiple marked items, as shown in Equation 6-2 (where n is the number of qubits and m is the number of marked items).

Equation 6-2. Optimal number of iterations for multiple flipped phases

$$N_{AA} = \left\lfloor \frac{\pi}{4}\sqrt{\frac{2^n}{m}} \right\rfloor$$

So long as we know m, we can use this expression to decide how many AA iterations we should apply to amplify the magnitudes of the flipped values to give a high probability of success. This raises an interesting point: if we *don't* know how many states are flipped, then how can we know how many AA iterations to apply to maximize our chance of success? When we come to employ amplitude amplification to applications in Chapter 10, we'll revisit this question and see how other primitives can help.

Using Amplitude Amplification

Hopefully you now have an idea of amplitude amplification's capabilities. Being able to convert unREADable phases into READable magnitudes definitely sounds useful, but how do we actually employ this? Amplitude amplification finds utility in a number of ways, but one strikingly pragmatic use is as part of a *Quantum Sum Estimation* technique.

AA and QFT as Sum Estimation

We've seen that the *frequency* with which probabilities fluctuate in these AA iteration examples depends on the number of flipped values. In the next chapter, we'll introduce the Quantum Fourier Transform (QFT)—a QPU primitive allowing us to `READ` out the frequency with which values vary in a quantum register.

It turns out that by combining the AA and QFT primitives, we can devise a circuit allowing us to `READ` not just one of our marked values, but a value corresponding to *how many marked values* in our initial register state were flipped. This is a form of Quantum Sum Estimation. We'll discuss Quantum Sum Estimation fully in Chapter 11, but mention it here to give a feeling for just how useful amplitude amplification can be.

Speeding Up Conventional Algorithms with AA

It turns out that AA can be used as a subroutine on many conventional algorithms, providing a quadratic performance speedup. The problems that AA can be applied to are those invoking a subroutine that repeatedly checks the validity of a solution. Examples of this type of problem are *boolean satisfiability* and finding *global* and *local minima*.

As we have seen, the AA primitive is formed of two parts, `flip` and `mirror`. It is in the `flip` part that we *encode* the equivalent to the classical subroutine that checks the validity of a solution, while the `mirror` part remains the same for all applications. In Chapter 14 we will cover this aspect of AA fully and learn how to encode classical subroutines in the `flip` part of AA.

Inside the QPU

So how do the QPU operations making up each AA iteration allow it to undertake its task? Rather than worry about the functioning of every individual operation, we'll try to build an intuitive understanding of an AA iteration's effect. It turns out there's a useful way of understanding amplitude amplification in terms of its geometrical effect on a QPU register's circle notation.

The Intuition

There are two stages to amplitude amplification: `flip` and `mirror`. The `flip` subroutine flips the phase of a marked term in the superposition that we ultimately want to extract from our QPU.

The `mirror` subroutine, which remains unchanged throughout the AA primitive, turns phase differences into contrasts in magnitude. But another way to understand `mirror` is that it causes each value in a state to *mirror about the average* of all values.

As well as explaining why we call this subroutine `mirror`, this alternative interpretation also helps us give a more precise step-by-step account of what `mirror` achieves.

Suppose we have a two-qubit input state to the `mirror` subroutine, which is in a superposition of $|0\rangle$ and $|3\rangle$ as shown in Figure 6-20.

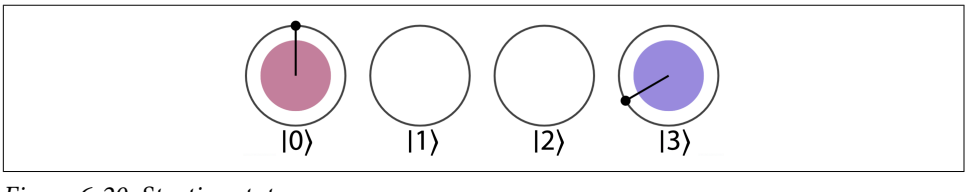


Figure 6-20. Starting state

In terms of circle notation, the `mirror` subroutine performs the following steps:

1. Find the *average* of all the values (circles). This can be done by numerically averaging the x and y positions of the points within the circles.[2] When calculating the average, the zero values (empty circles) should be included as `[0.0, 0.0]`, as shown in Figures 6-21 and 6-22.

2 In terms of the full-blown mathematical description of a QPU register state as a complex vector, this corresponds to averaging the real and imaginary parts of its components.

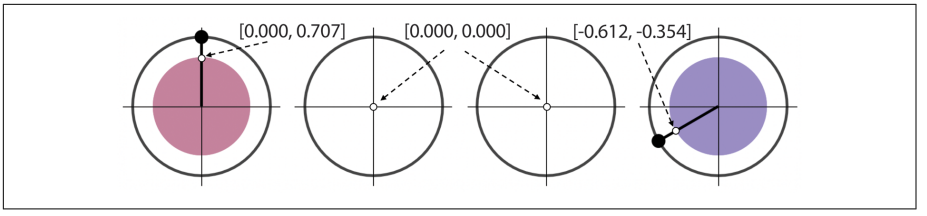


Figure 6-21. Calculating the average

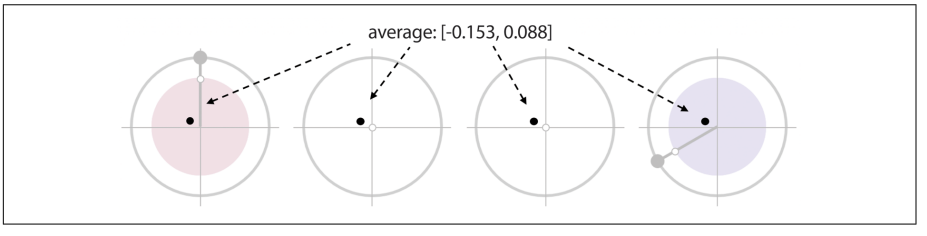


Figure 6-22. Plotting the average

2. Flip each value about the common average. Visually, this is simply a reflection, as shown in Figure 6-23.

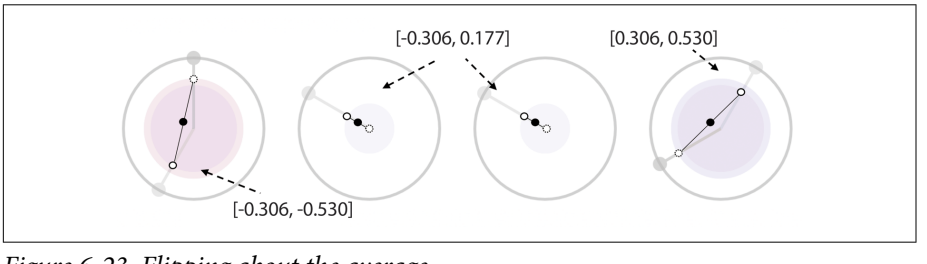


Figure 6-23. Flipping about the average

That's all there is to it. The result, as shown in Figure 6-24, is that the phase differences in our original state have been converted into magnitude differences. It's worth noting that the common average of the circles is still the same. This means that applying the transform again will simply return the initial state.

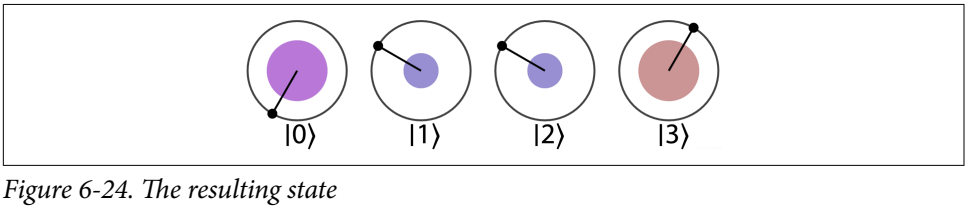


Figure 6-24. The resulting state

How is it that this "mirror about the average" operation ends up converting phase and magnitude differences? Imagine that there are many states with similar phases, but one oddball state with a very different phase, as shown in Figure 6-25.

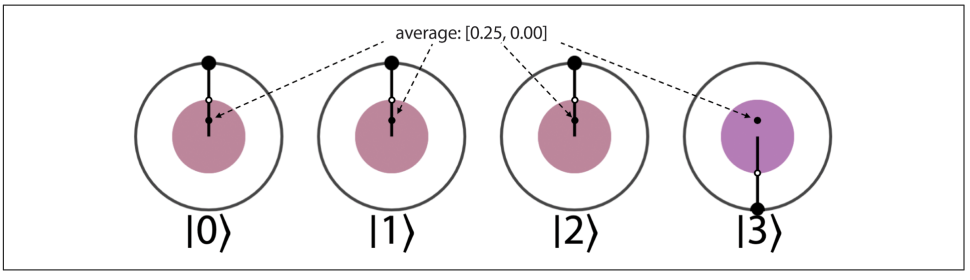


Figure 6-25. State with one oddball phase

Given that most values are the same, the average will lie closer to the value of most of the states, and very far from the state that has the opposite phase. This means that when we mirror about the average, the value with the different phase will "slingshot" over the average and stand out from the rest, as shown in Figure 6-26.

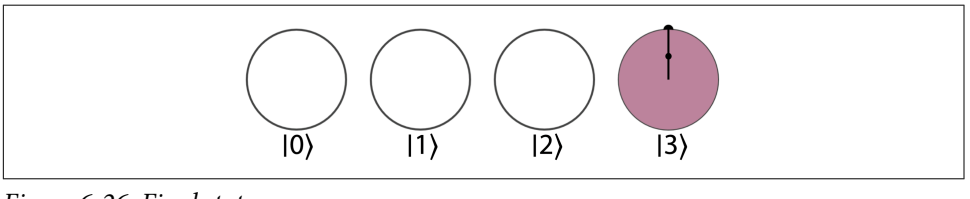


Figure 6-26. Final state

For most applications of AA that we discuss in Chapter 10, the replacements for `flip` will introduce a phase difference of 180° between the marked and unmarked states, so the preceding "slingshotting" example is particularly relevant.

In practice, it's actually simpler to implement *mirror about the average + flip all phases* rather than simply *mirror about the average*. Since only the *relative* phases in a register are actually important, this is entirely equivalent.

Conclusion

This chapter introduced one of the core operations in many QPU applications. By converting phase differences into magnitude differences, amplitude amplification allows a QPU program to provide useful output relating to phase information from a state that would otherwise remain invisible. We explore the full power of this primitive in Chapters 11 and 14.

CHAPTER 7

QFT: Quantum Fourier Transform

The *Quantum Fourier Transform* (QFT) is a primitive allowing us to access hidden patterns and information stored inside a QPU register's relative phases and magnitudes. While amplitude amplification allowed us to turn relative-phase differences into `READ`able differences in magnitudes, we'll see that the QFT primitive has its own distinct way of manipulating phases. In addition to performing *phase manipulation*, we'll also see that the QFT primitive can help us *compute in superposition* by easily preparing complex superpositions of a register. This chapter begins with some straightforward QFT examples, and then dives into subtler aspects of the tool. For the curious, "Inside the QPU" on page 146 will examine the QFT operation by operation.

Hidden Patterns

Let's make our state guessing game from Chapter 6 a little harder. Suppose we have a four-qubit quantum register containing one of the three states (A, B, or C) shown in Figure 7-1, but we don't know which one.

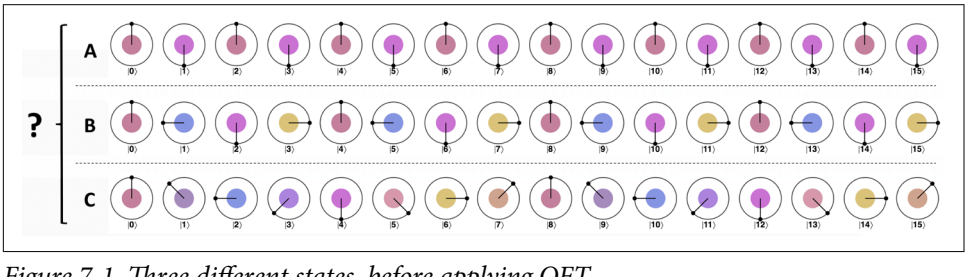


Figure 7-1. Three different states, before applying QFT

Note that these are *not* the same A, B, and C states that we discussed in the previous chapter.

Visually we can tell that these states are different from each other, but since the magnitudes of all values in each of these states are the same, reading the register returns an evenly distributed random value, regardless of which state it was actually in.

Even amplitude amplification isn't much help to us here, since no single phase stands out as being different in each state. But the QFT primitive comes to the rescue! (Cue dramatic music.) Applying the QFT to our register before readout would transform each of the states A, B, and C into the results shown in Figure 7-2.

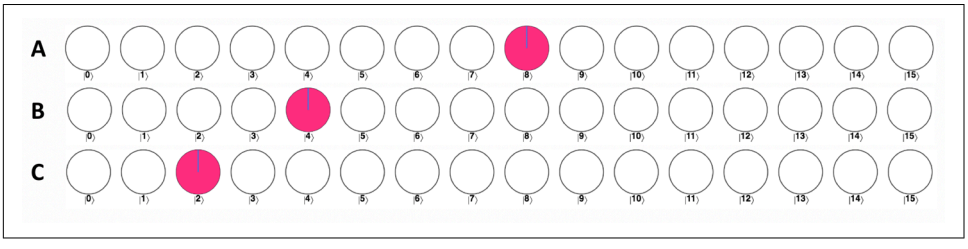


Figure 7-2. The same three states, after applying QFT

Reading the register now lets us immediately and unambiguously determine which state we started with, using only a single set of `READ`s. These QFT results have an intriguing relationship to the input states from Figure 7-1. In state A, the relative phase of the input state returns to 0 eight times, and the QFT allows us to `READ` the value `8`. In state B the relative phase rotates back to its starting value four times, and the QFT allows us to `READ 4`. State C continues the trend. In each case, the QFT has actually revealed a *signal frequency* contained in our QPU register.

The sample code in Example 7-1 allows you to generate the states A, B, and C to experiment with yourself. By altering the value of `which_signal` you can select which state to prepare.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-1*.

Example 7-1. Using QFT to distinguish between three states

```
var num_qubits = 4;
qc.reset(num_qubits);
var signal = qint.new(num_qubits, 'signal')
var which_signal = 'A';

// prepare the signal
signal.write(0);
signal.hadamard();
if (which_signal == 'A') {
    signal.phase(180, 1);
```

```
} else if (which_signal == 'B') {
    signal.phase(90, 1);
    signal.phase(180, 2);
} else if (which_signal == 'C') {
    signal.phase(45, 1);
    signal.phase(90, 2);
    signal.phase(180, 4);
}

signal.QFT()
```

If you want to apply the QFT primitive to the states produced by Example 7-1 you can do so in QCEngine using the built-in `QFT()` function. This can either be implemented using the global method `qc.QFT()`, which takes a set of qubits as an argument, or as a method of a `qint` object—`qint.QFT()`—which performs the QFT on all qubits in the `qint`.

The QFT, DFT, and FFT

Understanding the QFT's ability to reveal signal frequencies is best achieved by examining its very strong similarity to a classic signal-processing mechanism called the *Discrete Fourier Transform* (DFT). If you've ever fiddled with the graphical equalizer on a sound system, you'll be familiar with the idea of the DFT—like the QFT, it allows us to inspect the different *frequencies* contained within a signal. While the DFT is used to inspect more conventional signals, under the hood the transformation it applies is essentially identical to the QFT's mathematical machinery. It's easier to build intuition with the more tangible DFT, so we'll familiarize ourselves with its core concepts as we move through the chapter.

A widely used fast implementation of the DFT is the *Fast Fourier Transform* (FFT). The FFT is the fastest known method for determining Discrete Fourier Transforms, so we'll generally use the FFT as the point of comparison when talking about our quantum variant. The FFT also conveniently shares a similarity with the QFT in that it is restricted to power-of-two signal lengths.

Since there'll be a lot of *-FT* acronyms floating around, here's a quick glossary:

DFT
: The conventional *Discrete Fourier Transform*. Allows us to extract frequency information from conventional signals.

FFT

The *Fast Fourier Transform*. A specific algorithm for implementing the DFT. Performs precisely the same transformation as the DFT, but is significantly faster in practice. The one to beat for our QFT.

QFT

The *Quantum Fourier Transform*. Performs the same transformation as the DFT, only it operates on signals encoded *in quantum registers* rather than conventional streams of information.

Frequencies in a QPU Register

Thinking of the magnitudes and relative phases in quantum registers as signals is a very powerful idea, but one that warrants some clarification. Given the idiosyncrasies of the QPU, let's be a little more explicit about what it means for a quantum register to contain frequencies at all. Suppose that we have the four-qubit register state shown in Figure 7-3.

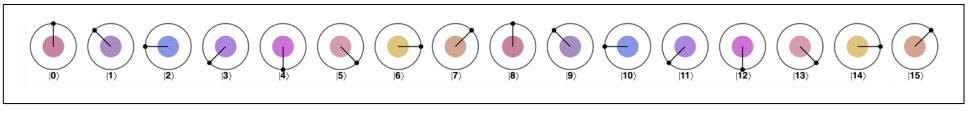


Figure 7-3. Example QFT input state

This is state C from our previous example. Notice that if we look along the register from left to right, the relative phases of the $2^4 = 16$ values go through two full anticlockwise rotations. Thus, we can think of the relative phases in our register as representing a *signal* that repeats with a frequency of *twice per register*.

That gives us an idea of how a quantum register might encode a signal having some frequency associated with it—but since that signal is tied up in the relative phases of our qubits, we know only too well that we won't be able to extract it by simply reading the register. As with the example at the beginning of this chapter, the information is *hidden* in the phases, but can be revealed by applying the QFT (see Figure 7-4).

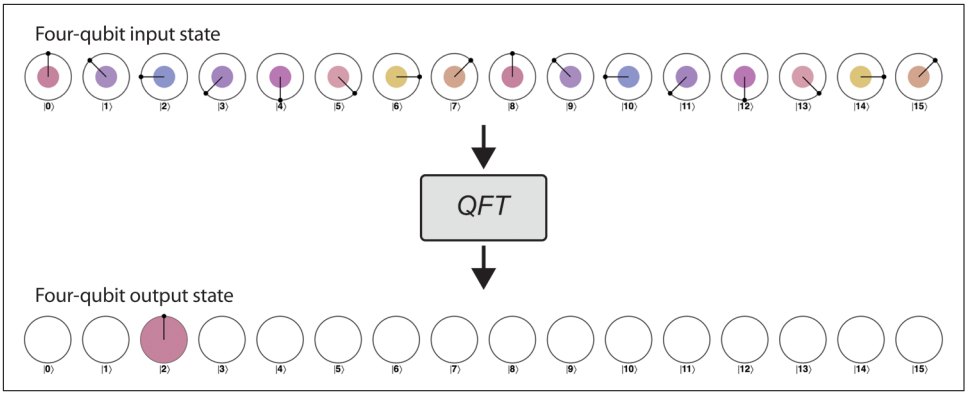


Figure 7-4. QFT example 2

In this simple case, reading out our register after the QFT allows us to determine that the frequency of repetition it contained was 2 (i.e., the relative phase makes two rotations per register). This is the key idea of the QFT, and we'll learn next to more accurately use and interpret it. To reproduce this, see Example 7-2, shown in Figure 7-5.

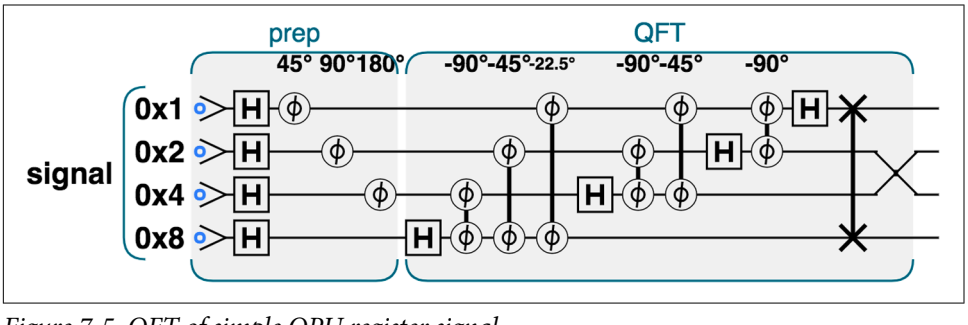


Figure 7-5. QFT of simple QPU register signal

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-2*.

Example 7-2. QFT of simple QPU register signal

```
var num_qubits = 4;
qc.reset(num_qubits);
var signal = qint.new(num_qubits, 'signal')

// prepare the signal
signal.write(0);
signal.hadamard();
signal.phase(45, 1);
signal.phase(90, 2);
```

```
signal.phase(180, 4);

// Run the QFT
signal.QFT()
```

Example 7-2 demonstrates an important property of the QFT. Although we saw in Chapter 5 that operations implemented on a QPU often need to use distinct input and output registers (for reversibility), the QFT's output occurs on the same register that provided its input—hence the suffix *transform*. The QFT can operate *in-place* because it is an inherently reversible circuit.

You may be wondering why we care about a tool that helps us find frequencies in a QPU register. How can this help us solve practical problems? Surprisingly, the QFT turns out to be a versatile tool in QPU programming, and we'll give some specific examples toward the end of the chapter.

So, is that all there is to the QFT? Pop in a register, get back its frequency? Well, yes and no. So far the signals we've looked at have, after applying the QFT, given us a single well-defined frequency (because we cherry-picked them). But many other periodic signals don't give us such nice QFTs. For example, consider the four signals shown in Figure 7-6.

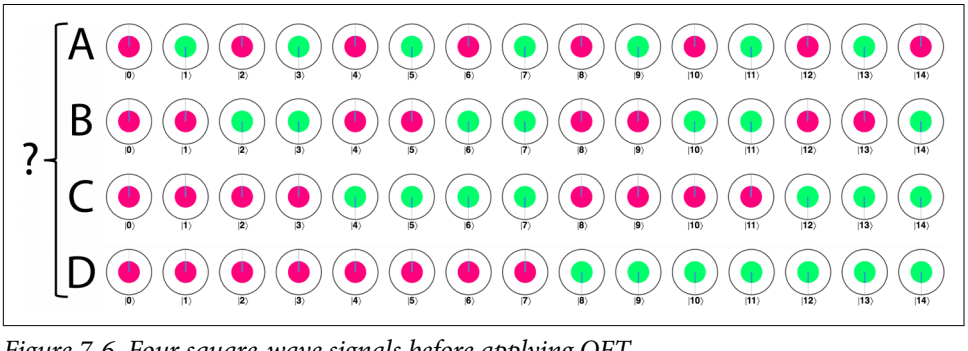


Figure 7-6. Four square-wave signals before applying QFT

A value in a QPU register's state having a phase of 180° (shown in circle notation as a south-pointing line) is equivalent to its amplitude having a negative value.

These are the square-wave equivalents of the examples we first considered in Figure 7-1. Though oscillating at the same frequency as these previous examples, their relative phases abruptly flip-flop between two positive and negative values,

rather than varying continuously. Consequently, their QFTs are a little harder to interpret, as you can see in Figure 7-7.

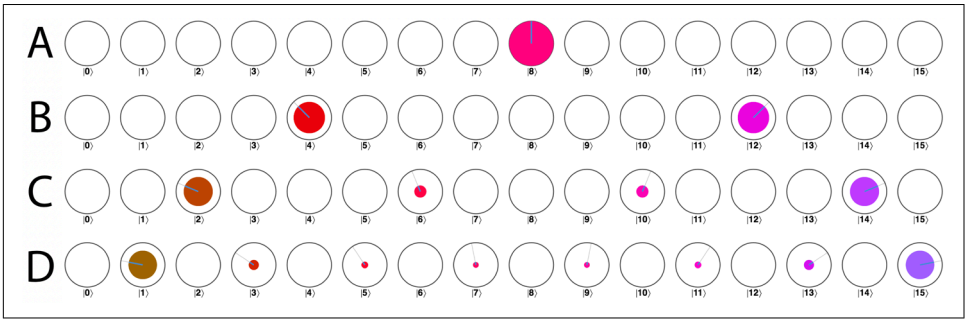


Figure 7-7. The same square-wave signals, after applying QFT

Note that we can produce the square-wave input states shown in these examples using `HAD` and carefully chosen `PHASE` QPU operations. The sample code in Example 7-3, shown in Figure 7-8, generates these square-wave states and then applies the QFT. Changing the `wave_period` variable allows us to select which state from Figure 7-6 we want to produce.

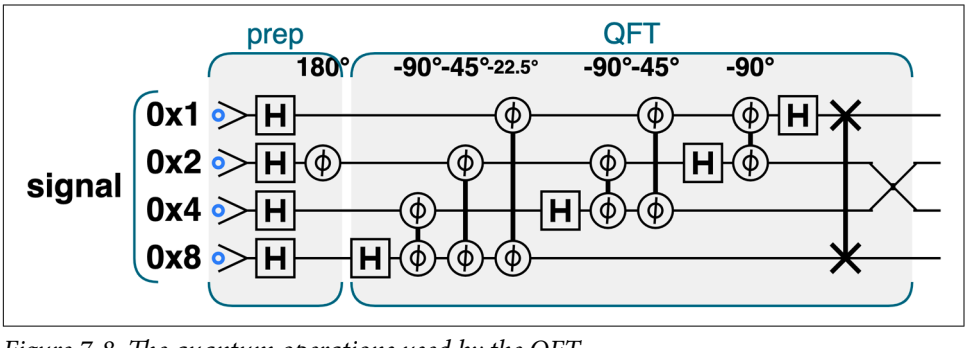


Figure 7-8. The quantum operations used by the QFT

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-3*.

Example 7-3. Square-wave QFT

```
var num_qubits = 4;
qc.reset(num_qubits);
var signal = qint.new(num_qubits, 'signal')
var wave_period = 2; // A:1 B:2 C:4 D:8

// prepare the signal
signal.write(0);
```

```
signal.hadamard();
signal.phase(180, wave_period);

signal.QFT()
```

If you're familiar with the conventional DFT, the results in Figure 7-7 may not be quite so perplexing. Since the QFT really just applies the DFT to QPU register signals, we'll begin by recapping this more conventional cousin.

The DFT

The DFT acts on discrete samples taken from a signal, whether it be a musical waveform or a digital representation of an image. Although conventional signals are normally thought of as lists of more tangible real values, the DFT will also work on complex signal values. This is especially reassuring for us, since (although we try our best to avoid it) the full representation the state of a QPU register is most generally described by a list of complex numbers.

We've used circle notation to intuitively visualize the mathematics of complex numbers wherever we can. Whenever you see "complex number," you can always think of a single circle from our notation, with a magnitude (size of circle) and phase (rotation of circle).

Let's try out the conventional DFT on a simple sine-wave signal, one for which the samples are real numbers and there is one well-defined frequency (if we were dealing with a sound signal, this would be a pure tone). Furthermore, let's suppose that we have taken 256 samples of the signal. Each sample is stored as a complex floating-point number (with magnitude and phase, just like our circle notation). In our particular example the imaginary part will be zero for all samples. Such an example signal is shown in Figure 7-9.

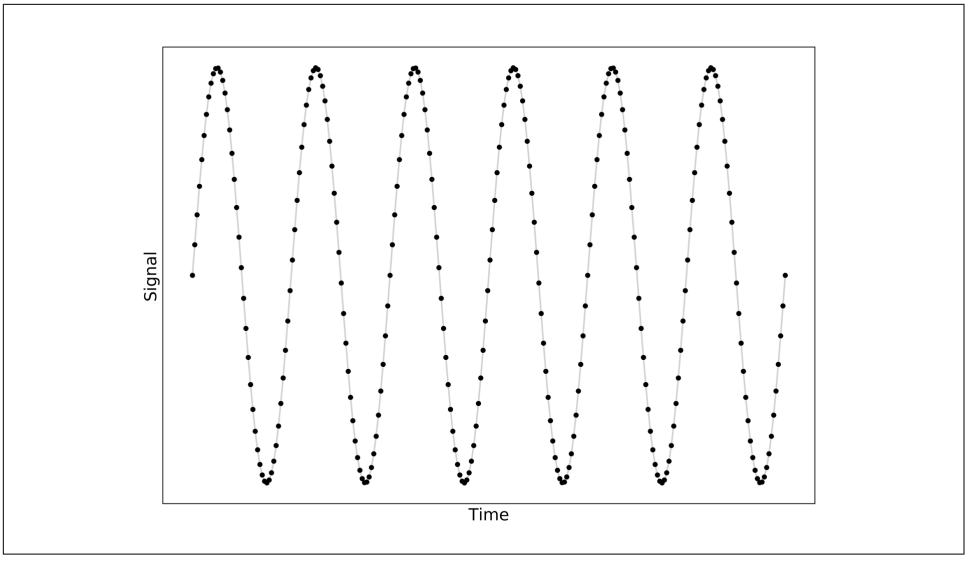


Figure 7-9. Sampling 256 points from a simple sine wave

If each sample is stored as a 16-byte complex float (8 bytes for real, and 8 for imaginary), then this signal will fit in a 4,096-byte buffer. Note that we still need to keep track of the imaginary parts of our samples (even though in this example we used only real input values), since the *output* of the DFT will be 256 complex values.

The *magnitudes* of the complex output values from the DFT tell us how significantly a given frequency contributes to making up our signal. The *phase* parts tell us how much these different frequencies are *offset* from each other in the input signal. Figure 7-10 shows what the magnitudes of the DFT look like for our simple real sine-wave signal.

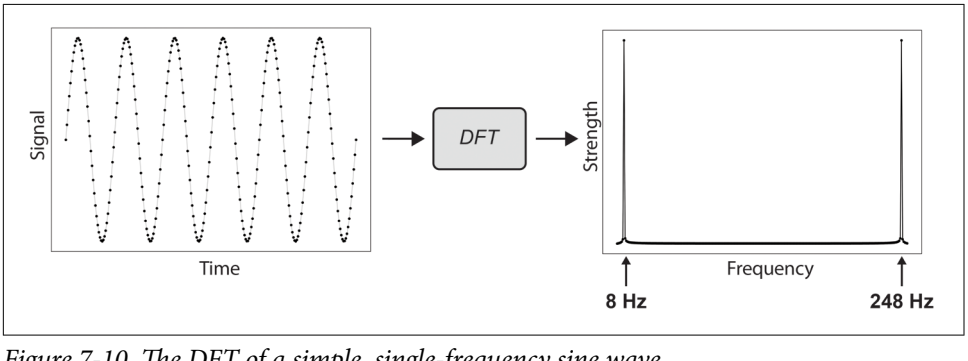


Figure 7-10. The DFT of a simple, single-frequency sine wave

Looking carefully around the base of the two peaks in Figure 7-10, you'll notice that they are surrounded by small nonzero values. Since the DFT output is limited to a finite number of bits, we can see peaks of nonzero width, even in cases where a signal truly contains only a single frequency. It's worth being aware that the same can be true for the QFT.

The DFT has transformed the signal into *frequency space*, where we can see all the frequency components present inside a signal. Since the input signal completes eight full oscillations within the sample time (1s), we expect it to have a frequency of 8 Hz, and this is precisely what the DFT returns to us in the output register.

Real and Complex DFT Inputs

Looking at our example DFT output, you may notice the 248 Hz elephant in the room. Alongside the expected 8 Hz frequency, the DFT also produces a conspicuous second *mirror-image* peak in frequency space.

This is a property of the DFT of any real signal (one where the samples are all real numbers, as is the case for most conventional signals). In such cases, only the first half of the DFT result is actually useful. So in our example, we should only take note of the first 256/2 = 128 points that the DFT returns. Everything else in the DFT after that point will be the mirror image of the first half (which is why we see the second peak symmetrically at 256 – 8 = 248 Hz). Figure 7-11 shows a few more examples of the DFTs of real signals, further highlighting this symmetry.

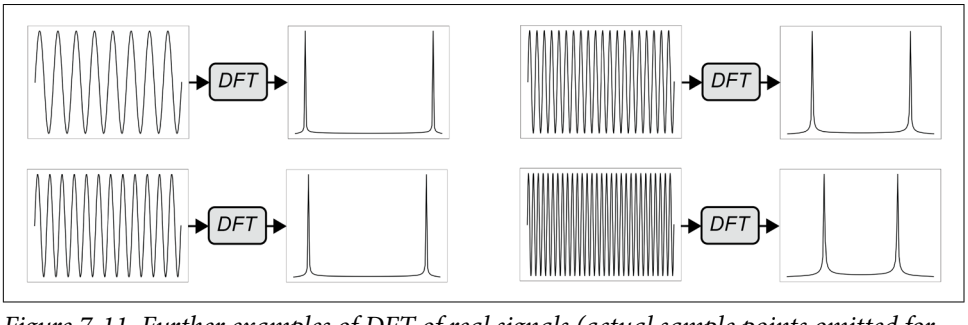


Figure 7-11. Further examples of DFT of real signals (actual sample points omitted for clarity)

Had we used a *complex* input signal, we wouldn't see this symmetry effect, and each of the 256 data points in the DFT's output would contain distinct information.

The same caveat on interpreting the output of real versus complex input signals also holds true for the QFT. Looking back at the QFT examples we first gave in Figures 7-2 and 7-4, you'll notice that there was no such symmetry effect—we only observed

a single "peak" in the output registers corresponding to precisely the frequency of the input. This is because the signal, being encoded in the relative phases of our input register's states, was complex, so no symmetrical redundancy was seen.

It is nevertheless possible to prepare entirely real signals in our QFT register. Suppose that the signal we planned to QFT was encoded in the magnitudes of our input QPU register, instead of the relative phases, as shown in Figure 7-12.[1]

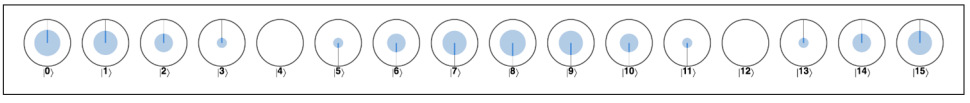


Figure 7-12. Quantum register with signal encoded in terms of magnitudes

This input signal is entirely real. Don't be fooled by the fact that the relative phases change across the state—recall that a phase of 180° is the same as a negative sign, so all we ever have are real positive and negative numbers.

Applying the QFT to this *real* input state we find that the output register exhibits precisely the mirror-image effect that we saw with the conventional DFT (see Figure 7-13).

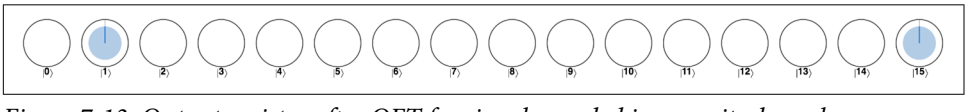


Figure 7-13. Output register after QFT for signal encoded in magnitudes only

Consequently, we need to take care of how we interpret QFT results dependent on whether our input register is encoding information in phases or magnitudes.

DFT Everything

So far we've vaguely talked about the DFT (and QFT) showing us what frequencies a signal contains. To be slightly more specific, these transforms are actually telling us the frequencies, proportions, and offsets of simple *sinusoidal* components that we could combine to produce the input signal.

Most of the example input signals we considered have actually been simple sinusoids themselves, with single well-defined frequencies. As a consequence, our DFTs and QFTs have yielded single well-defined peaks in frequency space (effectively saying, "You can build that from just one sine function having a certain frequency!"). One of the most useful things about the DFT is that we can apply it to much more complex signals that aren't obviously sinusoidal. Figure 7-14 shows the magnitudes from a

1 We will shortly show a QPU circuit that allows us to produce a register having such magnitude-encoded oscillatory signals.

conventional DFT performed on an input signal containing sinusoidal oscillations at three different frequencies.

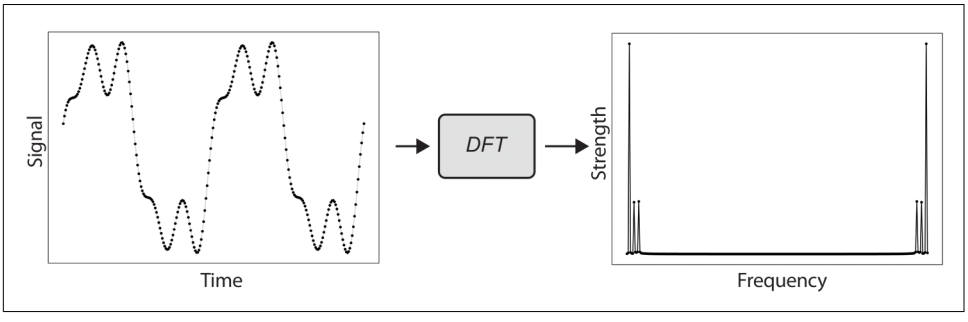


Figure 7-14. DFT on a signal consisting of a mixture of frequencies

It's always worth taking note of the value of a DFT (or QFT) corresponding to *zero* frequency. This is sometimes known as the *DC bias*, and it reveals the baseline value that the signal oscillates above and below. Since our examples have all oscillated about zero, they've had no DC bias component in their DFTs.

Not only does this DFT show us that three sinusoidal frequencies make up our signal, but the relative heights of the peaks in frequency space also tell us how significantly each frequency contributes to the signal (by inspecting the phase of the fully complex values returned by the DFT we could also learn how the sinusoids should be offset from each other). Don't forget, of course, that since our input signal here is real, we can ignore the second (mirror-image) half of the DFT.

One particularly common and useful class of input signals that don't look at all sinusoidal are square waves. We actually saw the QFT of a few square waves already, in Figure 7-7. In Figure 7-15 we show what the magnitudes look like for the conventional DFT of a square wave.

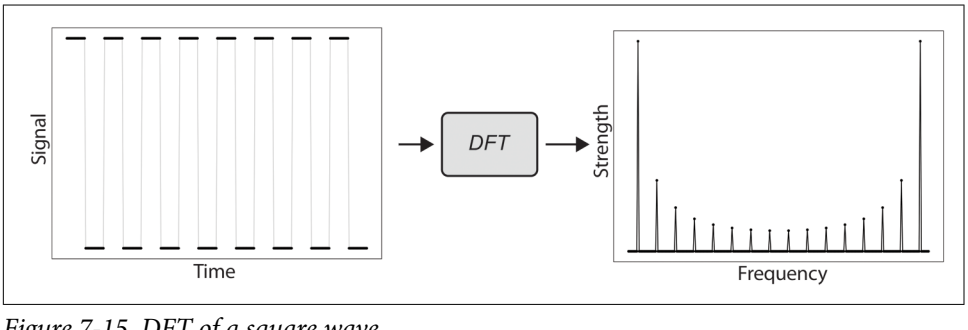


Figure 7-15. DFT of a square wave

Although Figure 7-7 already showed us the QFTs of a set of square waves, let's walk through an example more carefully. To make things a bit more interesting, we consider an eight-qubit register (i.e., a qubyte), giving us 256 state values to play with. The sample code in Example 7-4, shown in Figure 7-16, prepares a square wave repeating eight times—the QPU register equivalent to the signal shown in Figure 7-15.

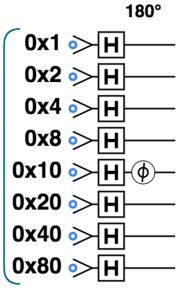


Figure 7-16. The quantum gates needed to prepare an eight-qubit square-wave signal

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-4*.

Example 7-4. Square-wave QFT circuit

```
// Setup
qc.reset(8);

// Create equal superposition
qc.write(0);
qc.had();

// Introduce a negative sign with a certain frequency
// (By placing the phase on different qubits we can alter
// the frequency of the square wave)
qc.phase(180, 16);

// Apply the QFT
qc.QFT();
```

The code just before the QFT leaves our QPU input register in the state shown in Figure 7-17.

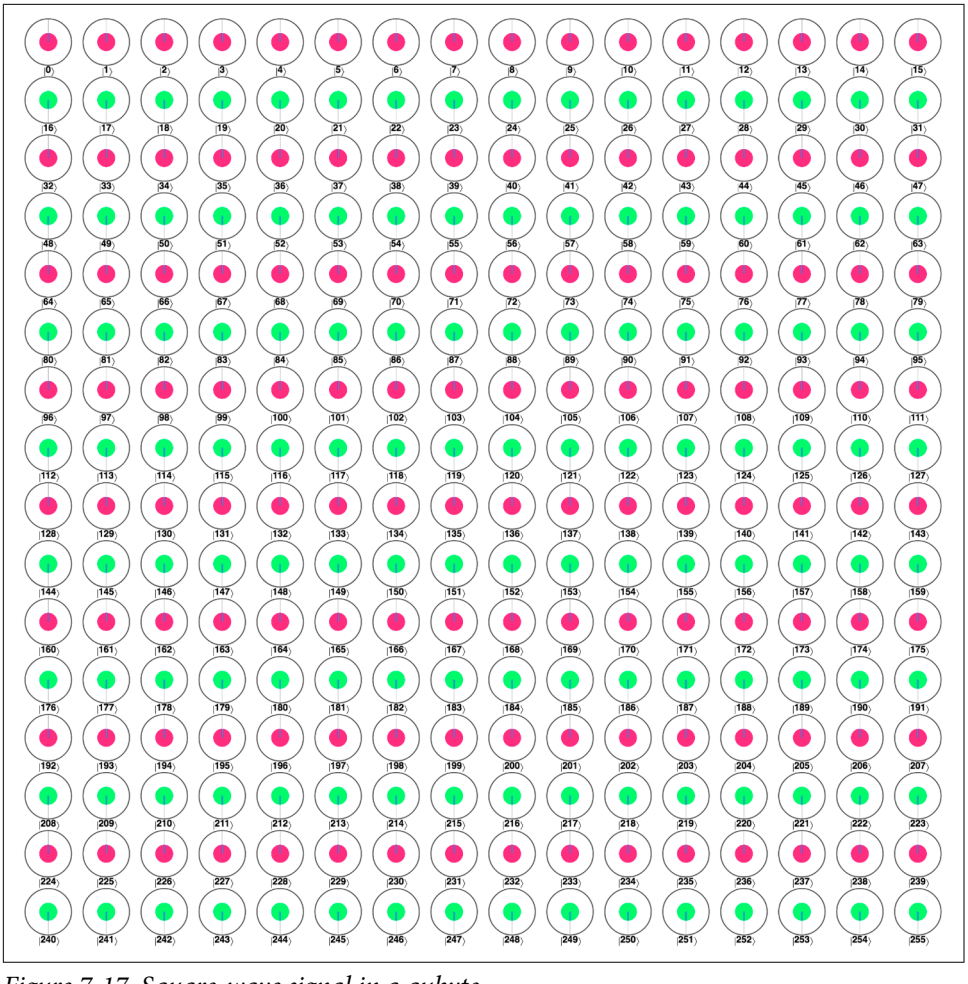


Figure 7-17. Square-wave signal in a qubyte

Recalling that the green circles (with a relative phase of 180°) represent negative signs, you should hopefully be able to convince yourself that this signal is both entirely real and completely analogous to the DFT input signal shown in Figure 7-15.

Applying the QFT to a register in this state (by calling `qc.QFT()` in QCEngine), we obtain the output state shown in Figure 7-18.

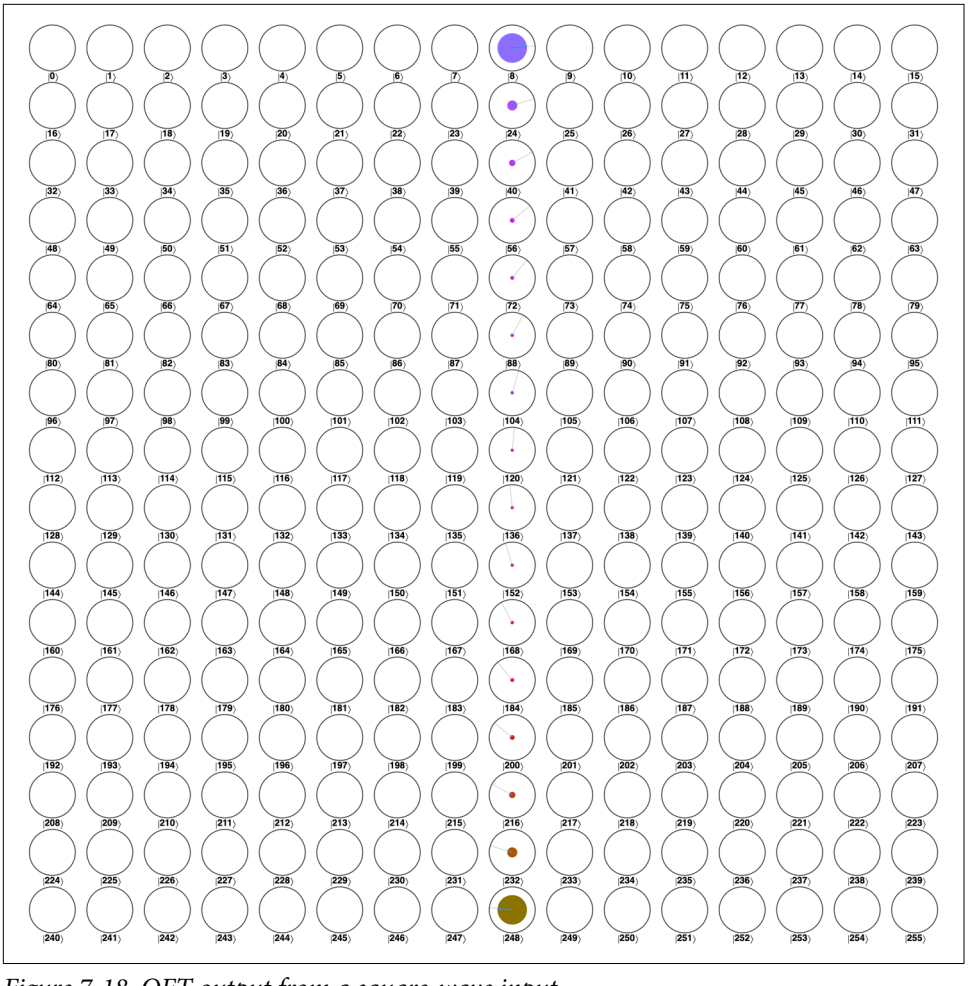


Figure 7-18. QFT output from a square-wave input

This register is precisely the square-wave DFT from Figure 7-15, with each component in frequency space being encoded in the magnitudes and relative phases of the states of our QPU register. This means that the probability of reading out a given configuration of the post-QFT register is now determined by how strongly a given frequency contributes to our signal.

Plotting these probabilities in Figure 7-19, we can see the similarity to the conventional DFT results previously obtained for a square wave in Figure 7-15.

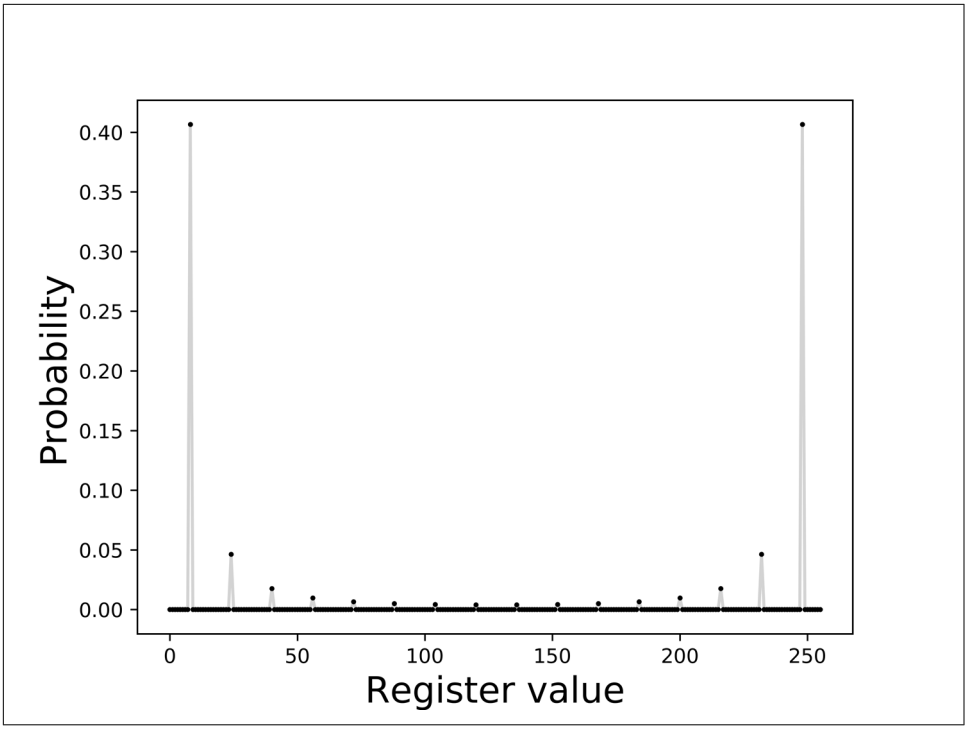


Figure 7-19. Readout probabilities for the QFT of a square-wave qubyte input

Using the QFT

We've gone to some lengths to show that we can think of the QFT as faithfully implementing the DFT on QPU register signals. Given how finicky and expensive a QPU is, this sounds like a lot of trouble just to reproduce a conventional algorithm. It's important to remember that our motivation for using the QFT primitive is primarily to manipulate phase-encoded computation results, rather than to provide a full-blown FFT replacement.

That said, we'd be remiss if we didn't evaluate how the computational complexity of the QFT compares to the FFT—especially since the QFT turns out to be far, *far* faster than the best conventional counterpart.

The QFT Is Fast

When talking about the speed of FFT and QFT algorithms, the figure we're interested in is how the runtime of the algorithm increases as we increase the size of our input signal (in terms of the total number of bits needed to represent it). For all practical purposes, we can think of the number of fundamental operations that an algorithm

uses as being equivalent to the time it takes to run, since each operation takes a fixed time to act on a QPU's qubits.

The FFT requires a number of operations that grows with the number of input bits n as $O(n2^n)$. The QFT, however—by leveraging the 2^m states available in an m-qubit register—uses a number of gates that grows only as $O(m^2)$.

Pragmatically, this means that for small signals (less than 22 bits, or about 4 million samples) the conventional FFT will be faster, even using just a laptop. But as the size of the input problem grows, the exponential advantage of the QFT becomes clear. Figure 7-20 shows how the two methods compare as the signal size increases. Note that the x-axis refers to the number of bits (in the case of the FFT) or qubits (for the QFT) making up the input register.

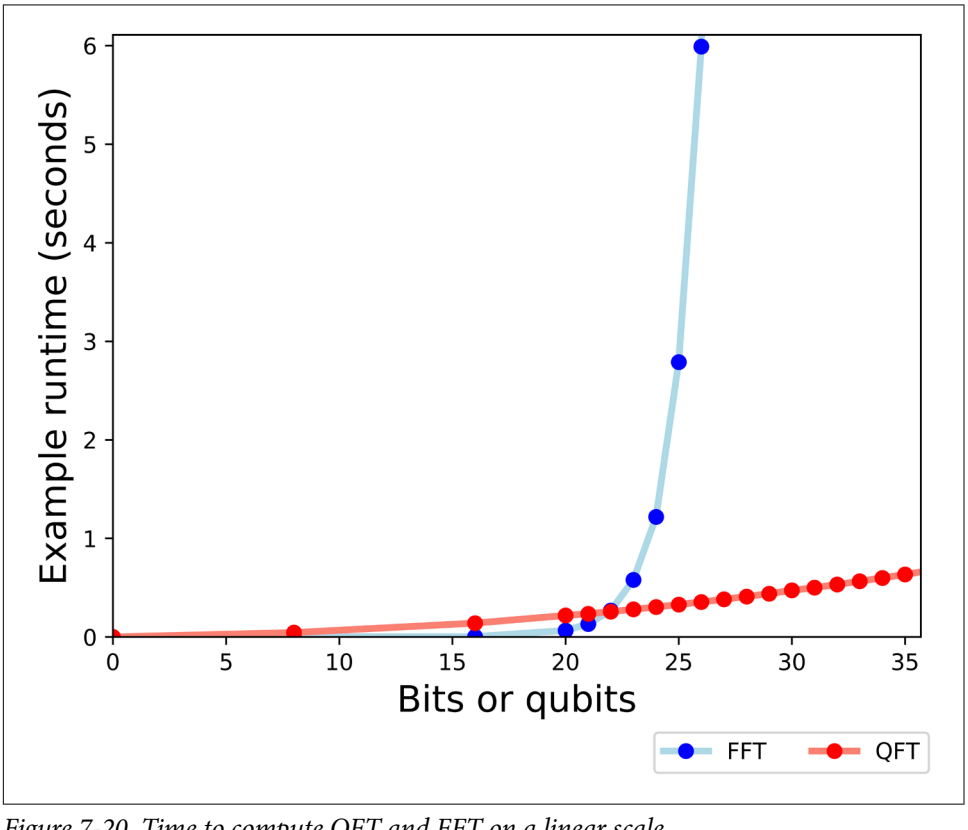


Figure 7-20. Time to compute QFT and FFT on a linear scale

Signal processing with the QFT

Given the importance of the FFT to the wide-ranging field of signal processing, it's very tempting to think that the main use of the QFT is surely to provide an exponen-

tially faster implementation of this signal-processing tool. But whereas the output of the DFT is an array of digital values that we can study at our leisure, the output of the QFT is locked in our QPU's output register.

Let's be a little more methodical in stating the limitations we face anytime we try to make practical use of output from the QFT. Having our input signal and its QFT output encoded in the amplitudes of a quantum state introduces two challenges:

1. How do we get our signal into an input quantum register in the first place?
2. How do we access the result of the QFT when we're done?

The first problem—getting an input signal into a quantum register—is not always easy to solve. The square-wave example we studied in Figure 7-18 could be generated with a fairly simple quantum program, but if we intend to input arbitrary conventional data into a quantum register, such a simple circuit may not exist. For some signals, the cost of initializing the input register may wipe out the QFT's benefit.

There are techniques for preparing certain kinds of superpositions in a QPU register more easily, although often these require additions to standard QPU hardware. In Chapter 9 we present one such technique.

The second problem we listed is, of course, the fundamental challenge of QPU programming—how do we `READ` out an answer from our register? For simple single-frequency QFTs like those we saw in Figure 7-2, reading out the register gives an unambiguous answer. But the QFT of more complex signals (such as those in Figure 7-18) produces a superposition of frequency values. When `READ`ing this out we only randomly obtain one of the present frequencies.

However, the final state after a QFT can still be useful to us under certain circumstances. Consider the output from the square-wave signal input in Figure 7-17:

1. If an application (the one calling the QFT) is okay with randomly getting an answer that is "the dominant frequency or a multiple of it," we're all set. There are eight correct answers in our square-wave example, and we'll always get one of them.
2. If an application can verify a desired answer, we can quickly detect whether a random outcome from a `READ`ing the QFT output state is suitable. If not, we can run the QFT again.

In these ways, the signal-processing capabilities of the QFT can still offer a valuable phase manipulation primitive; in fact, we'll see it play just such a role in Shor's factoring algorithm in Chapter 12.

Preparing superpositions with the inverse QFT

So far we've thought of the QFT as a *phase manipulation* primitive. Another use of the QFT is to prepare (or alter) periodically varying superpositions in ways that would otherwise be very difficult. Like all non-`READ` QPU operations, the QFT has an *inverse*. The inverse QFT (invQFT) takes as its input a qubit register representing frequency space, and returns as output a register showing the signal that this corresponds to.

We can use the invQFT to easily prepare periodically varying register superpositions as follows:

1. Prepare a quantum register representing the state you need in frequency space. This is often easier than preparing the state directly with a more complicated circuit.
2. Apply the invQFT to return the required signal in the QPU output's register.

Example 7-5, shown in Figure 7-21, shows how we can create a qubyte register with a relative phase oscillating three times across the register.

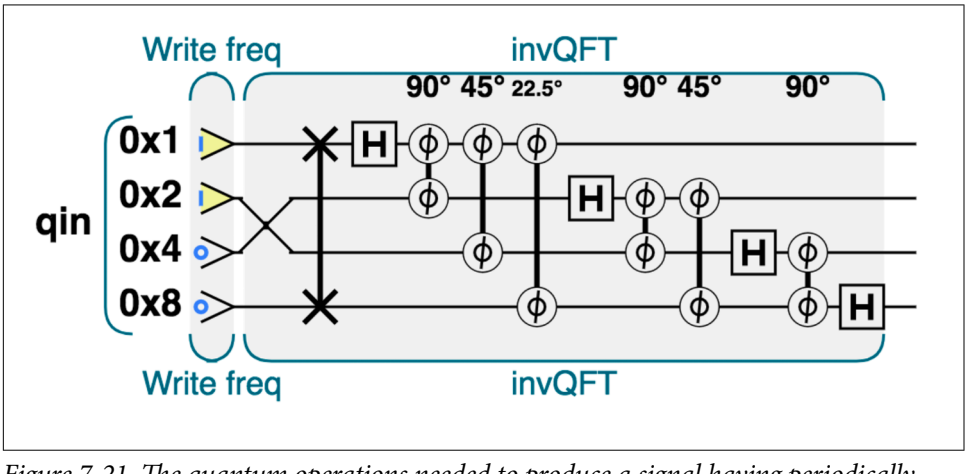


Figure 7-21. The quantum operations needed to produce a signal having periodically varying phase

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-5*.

Example 7-5. Converting frequency into state

```
// Setup
var num_qubits = 4;
qc.reset(num_qubits);
```

```
var qin = qint.new(num_qubits, 'qin');

// Write the frequency we want to register
qin.write(3);

// Inverse QFT to turn into a signal
qin.invQFT()
```

The invQFT can also be used to prepare a QPU register that varies periodically in magnitude rather than relative phase, like the example we saw in Figure 7-12. To do this we need only recall that a register with periodically varying magnitudes is a real signal, and so in frequency space we need a symmetric representation for the invQFT to act on. This is demonstrated in Example 7-6, and shown in Figure 7-22.

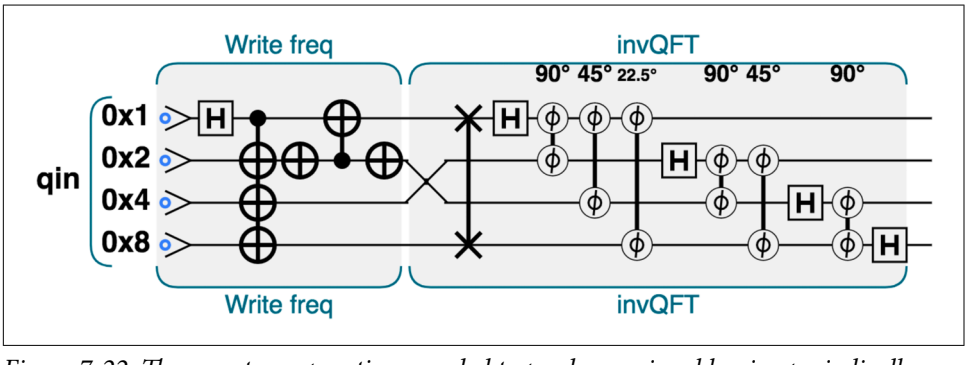


Figure 7-22. The quantum operations needed to produce a signal having periodically varying magnitude

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-6*.

Example 7-6. Prepare a state with invQFT

```
// Setup
var num_qubits = 4;
qc.reset(num_qubits);
var qin = qint.new(num_qubits, 'qin');
qin.write(0);

// Write the frequencies we want to register
qin.had(1);
qc.cnot(14,1);
qin.not(2);
qc.cnot(1,2);
qin.not(2);
```

```
//Inverse QFT to turn into a signal
qin.invQFT()
```

As well as preparing states with given frequencies, the invQFT also allows us to easily *alter* their frequency information. Suppose that at some point during an algorithm we want to increase the frequency with which the relative phases oscillate in our QPU register. We can take the following steps:

1. Apply the QFT to the register, so that we now have the signal represented in frequency space.
2. Add 1 to the value now stored in the register. Since our input signal is complex, this will increase the value of each frequency component.
3. Apply the invQFT to get back our original signal, only now with increased frequencies.

The sample code in Example 7-7, shown in Figure 7-23, provides a simple example of doing this.

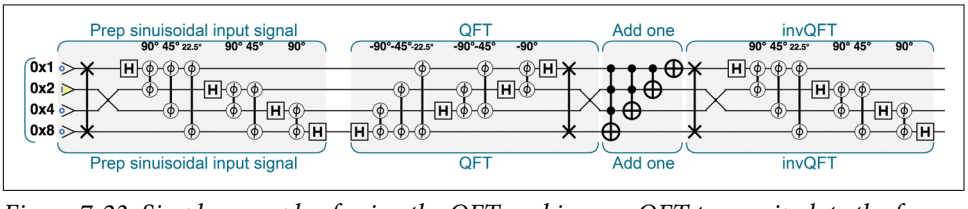


Figure 7-23. Simple example of using the QFT and inverse QFT to manipulate the frequency of a signal

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-7*.

Example 7-7. QFT frequency manipulation

```
// Set up input register
var n = 4;

// Prepare a complex sinusoidal signal
qc.reset(n);
var freq = 2;
qc.write(freq);
var signal = qint.new(n, 'signal');
signal.invQFT();

// Move to frequency space with QFT
signal.QFT();
```

```
// Increase the frequency of signal
signal.add(1)

// Move back from frequency space
signal.invQFT();
```

Moving to frequency space can allow us to perform otherwise tricky manipulations on a state, but there are cases where we need to take care. For example, *real* input signals are harder to manipulate.

If the QPU register contains a state representing a single frequency, or a superposition of frequencies that is not symmetric, after the application of the invQFT we will have a QPU register that varies periodically in *relative phase*. If, on the other hand, prior to the application of the invQFT, the register contains a symmetric superposition of frequencies, the output QPU register will contain a state that varies periodically in *magnitude*.

Inside the QPU

Figure 7-24 shows the fundamental QPU operations used for performing the QFT on a qubyte signal.

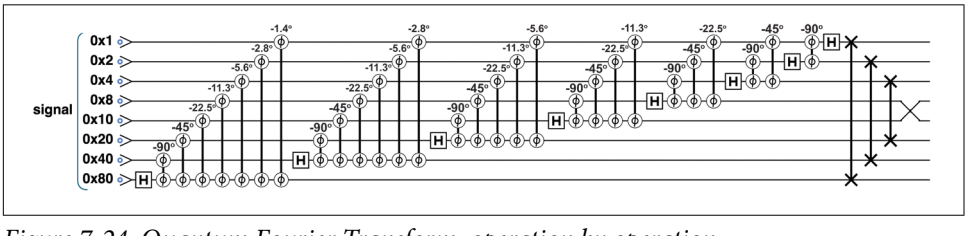


Figure 7-24. Quantum Fourier Transform, operation by operation

Our challenge is to understand how the arrangement of simple QPU operations in Figure 7-24 can extract the frequency components from a signal in the input register.

Explaining the QFT would require thinking about how this circuit acts on an input state looking something like Figure 7-6, where phases vary periodically across the register. This is quite a mind-bending task. We'll take a handily simpler approach and try to explain how the *inverse* QFT works. If we can understand the invQFT, the QFT is simply the reverse.

In the simple case of a four-qubit input, the invQFT has the decomposition shown in Figure 7-25.

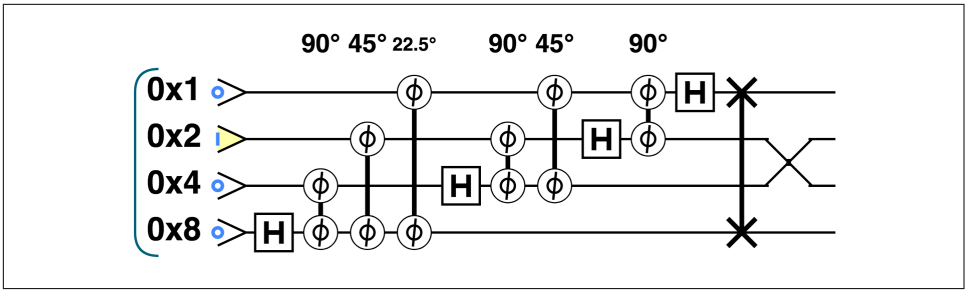


Figure 7-25. Four-qubit inverse QFT

Although this looks very similar to the QFT circuit, crucially the phases are different. Figure 7-26 shows what we get if we apply the preceding circuit to a register containing the value 2.

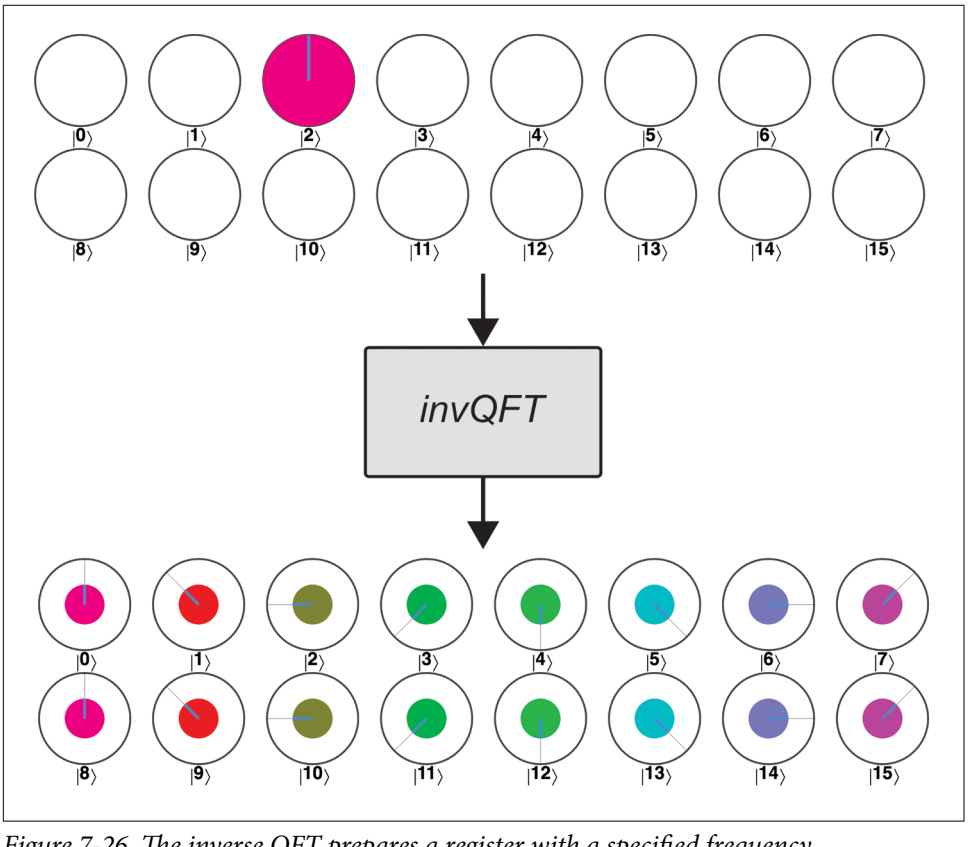


Figure 7-26. The inverse QFT prepares a register with a specified frequency

There's more than one way that the circuits for the QFT and invQFT can be written. We've chosen forms here that are especially convenient for helping us explain the circuit's action. If you see slightly different circuits for these primitives, it's worth checking if they are equivalent.

The Intuition

Suppose we provide the invQFT with an N-qubit input register encoding some integer n. As we imagine scanning along the output register's 2^N values, we need each consecutive value's relative phase to rotate enough to perform a full 360° every $2^N/n$ circles. This means that to produce our desired output state we need each consecutive circle to have its relative phase rotated by an additional 360° × $n/2^N$, as shown in Figure 7-27.

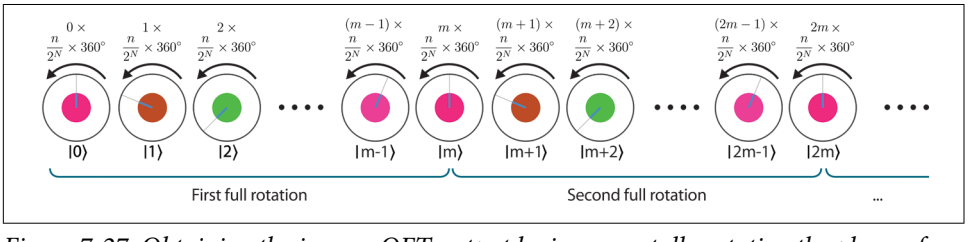


Figure 7-27. Obtaining the inverse QFT output by incrementally rotating the phase of each amplitude in the register

Perhaps this is easier to see in the case of the specific example from Figure 7-26. In this case, since we have a four-qubit register with an input value of 2, we will want to rotate each consecutive value by an angle of 360° × $n/2^N$ = 360° × $2/2^4$ = 45° to get our desired output state. This is shown in Figure 7-28.

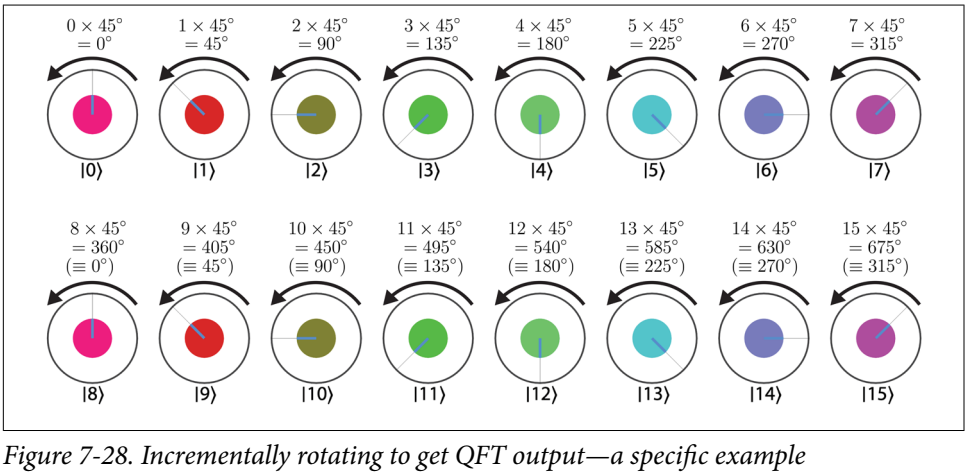


Figure 7-28. Incrementally rotating to get QFT output—a specific example

We see that rotating by this simple rule gets us precisely the required periodicity in the register's relative phases (noting of course that 360° ≡ 0°, 405° ≡ 45°, etc.).

Operation by Operation

The invQFT circuit in Figure 7-25 is just a cleverly compact way of implementing this conditional rotation rule. To help see this, we can split the tasks that the circuit in Figure 7-25 must perform into two separate requirements:

1. Determine the value $\theta = n/2 \times 360°$ (where n is the value initially stored in the register).
2. Rotate the phase of each circle in the register by a multiple of the angle θ found in the previous step, *where the multiple is the circle's decimal value*.

The invQFT actually performs both of these steps at once in an ingeniously compact way. However, we'll tease apart how it performs each individual step to see its operation more clearly. Although it might seem a little backward, it will actually be easiest to start with the second step. How can we apply a `PHASE` where the rotation angle for each value in the register is a multiple of that value?

Rotating each circle's phase by a multiple of its value

When expressing an integer in our QPU register, the 0/1 value of the k^{th} qubit indicates whether there is a contribution of 2^k to the integer's value—just like in a normal binary register. So to perform any given operation on the register *the number of times that is represented in the register*, we need to perform the operation once on the qubit with a weighting of 2^0, twice on the qubit with a weighting of 2^1, and so on.

We can see how this works with an example. Suppose we wanted to perform k lots of 20° rotations on a register, where k is the value stored in the register. We could do so as shown in Figure 7-29.

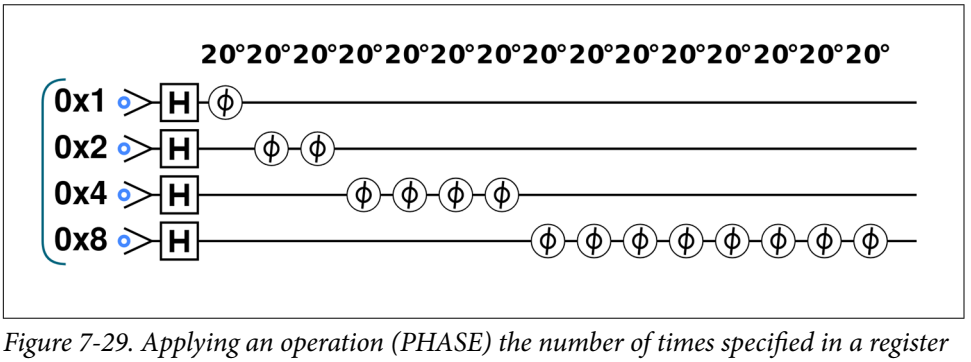


Figure 7-29. Applying an operation (PHASE) the number of times specified in a register

Note that we've performed HAD operations at the start as well, just so that we obtain the result on every possible value of k in superposition. Example 7-8 contains the code we used to generate this circuit.

Sample Code

Run this sample online at *http://oreilly-qc.github.io?p=7-8*.

Example 7-8. QFT rotating phases by different amounts

```
// Rotate kth state in register by k times 20 degrees
var phi = 20;

// Let's consider a 4-qubit register
qc.reset(4);
// First HAD so that we can see the result for all k values at once
qc.write(0);
qc.had();
// Apply 2^k phase operations to kth qubit
for (var i=0; i<4; i++) {
    var val = 1<<i;
    for (var j=0; j<val; j++) {
     qc.phase(phi, val);
    }
}
```

Running Example 7-8, we obtain the k^{th} state rotated by $k \times 20°$, as shown in Figure 7-30. Precisely what we wanted!

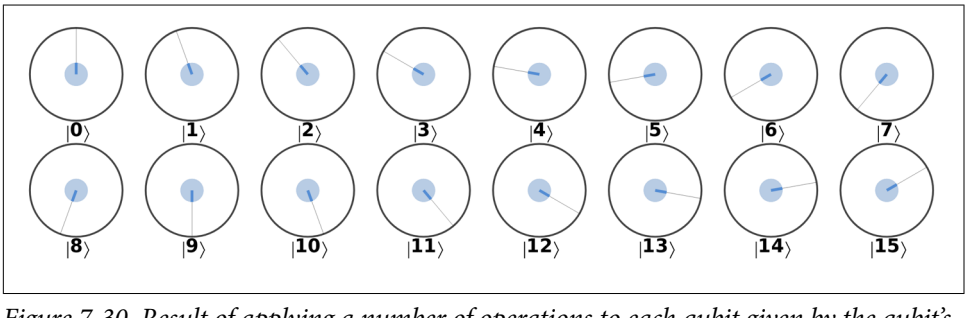


Figure 7-30. Result of applying a number of operations to each qubit given by the qubit's binary weighting

To implement the invQFT we need to perform this trick with the angle we use being $n/2^N \times 360°$ rather than 20° (where n is the initial value in the register).

Conditionally rotating by the angle $n/2^N \times 360°$

Referring back to Figure 7-25, we see that the invQFT consists of four subroutines, shown in Figure 7-31.

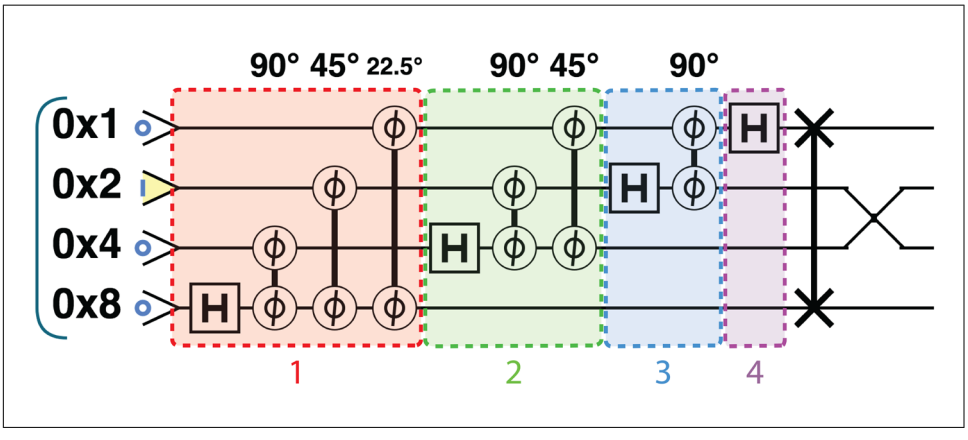


Figure 7-31. The four subroutines of invQFT

Each of these subroutines performs precisely the multiple rotations of increasing weighting that we described in Figure 7-28. The question is, what angle is it that these subcircuits are rotating multiples of? We'll now see that, in fact, the first subcircuit rotates the highest-weight qubit by $n/2^N \times 360°$ while the second rotates the next-highest-weight qubit by 2^1 times this value, and so on, as shown in Figure 7-32—precisely as prescribed by Figure 7-28.

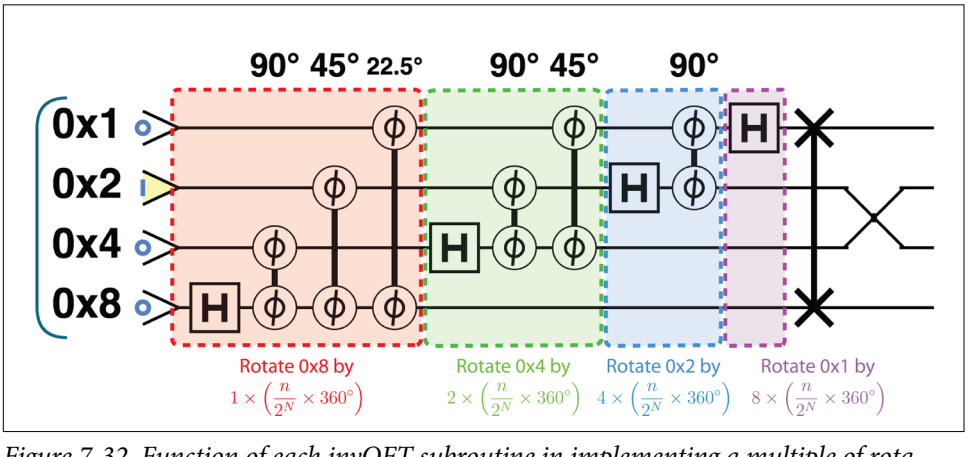


Figure 7-32. Function of each invQFT subroutine in implementing a multiple of rotations specified by input register

If you're following carefully you'll notice that this actually applies the rotations that Figure 7-28 specifies in the reverse order—but we'll see how to simply deal with this problem shortly.

Consider the first subcircuit from Figure 7-32, shown in more detail in Figure 7-33. We can confirm that it performs the stated rotation of $n/2^N \times 360°$ on the highest-weight (`0x8`) qubit.

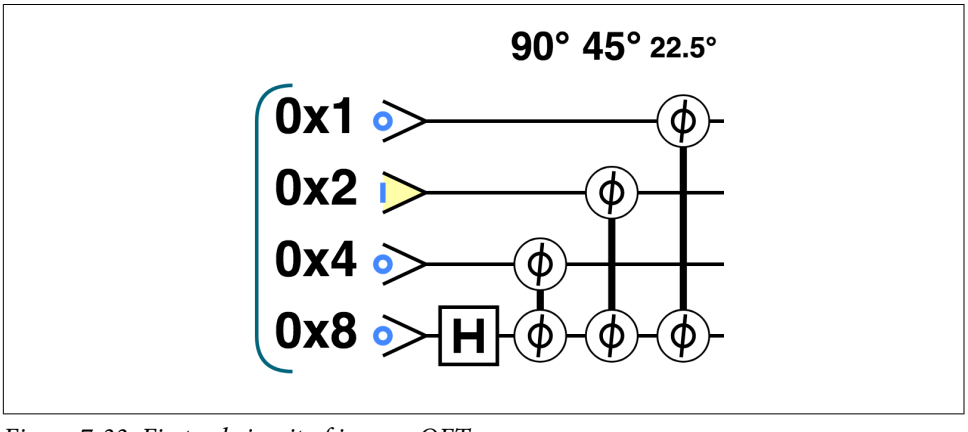


Figure 7-33. First subcircuit of inverse QFT

Each `CPHASE` in this subroutine conditionally rotates the `0x8` qubit by an angle that is the same proportion of 360° as the condition qubit is of 2^N. For example, the `CPHASE` acting between the `0x4` and `0x8` qubits in Figure 7-33 rotates the highest-weight qubit by 90°, and $4/2^4 = 90°/360°$. In this way we are building up the rotation $n/2^N \times 360°$ on the `0x8` qubit through each component of its binary expansion.

But what about the highest-weight qubit? This binary expansion should also require performing a conditional relative phase rotation of 180° *on* the highest-weight qubit, *dependent* on the value of the highest-weight qubit. The `HAD` operation in Figure 7-33 does precisely this for us (to see this, simply recall the defining action of `HAD` on the $|0\rangle$ and $|1\rangle$ states). This `HAD` also serves another purpose: generating the superposition of this qubit from the register as required in Figure 7-28. See what we meant when we said this circuit is cleverly compact?

Each subsequent subcircuit from Figure 7-32 effectively performs a *roll left*, shifting up the weight of the angle associated with each qubit (so that, for example, in subcircuit 2, the `0x2` qubit is associated with a rotation of 90°, rather than 45°). As a result, each subcircuit multiplies the phase of $n/2^N \times 360°$ that it applies by 2 before imparting it to the particular qubit it acts on—just as required in Figure 7-32.

Taken together, this performs the conditional rotations that we need for the invQFT—but with one problem. Everything is upside-down! Subcircuit 1 rotates the `0x8`